ABSTRACT INTERPRETATION

OF

LOGIC PROGRAMS

WITH

EXTERNAL FUNCTIONS

–

AN IMPLEMENTATION

Jan Alexandersson

Master's Thesis in Computer and Information Science

at

Linköping University
S-581 83 Linköping, Sweden

August 1993

Reg nr: LiTH-IDA-Ex-9323

Examiner: Jan Małuszyński, Linköping University
Thesis Supervisor: Johan Boye, Linköping University

"I'm trying to remember if I am confused"

Dr. Elizabeth Hinkelman

Contents

Acknowledgements

When writing this thesis I have received a lot of help from a lot of people.

To start with I want to express my warmest gratitude to prof. Jan Małuszyński for allowing me to write my thesis at the logpro group as well as for reading my drafts carefully and giving me invaluable suggestions and comments.

Thanks also to Johan Boye, my supervisor and friend for discussions and inspiration.

I also ow the following people a big “thank you” for helping me with various of things: Ulf Nilsson—for explaining his frame work, Andreas Kågedal—for invaluable help when implementing my ideas, Lars Degerstedt—for discussions and help while I was trying to understand the world of logic programming. Michael Hanus—for explaining parts of his paper to me. Stephen Spackman also read one of my final drafts.

Finally: Thank you Fredrik for being my opponent.

Saarbrücken
August 1993

Jan Alexandersson

Chapter 1

Introduction

In the recent years a lot of work has been put into the task of integrating functions into logic programming languages. Looking at the operational semantics, the languages developed can be divided into two main classes. The first contain completely new languages with complete operational semantics using *narrowing techniques*. These languages allow definitions of both functions and relations. Ex: EQLOG [GM86], SLOG [Fri85], K-LEAF [BGL+87], BABEL[MR92], ALF[Han92]. The other class contains attempts to integrate a logic programming language with an existing functional language. These languages use modified unification algorithms which reduce terms into a normal form. Ex: LOGLISP [RS82], QLOG [Kom82], POPLOG [MH84] and APPLOG [Coh86]. Functional expressions are evaluated before unification with another term if possible. In Le Fun [AKLN87] the call is delayed until the arguments eventually become fully instantiated.

[BM88] introduces a semantically clean way of amalgamating logic programs with external functional procedures written in any language. In this approach functions are regarded as "black boxes" that reduce to a ground term (when called with fully instantiated terms). The language, GAPLog, uses an incomplete unification algorithm *S-unification* [BM88, Bon89], with three possible outcomes: **success**, **fail** and **"don't know"**. The "don't know" case occurs when S-unification encounters equations whose solvability cannot be determined. S-unification is used in the *S-SLD-resolution* procedure [Boy91b], which provides an operational semantics for logic programs with external procedures. The current implementation of the language GAPLog is based on this semantics [KK91].

The idea of S-SLD-resolution is to use the concept of *delay* in order to improve the operational completeness of programs. When S-unification returns "don't know" in its attempt to unify a set U of equations, this is because U contains certain equations (called *constraints*) which contain non-ground function calls. These constraints are passed on to the next resolution step (that is, unification is partly delayed), in hope that further instantiation will ground the function calls.

In the GAPLog compiler (compiling GAPLog programs into SICStus Prolog), this is effected by means of a predicate `gfreeze/2`. `gfreeze/2` evaluates its second argument if the first argument is ground. If the first argument is not ground, it waits until the first argument eventually becomes ground. To each function call, a call to `gfreeze/2` is added, and hence the test for groundness is done dynamically.

It would however be desirable to determine at compile-time that certain variables will always be ground when the execution reaches a call to an external function. If one can ensure that the arguments to a function are always fully instantiated when the execution reaches the function call, the dynamic check for that variable can safely be removed.

The following "benchmark" test highlights the effect, in execution speed, of removing the dynamic check for a small program.

Consider the following GAPLog program relating `A` to the `A`'th fibonacci number.

```
f(1,1).
f(2,1).
f(A,B+A) :-
     f(A-1,B).
```

...which compiles[1] into the equivalent Prolog program (program 1):

```
f(1,1).
f(2,1).
f(A,C) :-
     gfreeze([A,B], fun_call(B+A,C)),
     gfreeze(A, fun_call(A-1,D)),
     f(D,B).
```

[1]The process of compiling a GAPLog program to an equivalent SICStus Prolog program is described in chapter 2

We see that If `f` is called with the first argument ground the first call to `gfreeze` can be simplified, since `A` will always be ground – that is instantiated to a ground term – when the execution reaches the first body atom. For the same reason (that `A` always is ground) the second call to `gfreeze/2` can be removed. This gives the following program (program 2).

```
f(1,1).
f(2,1).
f(A,C) :-
     gfreeze(B, fun_call(B+A,C)),
     fun_call(A-1,D),
     f(D,B).
```

If we want to remove the last call to `gfreeze` we have to reorganize the body atoms by putting the first body-atom last in the body. Since `B` is (always) ground after the recursive call to `s`, we can remove the call to `gfreeze` yielding program 3:

```
f(1,1).
f(2,1).
f(A,C) :-
     fun_call(A-1,D),
     f(D,B),
     fun_call(B+A,C).
```

The timings can be seen in fig 1.1. By removing all calls to `gfreeze` we reduce the execution time to 55% of the execution time of the "fully gfreezed" program.

Although this is a "worst case" example with two calls to an external function for every recursive call, there is clearly time to be saved by removing unnecessary calls to `gfreeze`. This can be done by performing data-flow analysis at compile-time. The objective of this thesis is to develop (and implement) a compile-time analysis of GAPLog programs in order to detect points in the programs where the dynamic check can safely be removed. This will be done by performing a groundness analysis of the programs.

The technique to be used is called *abstract interpretation*, which has received a lot of attention in recent years. Several abstract interpretation frameworks for logic programming languages have been developed (see for instance [Bru87] [JS87] [Mel87]). In this thesis we will use the framework of

Ulf Nilsson as described in [Nil92].

The rest of this paper is organized as follows: in chapter 2 we introduce the language GAPLog. It contains A brief description of the theoretical basis of the language as well as a look at how the compiler works. Chapter 3 presents the concept of abstract interpretation as defined by Ulf Nilsson [Nil92]. This is used in the succeeding two chapters. The first, chapter 4, develops tools capable of inferring runtime properties of pure Prolog programs. This chapter is based on the abstract interpretation presented in [Nil92]. In chapter 5 we extend this work so that we are able to analyze GAPLog programs. This chapter is closely related to [Han92], which however concentrates on the problem of detecting operational completeness of residuating logic programs. It turns out that a (slightly modified) subset of the domain developed in that paper can be used in groundness analysis. Chapter 6 discusses the problem of computing abstract interpretations. and different iteration schemas are described and compared by means of a small benchmark. The results of the experiment confirms the informal complexity analysis done in [Nil92]. Finally the thesis is concluded in chapter 7.

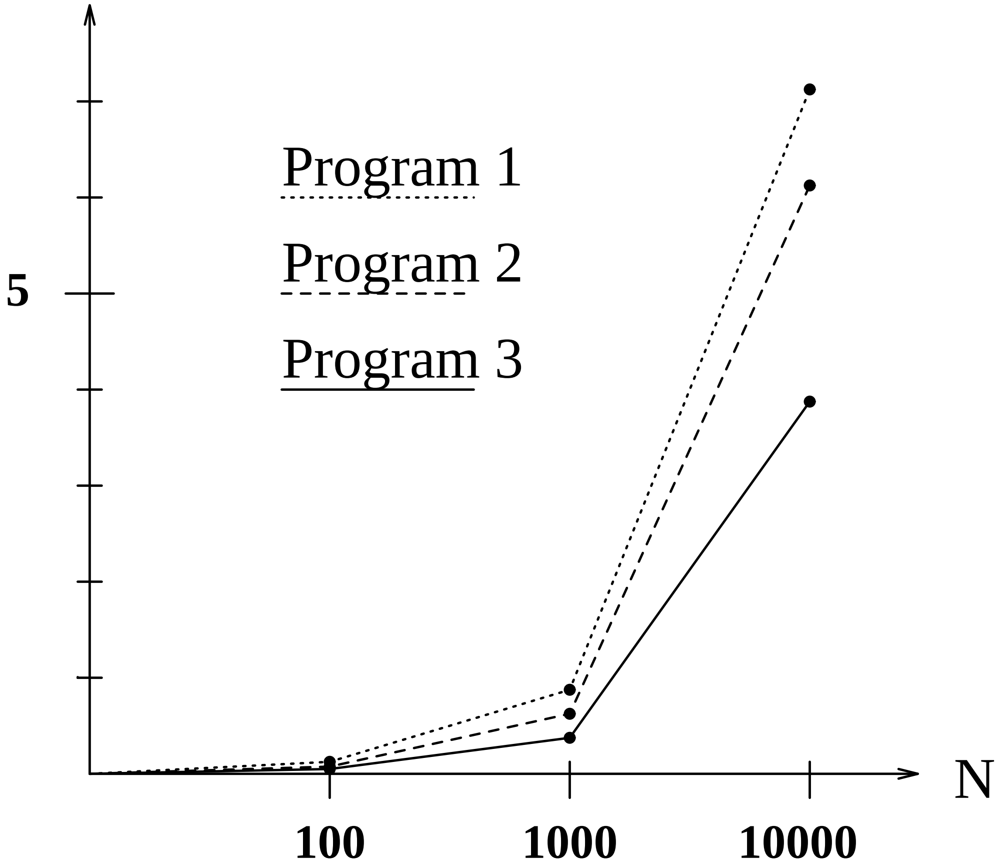

Figure 1.1: Benchmark results

Chapter 2

The language GAPLog

This chapter outlines the operational semantics of the language GAPLog and describes the compilation of GAPLog into SICStus Prolog. The formal basis of the language, the declarative semantics, will not be presented. The interested reader is referred to [MB+93].

For the rest of the thesis the reader is assumed to be familiar with basic discrete mathematics [Bä86] and logic programming [NM90]. Familiarity with lattice theory and the Tarski fixpoint theorem [Tar55] will also be helpful.

The GAPLog language provides a clean approach to integrating functions written in *any* language into logic programs. The only restriction on the functions is that when given ground terms as input, they can be called and will return a ground term. This approach is supported by many arguments. Significant among them is that we are able to give logic programmes with external functions a declarative reading. This is obtained by viewing the programs as equational logic programs.

The syntax of the GAPLog language is similar to the syntax of Prolog. GAPLog however allows the user to call external functions at any place in the program where a variable may appear. Some external functions are predefined (such as `+/2` – addition, `*/2` – multiplication, `//` – integer division,...); although they receive the natural interpretations, the compiler makes no assumptions about their behaviour. The following program is an example of a GAPLog program that relates a list to its length.

```
len([], 0).
len([_X|R], 1+Length) :-
        len(R, Length).
```

■ The operational semantics

The operational semantics is based on an incomplete equational unification algorithm, called S-unification[1]. The S-unification algorithm[2] tries to rewrite a set of equations until a normal form is yielded. A set of equations is in solved form iff:

- every left hand side is a variable, and
- the equations can be ordered as $X_1 \doteq t_1, \ldots, X_n \doteq t_n$, so that if a variable X_i occurs in t_j then $i > j$.

The S-unification procedure is given as a set of rules, which can be applied non-deterministically to a set U of equations. Basically, S-unification consists of the Martelli-Montanari rules [MM82] for syntactic unification, a rule for replacing a ground function call $f(s_1, \ldots, s_n)$ with s when $f(s_1, \ldots, s_n)$ reduces to s, and some auxiliary rules for handling defined symbols.

When given a set U of equations to unify, the algorithm transforms U by applying the rules (in any order) until no rule is applicable. Depending on the result of the transformation, it then gives the following result:

1. $\emptyset$, if the transformation ends in failure. (Fail)
2. $\{\sigma_{U'}\}$, if the transformation ends in a solved form U'.
3. "don't know", if neither 1 nor 2 applies.

■ An implementation

The implementation of S-SLD resolution in the GAPLog system is done by compilation of the GAPLog programs into SICStus Prolog. The compiled

[1] as in Structure-oriented

[2] The algorithm can be found in the appendix.

program may use external functions written in C. The compilation consists of *flattening* the program and then adding delays to the external function calls extracted by flattening[3].

The process of flattening a program is performed as follows. First "lift out" all occurrences of defined (or external) functional terms from the atom and replace them by a variable. Then build an equation of the variable and the functional term, and finally add the equation to the left or to the right of the atom depending on if the atom is a head or a body atom.

More formally:

- Let `H :- B` be a GAPLog clause. Then repeatedly do the following:
- Lift out all occurrences of defined symbols $\mathtt{f}_i$ in `H` and replace them by variables $\mathtt{F}_i$. Add equations $\mathtt{F}_i \doteq \mathtt{f}_i$ immediately after the head.
- For all body atoms in `B` do the same thing, but add the equation immedeately *before* each body atom.

The next step is to add a dynamic check to each function call that guarantees that the arguments to the function call are instantiated to ground terms. This is ensured by means of the predicate `gfreeze/2`, which calls its second argument only if all variables in the first argument are instantiated to ground terms. Otherwise it freezes until (eventually) all variables are properly instantiated.

To get some intuition of how the compiler works, we show an example of a GAPLog program and the result of applying the GAPLog compiler to it.

Example 2.1
Let `succ/1, +/2, -/2` and `*/2` be defined symbols (with their natural interpretations) in the following program:

```
foo(A,succ(A+B)):-
        fie(A - 1*A).
```

Then the GAPLog compiler transforms the program into the following equivalent Prolog program:

```
foo(A, F) :-
        gfreeze(C, fun_call(succ(C),F)),
```

[3]The interested reader will get a comprehensive presentation of the details of the implementation when reading [Kåg92] or [MB+93].

```
gfreeze([A,B], fun_call(A+B,C)),
gfreeze(A, (fun_call(1*A,E),fun_call(A-E,D))),
fie(D).
```

For typographical reasons we will, when analysing the programs, use the following shorthand:

```
foo(A, F) :-
    succ(C) = F,
    A + B = C,
    1 * A = E,
    A - E = D,
    fie(D).
```

This is obtained by first rewrite the third atom of compiled program to the following equivalent atoms:

```
gfreeze(A, fun_call(1*A,E))
gfreeze([A,E], fun_call(A-E,D))
```

and then substitute the `gfreeze` with the appropriate equations. For instance is "`gfreeze(A, fun_call(1*A,E))`" replaced by "`1 * A = E`".

□

We will in what follows only consider flattened programs.

Chapter 3

Abstract interpretation

This chapter surveys Ulf Nilsson's framework for abstract interpretation [Nil92] with the object of applying it to groundness analysis of GAPLog programs.

Abstract interpretation is a formal way of performing data-flow analysis of programs. The method was originally developed for imperative languages, but has also been adapted to the logic programming paradigm.

All abstract interpretation frameworks have one thing in common, namely a way of describing all possible program states. This is usually referred to as the *base interpretation* (or collecting function). Although they all have a base interpretation, the way the base interpretation is defined differs. The base interpretation consists of a *domain*, D, typically a lattice, and a (monotone) *semantic function*, Δ, defined on the domain. An element in the domain is a set of program states, and the semantic function computes a new element in the domain given an old one. The *least fixpoint* (LFP) of the semantic function is the set of all states reachable given a set of initial ones ($Init$). The LFP tells us exactly how the program behaves, and hence it could be used to draw conclusions about the behaviour of the program at compile-time.

However, the base interpretation is usually not computable, since for most interesting programs it is infinite, so we need a way of approximate it to be able to say something about the behaviour of the program. This is done by defining an abstract domain $\widetilde{D}$, an abstract semantic function $\widetilde{\Delta}$ and a so-called concretization function $\gamma\colon \widetilde{D} \to D$.

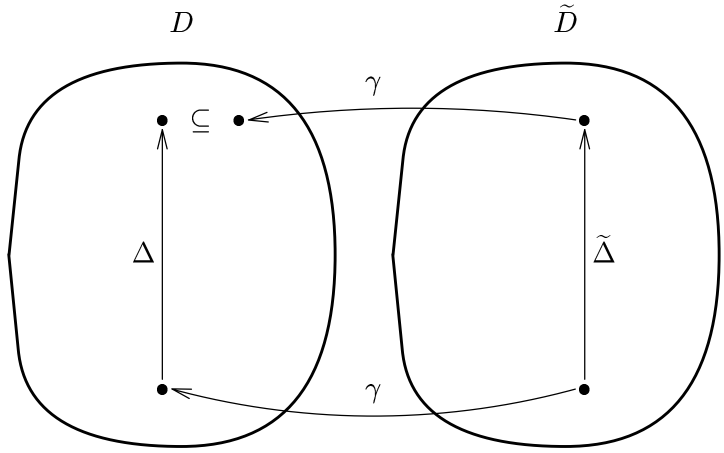

Figure 3.1: Conservative γ-approximation

If:

- γ is monotone, and
- for each $\widetilde{x} \in \widetilde{D}$:
$$\Delta(\gamma(\widetilde{x})) \subseteq \gamma(\widetilde{\Delta}(\widetilde{x}))$$
(see Figure 3.1)

then:
$$LFP(\Delta) \subseteq \gamma(LFP(\widetilde{\Delta}))$$

If an abstract interpretation is defined as described above it is said to be *conservative*.[1] The goal will be to define $\langle \widetilde{D} \ ; \ \widetilde{\Delta} \rangle$ and γ in such a way that, given a set of initial states, $Init$, the corresponding value in the abstract domain, $\widetilde{Init}$, makes $\gamma(LFP(\widetilde{\Delta}(\widetilde{Init})))$ come as close to $LFP(\Delta(\gamma(Init)))$ as possible. The closer we get, the more interesting things we are able to say. For each property one wants to analyse, a new abstract interpretation $\langle \widetilde{D} \ ; \ \widetilde{\Delta} \rangle$ together with a concretization function has to be defined.

There are many abstract interpretation frameworks around in the area of logic programming (See for instance [AH87], [Bru87]). In this thesis we

[1] Another term used for this is "safe".

will use one by Ulf Nilsson as defined in [Nil92][2], and in this chapter we will review some of the most important parts of his framework. Some things are intentionally skipped, so for a full presentation the reader is encouraged to read [Nil92]. This chapter is organized as follows. The first section provides the basis for the base interpretation, which is defined in the second section. Section three describes a formal method of approximating the abstract interpretation by defining a *Galois connection.* The last part of the framework—a way of computing the abstract interpretation is discussed in chapter 6, but a simple approach, used in the following two chapters, is described in section 4. A short description of *finite depth and solved form abstraction* ends the chapter.

3.1 The framework

The framework consists of three components:

- A fixpoint characterisation of the run-time behaviour of the program when executed by an interpreter, called the *base interpretation.*
- A way of *approximating* the base interpretation by means of an abstract interpretation.
- A way of *efficiently computing* the approximation.

■ Abstract interpretation

We first give a formal definition of abstract interpretation.

Definition 3.1 An *abstract interpretation* for a program P is a pair

$$\langle D_P \, ; \, \Delta_P \rangle$$

where D_P, called the *domain*, is a complete lattice and $\Delta_P : D_P \rightarrow D_P$, called *the semantic* function, is monotone. □

[2]This PhD thesis contains an extension of material previously presented in [Nil90], [Nil89a] and [Nil89b].

The domain is related to the operational semantics of the program; its elements characterize sets of the execution states of the program. The second component, Δ_P, defines the meaning of the program by $LFP(\Delta_P)$. We will first consider the problem of building the domain. This will be done by first developing a formal description of S-SLD-resolution by means of a transition system. Based on that we then define the base interpretation.

■ Program points

An important ingredient of one of the parts of the transition system, the transition relation, is the notion of program point. We will need "places" to collect the information—the space in between the atoms of the program.

Let C be the clause `a :- b, c, d.` then

- The leftmost point of the clause, ${}_\bullet C$, is called the *entry point* of C
- The rightmost point of the clause, $C_\bullet$, is called the *exit point* of C
- $\mathtt{b}_\bullet$ is called the *success point* of `b`
- ${}_\bullet\mathtt{b}$ is called the *calling point* of `b`
- Finally ${}_\bullet C = {}_\bullet\mathtt{b}$, $\mathtt{b}_\bullet = {}_\bullet\mathtt{c}$, $\mathtt{c}_\bullet = {}_\bullet\mathtt{d}$, $\mathtt{d}_\bullet = C_\bullet$,

If the clause is a fact, the entry point will be the same as the exit point. Note that the notion of success point also applies to the case where the atom is an equation. Note also that every point in the program falls into exactly two of these categories. The set of program points in a program will be denoted "Point".

■ Substitutions and Renamings

Definition 3.2 A *substitution* is a finite set $\{x_1/t_1, \ldots, x_n/t_n\}$ where

- $x_1, \ldots, x_n$ are distinct variables, and
- t_i is a term such that $x_i \neq t_i$ for $i \in \{1, \ldots, n\}$

□

Definition 3.3 A *renaming* is a substitution such that

- $dom(\phi_i) = var(P)$ for $i \geq 0$
- if $i \neq j$ then $range(\phi_i) \cap range(\phi_j) = \emptyset$
- if $X, Y \in var(P)$ then $\phi_i(X) = \phi_i(Y)$ iff $X = Y$ (for $i \geq 0$)

$\square$

■ Abstract syntax

The following abstract syntax will be used
Basic sets:

$$\begin{array}{llll} \textit{Equations} & e & \in & \text{Eqn} \\ \textit{Renamings} & \phi & \in & \text{Renaming} \\ \textit{Points} & p & \in & \text{Point} \\ \textit{Naturals} & i, j & \in & \text{Nat} \; = \; \{0,\ 1,\ 2, \ldots\} \end{array}$$

Derived sets:

$$\begin{array}{llll} \textit{Substitutions} & \sigma, \theta & \in & \text{Subst where} \\ & \sigma, \theta & ::= & \{e_1, \ldots e_n\} \quad (n \geq 0) \\ \textit{Goals} & g & \in & \text{Goal where} \\ & g & ::= & (\sigma, p, \phi)^+ \end{array}$$

Intuitively a goal of the form

$$(\sigma_n, p_n, \phi_n) :: \ldots :: (\sigma_0, p_0, \phi_0)$$

is an abstract description of a stack of activation records.

Example 3.4 Consider the following program together with a goal:

$$\texttt{:-}_1\texttt{p(X)}_2$$

$$\begin{array}{l} \texttt{p(X)}_3\texttt{:-} \\ \quad \texttt{q(X,y)}_4. \end{array}$$

$$\texttt{q(f(X),X)}_5.$$

Then the following goal represents the "memory" of an interpreter after first "calling" the procedure `p/1` and then the procedure `q/2`:

$$(\{\mathtt{f}(\mathtt{X}_2) \doteq \mathtt{X}_1, \mathtt{X}_2 \doteq y\}, 5, \phi_2) :: (\{\mathtt{X}_1 \doteq \mathtt{X}_0\}, 4, \phi_1) :: (\epsilon, 2, \phi_0)$$

□

Definition 3.5 Let $(\sigma_n, p_n, \phi_n) :: \ldots :: (\sigma_0, p_0, \phi_0)$ be a goal. We say that p_n is the *current* program point in the goal. □

Next we develop a formal way of describing S-SLD-resolution. This is done by defining a transition system.

■ Transition system

Definition 3.6 A *transition system* is a pair $\langle\, \Gamma \;\; ; \;\; \vdash \,\rangle$ where

- Γ is a set of configurations
- $\vdash \subseteq \; \Gamma \times \Gamma$ is a *transition relation*

□

A configuration in our case is a pair consisting of a goal and a natural number. Intuitively one can think of the configuration as a state of the interpreter.

■ Configurations

Definition 3.7 A *configuration* Γ is a pair

$$\Gamma \;\; := \;\; \langle g \; ; \; i \rangle$$

where g is a goal and $i \in Nat$. □

i is used for renaming purposes.

■ The transition relation

Defining the transition relation we have three cases to consider, since we are dealing with flattened programs. The first case is when the atom to the right of the current point is an equation. The second case is when the current program point is a calling point, and the third when the current point is an exit point.

Definition 3.8 The *transition relation* $\vdash$ is defined by the following three rules:

Equation

$$\frac{\langle(\sigma_n, {}_\bullet\alpha, \phi_n) :: \ldots :: (\sigma_0, p_0, \phi_0)\ \ ;\ i\rangle}{\langle(\{\phi_n(\alpha)\} \cup \sigma_n, \alpha_\bullet, \phi_n) :: (\sigma_0, p_0, \phi_0)\ \ ;\ i\rangle}$$

if α is an equational atom, and $\phi_n(\alpha) \cup \sigma_n\ \ldots\ \sigma_0$ is S-SLD-satisfiable. By S-SLD-satisfiable we mean that the S-SLD-resolution algorithm should not return fail when applied to the set [Boy91b].

Procedure call

$$\frac{\langle(\sigma_n, {}_\bullet\alpha, \phi_n) :: \ldots :: (\sigma_0, p_0, \phi_0)\ \ ;\ i\rangle}{\langle(\{\beta \doteq \phi_n(\alpha)\}, {}_\bullet C, \phi_{n+1}) :: (\sigma_n, \alpha_\bullet, \phi_n) :: \ldots :: (\sigma_0, p_0, \phi_0)\ \ ;\ i+1\rangle}$$

where β is the head of $\phi_{i+1}(C)$, and $\beta \doteq \phi_n(\alpha)\ \cup\ \sigma_n\ \ldots\ \sigma_0$ is S-SLD-satisfiable.

Procedure exit

$$\frac{\langle(\sigma_n, C_\bullet, \phi_n) :: (\sigma_{n-1}, p_{n-1}, \phi_{n-1}) :: \ldots :: (\sigma_0, p_0, \phi_0)\ \ ;\ i\rangle}{\langle(\sigma_n \cup \sigma_{n-1}, p_{n-1}, \phi_{n-1}) :: \ldots :: (\sigma_0, p_0, \phi_0)\ \ ;\ i\rangle}$$

□

■ Derivation

Finally we can describe the S-SLD-resolution in terms of derivation. We assume, without loss of generality, that all goals are of the form "`:-` α`.`", where α is an atom. Hence the goal can be viewed as a part of the program, and the notion of program point also applies to goals. A derivation is a sequence of configurations starting with

$$\langle(\sigma_0, {}_\bullet\alpha, \phi_0)\ ;\ 0\rangle$$

where α is the goal atom. Each new step is obtained from the previous one by a transition step via $\vdash$.

Example 3.9 Consider the program in example 3.4. Then we have the (only) derivation:

$$\frac{\frac{\frac{\frac{\langle(\epsilon, 1, \phi_0)\ ;\ 0\rangle}{\langle(\{\mathtt{X}_1 \doteq \mathtt{X}_0\}, 3, \phi_1) :: (\epsilon, 2, \phi_0)\ ;\ 1\rangle}}{\langle(\{\mathtt{f(X}_2\mathtt{)} \doteq \mathtt{X}_1, \mathtt{X}_2 \doteq \mathtt{y}\}, 5, \phi_2) :: (\{\mathtt{X}_1 \doteq \mathtt{X}_0\}, 4, \phi_1) :: (\epsilon, 2, \phi_0)\ ;\ 2\rangle}}{\langle(\{\mathtt{f(X}_2\mathtt{)} \doteq \mathtt{X}_1, \mathtt{X}_2 \doteq \mathtt{y}, \mathtt{X}_1 \doteq \mathtt{X}_0\}, 4, \phi_1) :: (\epsilon, 2, \phi_0)\ ;\ 2\rangle}}{\langle(\{\mathtt{f(X}_2\mathtt{)} \doteq \mathtt{X}_1, \mathtt{X}_2 \doteq \mathtt{y}, \mathtt{X}_1 \doteq \mathtt{X}_0\}, 2, \phi_0)\ ;\ 2\rangle}$$

□

3.2 The base interpretation

The meaning of the program is the set of all reachable configurations given a set of initial ones.

Definition 3.10 Let *Init* be a set of initial configurations. The *meaning* of a program P is the least solution to the equation:

$$X = \mathit{Init}\ \cup X \cup \{c_2\ \mid\ c_1 \in X\ \wedge\ c_1 \vdash c_2\}$$

□

Now define Δ in the following way:

$$\Delta := \lambda X.(\mathit{Init} \ \cup X \cup \{c_2 \ \mid \ c_1 \in X \ \wedge \ c_1 \vdash c_2\})$$

Then $\Delta : 2^{\Gamma} \rightarrow 2^{\Gamma}$ is monotone and hence the least fixpoint always exists [Tar55].

Since 2^{Γ} is a complete lattice under set-inclusion, $\langle 2^{\Gamma} \ ; \ \Delta \rangle$ is an abstract interpretation.

The intuition behind Δ is that each time we operate with Δ we collect more information. If the program includes recursive clauses we will collect an infinite number of sets of configurations.

However, the definition given above does not facilitate efficient computing of the meaning of a program so we need a new equivalent one. The meaning of a program is a solution to the equation given above. We can reformulate this by looking at the program points instead. If we say that the meaning of a program point is the union of all possible ways to get to that particular program point, the meaning of the program will be the collected meaning of all program points.

For this we define the call graph.

Definition 3.11 Let H be the set of all head atoms, and B the set of all body atoms in P. The *call graph* of P is a relation $\rightsquigarrow \subseteq (B \times H)$ where:

$$\alpha \ \rightsquigarrow \ \beta \quad \text{iff} \quad \alpha \in B, \ \beta \in H \quad \text{and} \quad mgu(\phi_0(\alpha), \phi_1(\beta))^3 \ \neq \bot$$

□

■ Entry Points

The meaning $X_{\bullet C}$ of an entry point $\bullet C$ is defined in terms of the calling points of the atoms able to invoke C. That is all atoms related to $\bullet C$ via the *call graph*

$$X_{\bullet C} = \bigcup_{\alpha \rightsquigarrow C} \{c_2 \ \in \ \Gamma_{\bullet C} \ \mid \ c_1 \ \in \ X_{\bullet \alpha} \ \wedge \ c_1 \vdash c_2\}$$

or short

$$X_{\bullet C} = \bigcup_{\alpha \rightsquigarrow C} \mathrm{Entry}(X, \alpha, C)$$

[3] We assume that *mgu* computes the most general unifier of the two atoms

■ Success Points

In defining the meaning of a success point we have two cases to consider. Possibly it is an success point of an atom calling some clause(s) via the call graph. The meaning $X_{\alpha_\bullet}$ of a point is then defined in terms of the exit points of the clauses invoked by α:

$$X_{\alpha_\bullet} = \bigcup_{\alpha \rightsquigarrow C} \{c_2 \in \Gamma_{\alpha_\bullet} \mid c_1 \in X_{C_\bullet} \wedge c_1 \vdash c_2\}$$

or short

$$X_{\alpha_\bullet} = \bigcup_{\alpha \rightsquigarrow C} \mathrm{Succ}(X, \alpha, C)$$

Otherwise the meaning $X_{\alpha_\bullet}$ of the success point is defined in terms of the "calling point" of the equation.

$$X_{\alpha_\bullet} = \{c_2 \in \Gamma_{\alpha_\bullet} \mid c_1 \in X_{\bullet\alpha} \wedge c_1 \vdash c_2\}$$

In this case we have no clause related to the point, so we use the following shorthand:

$$X_{\alpha_\bullet} = \mathrm{Succ}(X, \alpha)$$

■ The Base Interpretation

To define the base interpretation we introduce an auxiliary notion of the domain, D, the product of the domains of each program point, Π^n. Δ can also be partitioned as follows: each point is either a calling point (i) or an exit point (ii), (iii) represents the information given by the user.

$$(i) \qquad \lambda X. \bigcup_{\alpha \leadsto C} \text{Entry}(X, \alpha, C)$$

$$(ii) \qquad \lambda X. \bigcup_{\alpha \leadsto C} \text{Succ}(X, \alpha, C)$$

or

$$\lambda X. \text{Succ}(X, \alpha)$$

$$(iii) \qquad \lambda X. \pi \text{ where } \pi \in \Pi$$

Now, the call graph induces a set of equations of the following form:

$$\begin{array}{rcl} X_1 & = & \Delta_1(X_1, \ldots, X_n) \\ X_2 & = & \Delta_2(X_1, \ldots, X_n) \\ & \vdots & \\ X_n & = & \Delta_n(X_1, \ldots, X_n) \end{array}$$

The meaning of the program can then be defined to be the solution (the least fixpoint) of the equation $\Delta : \Pi^n \to \Pi^n$ defined by:

$$\begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{pmatrix} \overset{\Delta}{\mapsto} \begin{pmatrix} \Delta_1(X_1, \ldots, X_n) \\ \Delta_2(X_1, \ldots, X_n) \\ \vdots \\ \Delta_n(X_1, \ldots, X_n) \end{pmatrix}$$

■ An example

To get a picture of what is going on we show an example of the equations induced by a (well known) program which relate `X` to `Y` if `Y` is `X`!.

Example 3.12 Consider the following program:

C_1: $\texttt{:-}_1\texttt{fac(X,Y)}_2$

C_2: `fac(0,1)`$_3$.
C_3: `fac(A,B)`$_4$`:-`

```
    B = A * C5,
    E = A - 16,
    fac(E,C)7.
```

Then the following set of equations is induced, if we call `fac/2` with the first argument ground:

$$
\begin{array}{rcl}
\overline{X}_1 & = & \{(\{\mathtt{X}_0 \doteq t\}, 1, \Phi_0) \mid \mathit{ground}(t)\} \\
\overline{X}_2 & = & \mathit{Succ}(\overline{X}, \mathtt{fac}(\mathtt{X},\mathtt{Y}), C_1) \cup \mathit{Succ}(\overline{X}, \mathtt{fac}(\mathtt{A},\mathtt{B}), C_3) \\
\overline{X}_3 & = & \mathit{Entry}(\overline{X}, \mathtt{fac}(\mathtt{X},\mathtt{Y}), C_2) \cup \mathit{Entry}(\overline{X}, \mathtt{fac}(\mathtt{E},\mathtt{C}), C_2) \\
\overline{X}_4 & = & \mathit{Entry}(\overline{X}, \mathtt{fac}(\mathtt{X},\mathtt{Y}), C_3) \cup \mathit{Entry}(\overline{X}, \mathtt{fac}(\mathtt{E},\mathtt{C}), C_3) \\
\overline{X}_5 & = & \mathit{Succ}(\overline{X}, \mathtt{B} = \mathtt{A} * \mathtt{C}) \\
\overline{X}_6 & = & \mathit{Succ}(\overline{X}, \mathtt{E} = \mathtt{A} - \mathtt{1}) \\
\overline{X}_7 & = & \mathit{Succ}(\overline{X}, \mathtt{fac}(\mathtt{E},\mathtt{C}), C_1) \cup \mathit{Succ}(\overline{X}, \mathtt{fac}(\mathtt{E},\mathtt{C}), C_3)
\end{array}
$$

□

An important part of an abstract interpretation is the definition of the *concretization function*. It is used to prove the correctness of the abstract interpretation. This thesis does not include a formal proof but the following section is included to complete the framework.

3.3 Approximating the base interpretation

While the base interpretation provides the most precise characterisation of program behaviour, it is not directly useful to us. The reason for this is that it is often infinite and therefore not effectively computable. Hence we need some way to approximate the base interpretation. This can be done by defining a Galois connection between the base interpretation $\langle D \; ; \; \Delta\rangle$ and the abstract interpretation $\langle \widetilde{D} \; ; \; \widetilde{\Delta}\rangle$ we want to use. For practical and computational reasons the abstract interpretation is a finite object.[4] To realize abstract interpretation it suffices to construct abstract versions of

[4]This is not always necessary since there exists techniques for computing with infinite objects. They are often referred to as *widening—narrowing* - techniques.

the three operations $Succ$, $Entry$ and $\cup$.

■ The Galois connection

First let $\gamma : \widetilde{D} \rightarrow D$ be a monomorphism such that $LFP(\Delta) \subseteq \gamma(LFP(\widetilde{\Delta}))$. γ is called the *concretization function.*

Then define another monotone function $\alpha : D \rightarrow \widetilde{D}$ and we have the tools to define a Galois connection:

Definition 3.13 Let $\alpha : D \rightarrow \widetilde{D}$ and $\gamma : \widetilde{D} \rightarrow D$ be monotone. Then $\langle \alpha \; ; \; \gamma \rangle$ is called a *Galois connection* iff:

$$\alpha(\gamma(\widetilde{x})) = \widetilde{x} \text{ for all } \widetilde{x} \in \widetilde{D}$$
$$\gamma(\alpha(\widetilde{x})) \supseteq x \text{ for all } x \in D$$

□

■ Abstractions

We use Galois connections to relate different abstract interpretations.

Definition 3.14 Let $\langle \widetilde{D} \; ; \; \widetilde{\Delta} \rangle$ and $\langle D \; ; \; \Delta \rangle$ be abstract interpretations and $\alpha : D \rightarrow \widetilde{D}$ and $\gamma : \widetilde{D} \rightarrow D$ be a Galois connection. Then $\langle \widetilde{D} \; ; \; \widetilde{\Delta} \rangle$ is said to be an $\langle \alpha \; ; \; \gamma \rangle$-abstraction of $\langle D \; ; \; \Delta \rangle$. □

■ Conservative approximations

To restrict our search of interesting approximations so that we do not abstract away from the base interpretation in an incorrect way[5], we need to restrict the semantic function:

Definition 3.15 Let $\langle \widetilde{D} \; ; \; \widetilde{\Delta} \rangle$ be an $\langle \alpha \; ; \; \gamma \rangle$-abstraction of $\langle D \; ; \; \Delta \rangle$. The semantic function $\widetilde{\Delta}$ is said to be a *conservative γ-approximation* of Δ iff $\forall \, \widetilde{x} \in \widetilde{D} : \; \Delta(\gamma(\widetilde{x})) \subseteq \gamma(\widetilde{\Delta}(\widetilde{x}))$. □

[5]That is, when operating with concretization function γ on the least fixpoint of the abstract interpretation we must compute a value above LFP of the base interpretation. See figure 3.1.

Finally it is possible to characterize an abstract interpretation whose domain is a sub-lattice of the base domain and whose semantic function is conservative. That is, it "does not lose any information", although it may introduce some "noise".

Theorem 3.16 Let $\langle \widetilde{D} \ ; \ \widetilde{\Delta} \rangle$ be an $\langle \alpha \ ; \ \gamma \rangle$-abstraction of $\langle D \ ; \ \Delta \rangle$. If $\widetilde{\Delta}$ is a conservative γ-approximation of Δ, then $LFP(\Delta) \subseteq \gamma(LFP(\widetilde{\Delta}))$.

3.4 Computing Abstract Interpretations

The final aim of an abstract interpretation framework is to compute abstract interpretations. As we will conclude, computing abstract interpretations is a problem which requires efficient algorithms. A more elaborate discussion on this matter will be found in chapter 6. In the examples in chapter 4 and 5 however, we need *one* algorithm for computing our abstract interpretation.

Earlier in this chapter we said that an abstract interpretation $\langle \widetilde{D} \ ; \ \widetilde{\Delta} \rangle$ has to be recursively enumerable, or to be precise that the *domain* $\widetilde{D}$ has to be recursively enumerable. The reason is that we want to be able to compute it by finite means. If $\widetilde{D}$ is recursively enumerable then it is possible to represent each $x \in \widetilde{D}$ finitely. If the length of $\widetilde{D}$ is also finite we know that there exists a finite n for which $\widetilde{\Delta}^n = \widetilde{\Delta}^{(n+1)}$. We also know from Kleene's fixpoint theorem [Kle52] that $\widetilde{\Delta}^n$ is the least fixpoint, and we can compute the least fixpoint by computing the Kleene sequence:

$$\bot, \ \widetilde{\Delta}(\bot), \ \widetilde{\Delta}^2(\bot), \ldots$$

This is done by the following algorithm, which we later will refer to as the *Jacobi* iteration schema.

Algorithm 3.17 Jacobi iteration

```
X ← (⊥,...,⊥)
while X ≠ Δ(X)
   do
      X ← Δ(X)
   od
return X
```

3.5 Finite Depth and Normal Form Abstraction

One of the first steps in constructing abstract interpretations is to abstract away from the length of the goal. For most interesting programs we have infinite chains of derivations. To keep the goal finite we can *truncate* the tail of the goal when adding new "activation records". This means that we lose information—it is for instance no longer possible to check for satisfiability when adding the "activation record", since some equations are no longer available.

Moreover, if the goal is truncated to length 1, we cannot, when we compute the information at the success point of an atom, even remember the information we had at the calling point.

We present a two-step method of overcoming this imprecision. First, we construct a finite abstraction of our base interpretation (referred to as a *finite depth abstraction*). Second, we will keep the truncated goals in *normal form*.

■ Finite depth abstraction

The purpose of the finite depth abstraction is to put a limit of the length m of the goal. In what follows we will consider goals with $m = 1$.

The first modification of *Entry* will be to make sure that our goal is finite. Since our original *Entry* adds an "activation record" to the goal, we will get an infinite derivation for a recursive program. To keep the goal finite we must cut off the tail of the goal each time we add an activation record, .

Definition 3.18 Let

- $\widetilde{x}_i$ be truncated goals and
- $\widetilde{\vdash}$ the transition relation that adds an activation record and cuts off the tail of the goal.

Then the meaning of an entry point is then (re-)defined thus:

$$X_{\bullet C} = \bigcup_{\alpha \rightsquigarrow C} \{\widetilde{x}_2 \in \widetilde{\Gamma}_{\bullet C} \mid \widetilde{x}_1 \in X_{\bullet\alpha} \wedge \widetilde{x}_1 \widetilde{\vdash} \widetilde{x}_2\}$$

□

The modification of *Succ* consists of adding the information at the calling point of the atom to the success point.

Definition 3.19

$$X_{\alpha\bullet} = \bigcup_{\alpha \rightsquigarrow C} \{\tilde{x}_1 \mathbin{\tilde{\oplus}} \tilde{x}_2 \in \tilde{\Gamma}_{\alpha\bullet} \mid \tilde{x}_1 \in X_{\bullet\alpha} \wedge \tilde{x}_2 \in X_{C\bullet}\}$$

where

$$\langle(\sigma_m, {}_\bullet\alpha, \phi_m)\ ;\ i\rangle \mathbin{\tilde{\oplus}} \langle(\sigma_n, C_\bullet, \phi_n)\ ;\ j\rangle \;=\; \langle(\sigma_n \cup \sigma_m, \alpha_\bullet, \phi_n)\ ;\ j\rangle$$

□

The new definitions of *Succ* and *Entry* are conservative γ-approximations of the old ones.

■ Normal form

To make sure that we always have the interesting information in the topmost part of the goal we introduce a mechanism to transform our goals into normal form. The idea is to (i) collect all the equations in a goal, (ii) rewrite them into normal form and (iii) distribute them them back to the goal.

Consider the following goal taken from example 3.9:

$$\langle(\{\mathtt{f(X_2)} \doteq \mathtt{X_1}, \mathtt{X_2} \doteq \mathtt{y}\}, 5, \phi_2) :: (\{\mathtt{X_1} \doteq \mathtt{X_0}\}, 4, \phi_1) :: (\epsilon, 2, \phi_0)\ ;\ 2\rangle$$

Start with collecting all equations in the goal and applying the solved form algorithm. The set of equations from the example rewrites into solved form for example in the following way if we use the solved form algorithm for S-SLD-resolution and `f` is a constructor:

$$\begin{array}{lll}
\mathtt{f(X_2)} \doteq \mathtt{X_1}, & \mathtt{X_2} \doteq \mathtt{y}, & \mathtt{X_1} \doteq \mathtt{X_0} \\
\mathtt{X_1} \doteq \mathtt{f(X_2)}, & \mathtt{X_2} \doteq \mathtt{y}, & \mathtt{X_1} \doteq \mathtt{X_0} \\
\mathtt{X_1} \doteq \mathtt{f(y)}, & \mathtt{X_2} \doteq \mathtt{y}, & \mathtt{X_1} \doteq \mathtt{X_0} \\
\mathtt{X_1} \doteq \mathtt{f(y)}, & \mathtt{X_2} \doteq \mathtt{y}, & \mathtt{f(y)} \doteq \mathtt{X_0} \\
\mathtt{X_1} \doteq \mathtt{f(y)}, & \mathtt{X_2} \doteq \mathtt{y}, & \mathtt{X_0} \doteq \mathtt{f(y)}
\end{array}$$

Next, put the equations back into the goal again. We add an equation with the left variable adorned with i at the i:th part of the goal. The following goal is a solved form of the goal above.

$$\langle(\{\mathtt{X_2} \doteq \mathtt{y}\}, 5, \phi_2) :: (\{\mathtt{X_1} \doteq \mathtt{f(y)}\}, 4, \phi_1) :: (\{\mathtt{X_0} \doteq \mathtt{f(y)}\}, 2, \phi_0)\ ;\ 2\rangle$$

Similarly a set of configurations is in solved form *iff* the goal of every configuration is in solved form.

Remark 3.20 If f is a *defined* symbol that for instance returned its argument, we could rewrite the equations $\mathtt{X}_0 \doteq \mathtt{f(y)}$ and $\mathtt{X}_1 \doteq \mathtt{f(y)}$ into the following equations: $\mathtt{X}_0 \doteq \mathtt{y}$ and $\mathtt{X}_1 \doteq \mathtt{y}$. This would give the following goal in normal form.

$$\langle(\{\mathtt{X}_2 \doteq \mathtt{y}\}, 5, \phi_2) :: (\{\mathtt{X}_1 \doteq \mathtt{y}\}, 4, \phi_1) :: (\{\mathtt{X}_0 \doteq \mathtt{y}\}, 2, \phi_0) \ ; \ 2\rangle$$

$\square$

Finally we have to add the application of the solved form algorithm to the definitions of *Entry* and *Succ* given above.

First, if the current point is a calling point we (re-)define $\widetilde{\vdash}$ in the following way:

$$\widetilde{x}_1 \widetilde{\vdash} \widetilde{x}_2 \text{ iff } \widetilde{x}_1 \widetilde{\vdash} x_2 \text{ and } \widetilde{x}_2 = \mathit{head}(\mathit{sf}(x_2))$$

Second, we (re-)define $\widetilde{\oplus}$ thus:

$$\langle(\sigma_m, {}_\bullet\alpha, \phi_m) \ ; \ i\rangle \ \widetilde{\oplus} \ \langle(\sigma_n, C_\bullet, \phi_n) \ ; \ j\rangle \ = \ \langle sf((\sigma_n \cup \sigma_m, \alpha_\bullet, \phi_n)) \ ; \ j\rangle$$

Chapter 4

A first shot

In this chapter we develop an abstract interpretation for groundness analysis of restricted GAPLog programs which do not use external functions, i.e. for definite programs. The idea originates in [Nil92]. This abstract interpretation will be extended in Chapter 5 to full GAPLog.

This chapter is organized in the following way: we first define the abstract semantic function, $\widetilde{\Delta}$, in terms of $\overline{Entry}$, $\overline{Succ}$ and $\bigvee$. We then define the domain $\widetilde{D}$, and finally we show an example of how to infer groundness properties of a well known Prolog program.

4.1 The semantic function $\widetilde{\Delta}$

First we recall that the second component of an abstract interpretation is the semantic function $\widetilde{\Delta}$. In this framework it consists of three abstract operations:

$\overline{Entry}$, which is used to approximate the meaning of an entry point of a clause, in terms of the meaning of the calling point of an atom invoking the clause.

$\overline{Succ}$, which is used to approximate the meaning of a success point, in terms of the meaning of an exit point of a clause invoked by the atom.

$\bigvee$, which is the abstract version of the union in the base interpretation. Recall that the meaning of any point in the program is the *collected* meaning of all possible ways to reach the point. Hence the operation is *union* in the base interpretation but can, as we will see, be something quite different in the abstract interpretation.

■ $\overline{Entry}$

We will start with the $\overline{Entry}$-operation, which approximates the meaning of an entry point—the point immediately to the right of the head of a clause. Consider the following program together with a goal:

Program 4.1
```
:- p(a,b,C).

p(X,Y,X).
```

The goal is one of (infinitely) many goals to the program with the two first arguments ground. Since we want to say something universal about the program, that is, we want to draw conclusions about the program when called with *any* goal with its two first arguments ground, the first thing we do is to abstract away the actual values of the arguments in the goal. Hence we introduce an abstract object, **ground**(`X`), meaning that, the variable `X` is instantiated to a ground term.

In what follows we will refer to such an object as a *ground object.* A set of ground objects will be denoted G, possibly with a subscript n meaning *the set of ground objects at point* n.

We can then write the program in the following way:

Program 4.2
```
:- p(A,B,C).

p(X,Y,X).
```

where we have the information $\{\mathbf{ground}(\mathtt{A}), \mathbf{ground}(\mathtt{B})\}$ at the calling point of the goal.

To be precise, on the concrete side one of the (infinitely many) initial configurations will be:

$$\langle(\{\mathtt{A}_0 \doteq \mathtt{a}, \mathtt{B}_0 \doteq \mathtt{b}\}, 0, \phi_0)\ ;\ 0\rangle$$

instead of

$$\langle(\epsilon, 0, \phi_0)\ ;\ 0\rangle$$

The set of all such initial configurations, where $\{\mathtt{A}_0 \doteq \mathtt{a}, \mathtt{B}_0 \doteq \mathtt{b}\}$ is an instance of all possible initial equations with A and B bound to ground terms, will be abstracted by $\{$**ground**(A), **ground**(B)$\}$

What should we be able to say about the *entry point* of the clause p? Since both A and B are ground in the call, and the goal unifies with the head of the clause, X and Y should be instantiated to ground terms after unification. Or to put it differently: we want to conclude $\{$**ground**(X), **ground**(Y)$\}$ at the entry point of the clause. What we need on the abstract side is a way of passing groundness information between the atoms—an *abstract unification algorithm.*

■ The abstract unification algorithm

The aim of the abstract unification algorithm is to pass groundness information from one atom, a to another atom, b. On the abstract side we can say as follows: Suppose a has some ground variables (represented by a set of ground items). Then the result of the abstract unification algorithm should be a new set of ground items which says which variables in b have become ground after the unification.

On the concrete side, one of the derivations will be the following, if we use the solved form abstraction:

$$\frac{\langle(\{\mathtt{A}_0 \doteq \mathtt{a}, \mathtt{B}_0 \doteq \mathtt{b}\}, 0, \phi_0)\ ;\ 0\rangle}{\langle(\{\mathtt{X}_1 \doteq \mathtt{a}, \mathtt{Y}_1 \doteq \mathtt{b}\}, 3, \phi_1) :: (\{\mathtt{A}_0 \doteq \mathtt{a}, \mathtt{B}_0 \doteq \mathtt{b}, \mathtt{C}_0 \doteq \mathtt{a}\}, 1, \phi_0)\ ;\ 1\rangle}$$

This directly yields the two equations $\mathtt{X}_1 \doteq \mathtt{a}, \mathtt{Y}_1 \doteq \mathtt{b}$ at the topmost part of the goal, which is what we want. From this we could abstract the equations to the abstract objects $\{$**ground**(A), **ground**(B)$\}$.

- The abstract unification algorithm has to simulate this behaviour.

To start with, let's assume we have such an algorithm, denoted $AU/3$. $AU/3$ will be formally defined in the next chapter. We can then define our abstract version of *Entry*:

Definition 4.3

$$\overline{Entry}(X, \alpha_i, C_j) \quad = AU(\alpha_i, head(C_j), G_i)$$

where G_i is the set of **ground** objects at the point i and $head(C_j)$ is the head atom of the clause C_j. □

■ $\overline{Succ}$

The next step is to define the $\overline{Succ}$ operation. $Succ$ is used to compute the meaning of a success point of a body atom (A) in terms of an exit point of a clause invoked by A. Recall example 4.2 again. The last step of the derivation on the concrete side is just to pop the topmost element of the goal.

$$\dfrac{\dfrac{\langle(\{\mathtt{A}_0 \doteq \mathtt{a}, \mathtt{B}_0 \doteq \mathtt{b}\}, 0, \phi_0)\ ;\ 0\rangle}{\langle(\{\mathtt{X}_1 \doteq \mathtt{a}, \mathtt{Y}_1 \doteq \mathtt{b}\}, 3, \phi_1) :: (\{\mathtt{A}_0 \doteq \mathtt{a}, \mathtt{B}_0 \doteq \mathtt{b}, \mathtt{C}_0 \doteq \mathtt{a}\}, 1, \phi_0)\ ;\ 1\rangle}}{\langle(\{\mathtt{A}_0 \doteq \mathtt{a}, \mathtt{B}_0 \doteq \mathtt{b}, \mathtt{C}_0 \doteq \mathtt{a}\}, 1, \phi_0)\ ;\ 1\rangle}$$

The problem on the abstract side is that we do not have access to the equations $\{\mathtt{A}_0 \doteq \mathtt{a}, \mathtt{B}_0 \doteq \mathtt{b}, \mathtt{C}_0 \doteq \mathtt{a}\}$ (since the length of the goal on the abstract side is 1). So again we have to use the abstract unification algorithm to pass the groundness information. In order to compute the information at the success point of the goal atom, we look at the exit point of the clause "`p(X,Y,X).`" (which in this case is the same as the entry point) and then (abstractly) unify the head of the clause with the goal atom. In the example we should be able to conclude that, since both `X` and `Y` are ground at the exit point of the clause, all variables in the goal atom should be ground as well. The abstract version of $Succ$ is defined thus:

Definition 4.4

$$\overline{Succ}(X, \alpha_i, C_j) \quad = AU(head(C_j), \alpha_i, G_{Exit(C_j)})$$

where $G_{Exit(C_j)}$ is the set of **ground** objects at the exit point of the clause C_j (and $head(C_j)$ is the head of the clause C_j). □

■ $\bigvee$

What should $\bigvee$ look like? The operation should join the information computed by two (or more) $\overline{Entry}$ (or $\overline{Succ}$) operations. Let's extend program 4.2 a bit:

Program 4.5

```
:- p(A,B,C), p(A,D,D).

p(X,Y,X).
```

where we have the information {**ground**(A), **ground**(B)} at the calling point of the first atom in the goal, and {**ground**(A), **ground**(B), **ground**(C)} at the calling point of the second.

Due to the first atom of the goal, both X and Y should be ground at the entry point of the fact. But due to the call of the second atom in the goal, only X will become ground. Since we want our interpretation to be safe, the $\bigvee$-operation should be *intersection*.

Intersection is thus clearly the correct operation for operating together information at the success point of an atom! Consider the following example:

Example 4.6

```
:- p(A,B,C).

p(X,Y,X).
p(X,X,Y).
```

Let's assume we have the information {**ground**(X)} at the exit points of the two facts. We now want to compute the information at the success point of the goal. Due to the first clause we derive {**ground**(A), **ground**(C)} and due to the second {**ground**(A), **ground**(B)}. The only variable we can be sure is grounded at the success point of the goal is A. □

The three operations $\bigvee$, $\overline{Entry}$, and $\overline{Succ}$ define the second element of the abstract interpretation $\langle \widetilde{D} \ ; \ \widetilde{\Delta} \rangle$. We will in the next section define the first element, the domain.

4.2 The domain $\widetilde{D}$

We will now try to construct the abstract domain, $\widetilde{\Delta}$. The properties we have come up with so far are (*i*) the domain should be built of **ground** objects, (*ii*) the *least upper bound* operation is intersection, and finally (*iii*) the domain should be finite. The lattice should model all program points in the program. One possible solution to these demands is to define a (sub-)

lattice for every program point, and then define a product lattice to model all program points.

How do we construct a lattice for one point? To simplify the presentation in what follows we demand that no clause use the same variables as any other. Formally: $var(C_i) \cap var(C_j) = \emptyset \ \ ; \ i \neq j$. Next, let's assume the presence of a function $Cvars$ which, given a program point, returns the set of all variables in the clause at that particular point.

Example 4.7 Let C be the clause:

```
p(A,B,C) :-
     q(A,Z),
     r(B,Z).
```

Then for any point "inside" the clause $Cvar(\bullet) = \{\texttt{A,B,C,Z}\}$. $\Box$

The lattice $\widetilde{D}_i$ is defined:

Definition 4.8 Let

$$G_i = \{\mathbf{ground}(X) \mid X \in Cvars(\bullet_i)\} \text{ and}$$

$$\widetilde{D}_i = 2^{G_i} \ \cup \ \{\perp\}$$

Then $\langle \widetilde{D}_i \ ; \ \leq \rangle$ is a finite, complete lattice where

(i) $\quad G_n \ \leq \ G_m$ iff $G_m \ \subseteq \ G_n$

(ii) $\quad \forall X \in \widetilde{D}_i : (\perp \leq X)$ and $(X \leq \perp \Rightarrow X = \perp)$

$\Box$

The topmost element is $\emptyset$. Informally the bottom element, $\perp$, represents the fact that the point is not reached yet (if ever). This definition clearly yields a lattice with the properties we wanted; for instance, the least upper bound operation ($\bigvee$) is intersection:

Example 4.9 Let

- $G_1 = \{\mathbf{ground}(X), \mathbf{ground}(Y)\}$ and
- $G_2 = \{\mathbf{ground}(X), \mathbf{ground}(Z)\}$

Then the least upper bound is $\{\mathbf{ground}(X)\}$ which is the intersection of the two sets. $\square$

We have to extend the $\overline{Succ}$ and $\overline{Entry}$ operations to operate on $\perp$. In what follows we will always consider $\overline{Entry}$ and $\overline{Succ}$ to be strict [1].

The definition of $\widetilde{D}_i$ is extended to a product lattice in the obvious way:

Definition 4.10 Let $\langle \widetilde{D}_1 \ ; \ \leq_1 \rangle, \ldots, \langle \widetilde{D}_n \ ; \ \leq_n \rangle$ be lattices for all program points, and

$$\widetilde{D} \ = \ \prod_{i=1}^{n} \widetilde{D}_i$$

Then $\langle \widetilde{D} \ ; \ \leq \rangle$ is a complete lattice where

(i) $\langle X_1, \ldots, X_n \rangle \leq \langle Y_1 \ldots Y_n \rangle$ iff $X_1 \leq_1 Y_1, \ldots, X_n \leq_n Y_n$

$\square$

In what follows we will refer to an element in the domain as well as an element in a sub-domain as a *domain element*. It will be clear from the context which is intended.

Strictness is, as we will see, an important property. If a point in the program is never (or not yet) reached we do not want to bother about that point when computing other points. This is highlighted by following example:

Example 4.11 Consider the following program:

$\texttt{:-}\ {}_0\ \texttt{p(A,B,C)}_1.$

$\texttt{p(X,Y,X)}_2.$
$\texttt{p(X,X,Y):-}_3$
$\qquad \texttt{q(X)}_4.$

where `A` is ground at the calling point of the goal.

Since there are no clauses that unify with the body atom of the second clause of `p`, `q(X)`, we will never reach point 4. Hence it will be of no interest at all to care about the second clause of `p`.

[1] To be precise we demand that AU is strict, in the sense that $\forall X, Y : \ AU(X, Y, \perp) \ = \perp$.

Now consider the problem of computing point[2] 1, that is the success point of the goal. This is done by solving the equation:

$$G_1 = \overline{Succ}(X, \texttt{p(A,B,C)}, C_1) \bigvee \overline{Succ}(X, \texttt{p(A,B,C)}, C_2)$$

where the exit points of the two clauses have the domain elements $G_2 = \{\mathbf{ground}(\texttt{X})\}$ and $G_3 = \perp$ respectively.

The result of solving the equation should only depend on the result of the first $\overline{Succ}$. If $\overline{Succ}$ is strict this will be the case since $X \bigvee \perp = X$. The solution of the equation above will be $\{\mathbf{ground}(\texttt{A}), \mathbf{ground}(\texttt{C})\} \bigvee \perp = \{\mathbf{ground}(\texttt{A}), \mathbf{ground}(\texttt{C})\}$, which is what we wanted. □

4.3 Extending $\overline{Succ}$

The definitions of $\overline{Entry}$ and (especially) $\overline{Succ}$ given in the former section are very simple and not very powerful. It is easy to construct an example where our approximation of the base interpretation leads to unnecessarily great loss of information. Consider the following program:

Program 4.12

$\texttt{:-}_0$ $\texttt{p(A,A,B)}_1$, $\texttt{p(C,D,E)}_2$.

$\texttt{p(X,Y,Y)}_3$.

where we have the domain element $\{\mathbf{ground}(\texttt{A}),\mathbf{ground}(\texttt{C})\}$ at the calling point of the goal.

Consider the problem of computing point 1.

To start with we compute point 3. This gives the element $\{\mathbf{ground}(\texttt{X}), \mathbf{ground}(\texttt{Y})\}$. Computing the success point of `p(A,A,B)`, point 1, clearly yields the element $\{\mathbf{ground}(\texttt{A}), \mathbf{ground}(\texttt{B})\}$. We have lost the information $\mathbf{ground}(\texttt{C})$! The reason for this is that we have not propagated the object $\mathbf{ground}(\texttt{C})$ to the clause. What we need is to extend the $\overline{Succ}$-operation so that we do not throw away groundness information within a clause.

The way of doing this is straightforward, we just add all ground elements at the calling point of the atom to the success point:

Definition 4.13

$$\overline{Succ}(X, \alpha_i, C_j) \quad = AU(C_j, \alpha_i, G_{Exit(C_j)}) \cup G_i$$

[2]This is equivalent to say "computing the domain element of point 1". We will in what follows use this shorter expression.

where $G_{Exit(C_j)}$ is the set of **ground** objects at the exit point of the clause C_j. □

4.4 An example

Before we summarize what we have done so far, we show an example of an analysis of a small program implementing append.

```
C0:  :-₁append(L1, L2, L3).₂
C1:  append([],L,L)₃.
C2:  append([X|Xs],Y,[X|Z])₄:-
        append(Xs,Y,Z).₅
```

If we call the program with `L1` and `L2` ground, the following set of equations are induced by the program:

$$
\begin{array}{lcl}
\overline{X}_1 & = & \{\mathbf{ground}(\mathtt{L1}), \mathbf{ground}(\mathtt{L2})\} \\
\overline{X}_2 & = & \overline{Succ}(\overline{X}, \mathtt{append(L1,L2,L3)}, C_1) \ \bigvee \\
 & & \overline{Succ}(\overline{X}, \mathtt{append(L1,L2,L3)}, C_2) \\
\overline{X}_3 & = & \overline{Entry}(\overline{X}, \mathtt{append(L1,L2,L3)}, C_1) \ \bigvee \\
 & & \overline{Entry}(\overline{X}, \mathtt{append([X|Xs],Y,[Z|Zs])}, C_1) \\
\overline{X}_4 & = & \overline{Entry}(\overline{X}, \mathtt{append(L1,L2,L3)}, C_2) \ \bigvee \\
 & & \overline{Entry}(\overline{X}, \mathtt{append([X|Xs],Y,[Z|Zs])}, C_2) \\
\overline{X}_5 & = & \overline{Succ}(\overline{X}, \mathtt{append(Xs,Y,Zs)}, C_1) \ \bigvee \\
 & & \overline{Succ}(\overline{X}, \mathtt{append(Xs,Y,Zs)}, C_2)
\end{array}
$$

The starting element will be a vector of bottom values.

$$(\bot, \bot, \bot, \bot, \bot)$$

The first iteration computes a vector where the first element consist of the information that `L1` and `L2` are ground. For typographical reasons we just write the variable, for instance `L1`, instead of the whole object, **ground**(`L1`).

$$(\{\mathtt{L1,L2}\}, \bot, \bot, \bot, \bot)$$

The first iteration propagates the groundness to the entry points of the two clauses.

$$(\{\mathtt{L1,L2}\}, \bot, \{\mathtt{L}\}, \{\mathtt{X,Xs,Y}\}, \bot)$$

Since the entry point of the first clause, point 3, is also an exit point and more important, related to the success point of the goal atom, we can compute the success point of the goal. The groundness of `L` (at point 3) is thus propagated back.

Since both `Xs` and `Y` are ground in the second clause, the information at point 3 will be the same. When we compute point 5 we just consider the information at the exit point of the first clause, since the exit point of the second is not reached yet.

$$(\{\texttt{L1,L2}\}, \{\texttt{L1, L2, L3}\}, \{\texttt{L}\}, \{\texttt{X,Xs,Y}\}, \{\texttt{X,Xs,Y,Z}\})$$

We will have to compute this element once more to realize that we have reached the fixpoint.

4.5 Conclusions

In this chapter we have constructed an abstract interpretation, $\langle \widetilde{D}\ ;\ \widetilde{\Delta} \rangle$, which we can use to infer properties of pure Prolog programs. This was done by:

- introducing an abstract object **ground**(X) meaning "X is instantiated to a ground term". This abstracted away from the exact values of a variable.
- defining $\widetilde{\Delta}$ in terms of $\overline{Entry}$, $\overline{Succ}$ and $\bigvee$, based on the ground objects. $\overline{Succ}$ was extended so that we did not loose groundness information inside a clause.
- defining the domain, $\widetilde{D}$—the first component of the abstract interpretation.

We did not:

1. Define the abstract unification algorithm.
2. Define a *Galois connection* between the abstract interpretation and the base interpretation and hence prove the correctness of the work. This is something we will not do in this thesis. For this domain, however, the Galois connection is defined in [Nil92].

In the next chapter we will extend the abstract interpretation to handle equations.

Chapter 5

A second shot

In this chapter we will extend the abstract interpretation. The aim is to be able to infer properties not just about pure Prolog programs, as in the former chapter, but of GAPLog programs.

A program written in GAPLog may contain equations as body atoms, such as "`X = Y + Z`" where +/2 is a defined symbol. The equations appear in the program as an affect of that the GAPLog compiler flattens the program before adding the delays. The problem we must solve now is how to treat these *equational* atoms. To highlight the problem let us look at a program with an equation.

Program 5.1

$\texttt{:-}_0\texttt{sum(A,B,C)}_1$

$\texttt{sum(X,Y,Z)}_2\texttt{:-}$
$\quad\texttt{Z = X + Y}_3\texttt{.}$

with $G_0 = \{\textbf{ground}(\texttt{A}), \textbf{ground}(\texttt{B})\}$.

Computing the entry point of the clause yields the element $\{\textbf{ground}(\texttt{X}), \textbf{ground}(\texttt{Y})\}$. The next point to compute is the exit point of the clause, point 3. Now we run into problems. Our previous method for groundness propagation was restricted to non-equational body atoms. We now extend it to handle the case when the next atom is an equation.

We want to be able to conclude **ground**(`Z`) since both `X` and `Y` are ground. Hence we could say that since both **ground**(`X`) and **ground**(`Y`) we could conclude that **ground**(Z). This would yield {**ground**(X), **ground**(Y), **ground**(Z)} at the exit point of the clause, and by propagating the groundness information back to the success point of the goal atom we conclude {**ground**(A), **ground**(B), **ground**(C)}, which is what we expected.

This way of reasoning is however not enough, as illustrated by the following modification of the example program.

Program 5.2

$$\texttt{:-}_0\texttt{p(A,B,C)}_1.$$

$$\begin{array}{l}\texttt{p(X,Y,Z)}_2\texttt{:-}\\ \quad \texttt{Z = f(X,W)}_3,\\ \quad \texttt{W = g(X,Y)}_4.\end{array}$$

with $G_0 = \{\textbf{ground}(\texttt{A}), \textbf{ground}(\texttt{B})\}$.

As before we compute $G_2 = \{\textbf{ground}(\texttt{X}), \textbf{ground}(\texttt{Y})\}$. Reasoning in the same way, we try to conclude something about the success point of the first atom of the clause. But since we do not know anything about the groundness of W, we cannot draw any conclusion about groundness of Z. However G_2 will be propagated to the next point as before. Finally at the exit point we conclude **ground**(W) since {**ground**(X), **ground**(Y)}. So the ground element we yield at the exit point of the clause is {**ground**(X), **ground**(Y), **ground**(W)} and hence {**ground**(A), **ground**(B)} at the success point of the goal atom.

■ The **if** object

This analysis is however imprecise. In a real implementation the call to + will be frozen until (eventually) both X <u>and</u> W become instantiated, in this case at the exit point of the clause. What we need is a way of "remembering" the fact that Z will become ground if both X and Y are ground. The solution to this will be to introduce a new abstract object, **if**(Z, V), where **Z** is a variable and **V** a set of variables, meaning: *The variable Z is ground if all variables in V are ground.* In what follows an object of this kind will be referred to as an *if object.* A set of **if** objects will be denoted *I*, possibly with a subscript *n*, meaning *the set of* **if** *objects at program point n.*

The information at a program point will from now on consist of two sets of abstract objects. G, a set of **ground** *objects* and I a set of *if objects*. We first consider how to compute **if** objects.

■ Generating **if** objects

The only construction in the GAPLog language which gives rise to new **if** objects is the equation. An equation is an atom of the following form:

$$T_1 \doteq T_2$$

where each T_i is either a *variable*, a *constructor term*, or a *functional term*. When we compute a domain element of a success point of an equation, we will use the function $constr(\alpha)$ to compute new **if** objects:

Definition 5.3 $constr(\alpha)$

The function *constr* is defined by cases, depending on the form of equation. In the following definition, let *alpha* be a (body) atom, X a variable, t, s constructor symbols, f, g defined symbols, and T, S arbitrary terms.

$$\begin{array}{lcl}
constr(t(T_1,\ldots,T_n) \doteq X) & = & constr(X \doteq t(T_1,\ldots,T_n)) \\
constr(X \doteq t(T_1,\ldots,T_n)) & = & \{\mathbf{if}(X, vars(t(T_1,\ldots,T_n)))\} \cup \\
 & & \{\mathbf{if}(Y,\{X\}) \mid Y \in vars(t(T_1,\ldots,T_n))\} \\
constr(f/\mathrm{n} \doteq t/\mathrm{m}) & = & constr(t/\mathrm{m} \doteq f/\mathrm{n}) \\
constr(t/\mathrm{m} \doteq f/\mathrm{n}) & = & \{\mathbf{if}(X, vars(f)) \mid X \in vars(t/\mathrm{m})\} \\
constr(t(T_1,\ldots,T_n) \doteq t(S_1,\ldots,S_n)) & = & constr(T_1 \doteq S_1) \cup, \ldots, \\
 & & \cup\, constr(T_n \doteq S_n) \\
constr(f/\mathrm{n} \doteq g/\mathrm{m}) & = & \emptyset \text{ if } f/n \neq g/m \\
constr(t/\mathrm{n} \doteq s/\mathrm{m}) & = & \perp \ ; t/n \neq s/m \\
constr(\alpha) & = & \emptyset;\ \alpha \text{ not equation}
\end{array}$$

□

■ **if** objects as dynamic objects

One way to proceed from here is to view the **if** *objects* as dynamical objects created while analyzing the program. To start with we say that every program point is associated with a domain element, $(G, I)_i$, consisting of a set of **ground** objects and a set of if objects. Following the ideas of Hanus in [Han92] we define a set of *normalization rules* operating on $(G, I)_i$.

Definition 5.4 *Normalization rules*

$$(1)\left[\begin{array}{lll} G_i & = & G \\ I_i & = & I \cup \{\mathbf{if}(Y, \emptyset)\} \end{array}\right] \Rightarrow \left[\begin{array}{lll} G_i & = & G \cup \{\mathbf{ground}(Y)\} \\ I_i & = & I \end{array}\right]$$

$$(2)\left[\begin{array}{lll} G_i & = & G \cup \{\mathbf{ground}(X)\} \\ I_i & = & I \cup \{\mathbf{if}(Y, \{X\} \cup V)\} \end{array}\right] \Rightarrow \left[\begin{array}{lll} G_i & = & G \cup \{\mathbf{ground}(X)\} \\ I_i & = & I \cup \{\mathbf{if}(Y, V)\} \end{array}\right]$$

$$(3)\left[\begin{array}{lll} G_i & = & G \cup \{\mathbf{ground}(X)\} \\ I_i & = & I \cup \{\mathbf{if}(X, V)\} \end{array}\right] \Rightarrow \left[\begin{array}{lll} G_i & = & G \cup \{\mathbf{ground}(X)\} \\ I_i & = & I \end{array}\right]$$

$$(4)\left[\begin{array}{lll} G_i & = & G \\ I_i & = & I \cup \{\mathbf{if}(X, V_1), \mathbf{if}(X, V_2)\} \end{array}\right] \Rightarrow \left[\begin{array}{lll} G_i & = & G \\ I_i & = & I \cup \{\mathbf{if}(X, V_1)\} \\ & & \text{if } V_1 \subseteq V_2 \end{array}\right]$$

□

Given these rules we look at an example of how to analyze a program:

Example 5.5

$\texttt{:-}_0\texttt{q(A)}_1$

$\texttt{q(T)}_2\texttt{:-,}$
$\quad\texttt{T = X*Y}_3\texttt{,}$
$\quad\texttt{Y = V+X}_4\texttt{,}$
$\quad\texttt{p(X)}_5\texttt{.}$
$\quad\texttt{r(V)}_6\texttt{.}$

$\texttt{p(1).}_7$
$\texttt{r(3).}_8$

with $(G, I)_0 = (\emptyset, \emptyset)$.

Computing the (interesting) program points (in the obvious order) yields the following domain elements:

$(G, I)_2 = (\emptyset, \emptyset)$

$(G, I)_3 = (\emptyset, \{\mathbf{if}(\texttt{T}, \{\texttt{X}, \texttt{Y}\})\})$

$(G, I)_4 = (\emptyset, \{\mathbf{if}(\texttt{T}, \{\texttt{X}, \texttt{Y}\}), \mathbf{if}(\texttt{Y}, \{\texttt{V}, \texttt{X}\})\})$

$(G, I)_5 = (\{\mathbf{ground}(\mathtt{X})\}, \{\mathbf{if}(\mathtt{T}, \{\mathtt{Y}\}), \mathbf{if}(\mathtt{Y}, \{\mathtt{V}\})\})$

$(G, I)_6 = (\{\mathbf{ground}(\mathtt{X}), \mathbf{ground}(\mathtt{V}), \mathbf{ground}(\mathtt{Y}), \mathbf{ground}(\mathtt{T})\}, \emptyset)$

$(G, I)_1 = (\{\mathbf{ground}(\mathtt{A})\}, \emptyset)$

□

This way of viewing the **if** objects has the following disadvantages:

- It is hard to compare domain elements (e.g. define the $\leq$ relation).
- We might, if we allow **if** objects to be passed to other clauses, compute infinite domain elements. In case of a recursive program, we can not as in the case of the **ground** objects, just pass the groundness information to the entry point of clause with the result that no new objects are generated. Instead we have to pass the whole (renamed) **if** object, and as a result of this the number of objects increases for each time we compute the entry point.
- The domain will be more complex.

■ **if** objects as static objects

There is an easier way to treat the **if** *objects*. Following the framework of Nilsson we put a limit on the length of a goal. This is because programs involving recursion make the base interpretation infinite. A limited abstract interpretation will therefore just contain a finite part of the base interpretation. To start with, we will

- truncate our goal to length 1.
- Not pass **if** objects generated in a clause to another clause.

This also has the advantage that it is possible to view the information we get from **if** objects as static information. Hence we can compute them in advance, and instead of making use of a set of normalization rules use a *saturation function*, which given a set of **ground** objects and a set of **if** objects, *saturates* the set of **ground** objects wrt the set of **if** objects. This approach will in what follows be referred to as the "static approach".

To do this we introduce a function, *pconstr*($\bullet$), which given a program point, computes the set of **if** objects *to the left of the point inside the clause.*

Definition 5.6 *pconstr*

Let $\bullet_i$ be the ith point and α_k the atom to the right of program point k. Then

$$pconstr(\bullet_i) = \emptyset \text{ if } \bullet_i \text{ is an entry point}$$

$$pconstr(\bullet_i) = \bigcup_{k=Entry(\bullet_i)}^{i-1} constr(\alpha_k)$$

where $Entry(\bullet_i)$ is the number of the entry point of the clause of $\bullet_i$. □

Example 5.7 Let C be the clause:

$\texttt{q(T)}_{10}\texttt{:-},$
$\quad\texttt{p(X,Y,72)}_{11},$
$\quad\texttt{X = V-W}_{12},$
$\quad\texttt{Y = V+W}_{13},$
$\quad\texttt{pick(V,W)}_{14}.$

Then for instance

- $pconstr(12) = \bigcup_{k=10}^{11} constr(\alpha_k) = \{\mathbf{if}(\texttt{X}, \{\texttt{V,W}\})\}$
- $pconstr(14) = \{\mathbf{if}(\texttt{X}, \{\texttt{V,W}\}), \mathbf{if}(\texttt{Y}, \{\texttt{V,W}\})\}$

□

The saturation function, *gsat*, is defined thus:

Definition 5.8 *gsat*

Let G be a set of **ground** objects, and I a set of **if** objects. Then $gsat(G, I)$ is the smallest extension G' of G such that

$$\begin{array}{ll} \text{if} & \mathbf{ground}(X_1), \ldots, \mathbf{ground}(X_n) \in G' \text{ and} \\ & \mathbf{if}(Y, \{X_1, \ldots, X_n\}) \in I \quad (n \geq 0) \\ \text{then} & \mathbf{ground}(Y) \in G' \end{array}$$

□

Remark 5.9 *gsat* is assumed to be *strict* in its first argument, so for instance $gsat(\perp, I) = \perp$. □

5.1 $\widetilde{\Delta}$

We can now define $\widetilde{\Delta}$ in terms of $\overline{Entry}$, $\overline{Succ}$ and $\bigvee$.

■ $\overline{Entry}$

Since we chose not to pass **if** objects between clauses, the definition of $\overline{Entry}$ will be the same as in the former chapter:

Definition 5.10 $\overline{Entry}$

$$\overline{Entry}(X, \alpha_i, C_j) \quad = AU(\alpha_i, C_j, G_i)$$

where G_i is the set of ground variables in the point i. □

■ $\overline{Succ}$

However, the definition of $\overline{Succ}$ will be extended to:

Definition 5.11 $\overline{Succ}$

$$\begin{array}{lll} & \overline{Succ}(X, \alpha_i) & = gsat(G_i,\ I_{i+1}) \\ \text{if} & & \alpha_i \text{ equation} \\ & & \\ & \overline{Succ}(X, \alpha_i, C_j) & = gsat(G_i\ \cup\ AU(C_j, \alpha_i, G_{Exit(C_j)}),\ I_i) \\ & & \text{otherwise} \end{array}$$

where G_i is the set of ground variables of the point i and $G_{(Exit(C_j))}$ is the set of **ground** objects at the exit point of the clause C_j. □

■ $\bigvee$

Since we chose not to change our domain, the $\bigvee$-operation will be the same as in the former chapter, namely intersection.

5.2 The abstract unification algorithm

Before we end this chapter (with a bigger example) we show one possible way of defining the abstract unification algorithm, AU. The purpose of the abstract unification algorithm is to unify two atoms and return the variables which become ground after unification. Since we chose not to pass **if** objects between clauses, we only have to worry about **ground** objects. There is however one problem, we allow constructor terms to be part of an non-equational atom. (Remember that we only abandoned the presence of functional terms in non-equational atoms[1].)

There are of course many ways of constructing such an algorithm. The one presented below is a modified version of the one presented in [Nil92].

Example 5.12 Suppose we have the following (part of a) program.

$$:\!-_0\ \mathtt{p(X,t(v(17,B)))}_1.$$

$$\mathtt{p([V|Vs],t(Y))}_2\mathtt{:-}$$
$$\vdots$$

where $G_0 = \{\mathbf{ground}(\mathtt{X})\}$. If we try to compute G_2, then we want `V` and `Vs` to be ground at the entry point of the clause. However we do not want `Y` to be ground since the term `t(v(17,B))` is not fully instantiated to a ground term. One way of proceeding is to make use of the machinery we already have:

We can view the two atoms we want to unify as a "directed" equation. Take the two atoms in the example above. Then we get the following equation:

$$\{\mathtt{p(X,t(v(17,B)))} \doteq \mathtt{p([V|Vs],t(Y))}\}$$

If we "decompose" the equation, we get the following set of equations:

$$\{\mathtt{X} \doteq \mathtt{[V|Vs]},\ \mathtt{v(17,B)} \doteq \mathtt{Y}\}$$

From this set we can make a set of **if** objects, in this case:

$$\{\mathbf{if}(\mathtt{X},\{\mathtt{V},\mathtt{Vs}\}),\mathbf{if}(\mathtt{B},\{\mathtt{Y}\})\}$$

[1]A non-equational atom is an atom of the form α. An equational atom is an atom of the form $\alpha \doteq \beta$.

If we then add the set of **ground** objects at the, in this case, calling point of the calling atom. we can saturate the set of ground objects and we are home. To keep the variables from different atoms apart from each other, we assume the presence of a "colouring"-function, $\downarrow$, to colour the variables in one of the atoms. □

■ *dec*

To make the set of **if** *objects* we use a modification of the *constr*-algorithm, called *dec*, where we do not have to bother about external function calls, but must extend it to handle the case where the left hand side of the equation is a compound term:

Definition 5.13 $dec(\alpha_i = \alpha_j)$

Let X, Y represent variables, t, s constructor symbols, and T, S compound terms. Then

$$
\begin{array}{lcl}
dec(X \doteq Y) & = & \{\mathbf{if}(X, \{Y\})\} \\
dec(X \doteq t) & = & \{\mathbf{if}(X, vars(t))\} \\
dec(t/\mathrm{n} \doteq X) & = & \{\mathbf{if}(Y, \{X\}) \mid Y \in vars(t)\} \\
dec(t(T_1, \ldots, T_n) \doteq t(S_1, \ldots, S_n)) & = & dec(T_1 \doteq S_1) \cup, \ldots \cup dec(T_n \doteq S_n) \\
dec(t/\mathrm{n} \doteq s/\mathrm{m}) & = & \perp \ ; t/n \neq s/m
\end{array}
$$

□

The last step of the abstract unification algorithm will be to throw away coloured objects and keep the uncoloured. This is done by a filtering function, *down*, which filters away the **ground** objects coloured by $\downarrow$.

The definition of the abstract unification algorithm is straightforward.

Algorithm 5.14 $AU(A_i, A_j, G)$

1. colour A_i and G with $\downarrow$
2. apply the *dec*-algorithm on $A_i \doteq A_j$ (giving S)
3. apply *gsat* wrt G on S (giving S')
4. filter away coloured objects

Formally:

Definition 5.15 *AU*

$$AU(A_i, A_j, G) = down(gsat(G\downarrow, dec(A_j \doteq A_i\downarrow)))$$

□

Remark 5.16 Since *gsat* is **strict**, it follows that *AU* is strict. □

Example 5.17 Abstract unification
We want to unify the atoms in example 5.12 where $G = \{\mathbf{ground}(\mathtt{X})\}$. The abstract unification algorithm performs the following steps:

$AU(\mathtt{p(X,t(v(17,B)))},\ \mathtt{p([V|Vs],t(Y))},\ G) =$

$down(gsat(G\downarrow, dec(\mathtt{p([V|Vs],t(Y))} \doteq \mathtt{p(X,t(v(17,B)))}\downarrow))) =$

$down(gsat(G\downarrow, dec(\mathtt{[V|Vs]} \doteq \mathtt{X}\downarrow, \mathtt{Y} \doteq \mathtt{v(17,B}\downarrow\mathtt{)}))) =$

$down(gsat(\{\mathbf{ground}(\mathtt{X}\downarrow)\}, \{\mathbf{if}(\mathtt{V}, \{\mathtt{X}\downarrow\}), \mathbf{if}(\mathtt{Vs}, \{\mathtt{X}\downarrow\}), \mathbf{if}(\mathtt{Y}, \{\mathtt{B}\downarrow\})\})) =$

$down(\{\mathbf{ground}(\mathtt{X}\downarrow), \mathbf{ground}(\mathtt{V}), \mathbf{ground}(\mathtt{Vs})\}) =$

$\{\mathbf{ground}(\mathtt{V}), \mathbf{ground}(\mathtt{Vs})\}$

□

5.3 An example

Finally we will show an animation of an analysis of a well-known program, implementing the factorial function. Before we start we make an observation. We do not need to recompute the set of **if** objects at different program points every time we compute a particular point. They can be computed once and for all as we visit the point for the first time. If the point is not visited before we just add the constraints from the point immediately to the left of that point (and of course the new **if** object in case the atom to the left was an equational one). Consider the following program together with a goal where `X` is ground.

Program 5.18 [2]

```
    :- 0fac(X,Y)1.

fac(0,1)2.
fac(A,B)3:-
    B = A * C 4,
    E = A - 1 5,
    fac(E,C) 6.
```

where `*` and `-` have their natural interpretations.

The program gives the following set of equations:

$$\begin{array}{lcl}
\overline{X}_0 & = & \{\mathbf{ground}(\mathtt{X})\} \\
\overline{X}_1 & = & \overline{Succ}(\overline{X}, C_1, \mathtt{fac(X,Y)}) \ \bigvee \ \overline{Succ}(\overline{X}, C_2, \mathtt{fac(A,B)}) \\
\overline{X}_2 & = & \overline{Entry}(\overline{X}, C_1, \mathtt{fac(X,Y)}) \ \bigvee \ \overline{Entry}(\overline{X}, C_1, \mathtt{fac(E,C)}) \\
\overline{X}_3 & = & \overline{Entry}(\overline{X}, C_2, \mathtt{fac(X,Y)}) \ \bigvee \ \overline{Entry}(\overline{X}, C_2, \mathtt{fac(E,C)}) \\
\overline{X}_4 & = & \overline{Succ}(\overline{X}, \mathtt{B} = \mathtt{A} * \mathtt{C}) \\
\overline{X}_5 & = & \overline{Succ}(\overline{X}, \mathtt{E} = \mathtt{A} - \mathtt{1}) \\
\overline{X}_6 & = & \overline{Succ}(\overline{X}, C_1, \mathtt{fac(E,C)}) \ \bigvee \ \overline{Succ}(\overline{X}, C_2, \mathtt{fac(E,C)})
\end{array}$$

The starting element is as usual a vector of $\perp$.

$$(\perp, \perp, \perp, \perp, \perp, \perp, \perp)$$

The first iteration yields a vector where the first element consist of the information that `X` is ground – the information from the user. For typographical reasons we use the shorthand `X` denoting **ground**(X). The elements in the vector should be read: $\left[\begin{array}{c}\text{ground elements}\\ \textbf{if}\text{-elements}\end{array}\right]$.

$$(\left[\begin{array}{c}\mathtt{X}\\ \emptyset\end{array}\right], \perp, \perp, \perp, \perp, \perp, \perp)$$

The first iteration makes the information in point 0 to propagate to point 2 and 3:

$$(\left[\begin{array}{c}\mathtt{X}\\ \emptyset\end{array}\right], \perp, \left[\begin{array}{c}\emptyset\\ \emptyset\end{array}\right], \left[\begin{array}{c}\mathtt{A}\\ \emptyset\end{array}\right], \perp, \perp, \perp)$$

[2]This is in fact the result of the compilation of the following program with the GAPLog compiler: `fac(0,1).  fac(X,X*Y) :- fac(X-1,Y).` .

Next iteration makes an exit-point (2) current so the information in that point propagates back to point 1. The groundness information in the second clause moves to point 4, where the dependency information generated from atom 3 is added.

$$(\begin{bmatrix} \mathtt{X} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{X,Y} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}) \end{bmatrix}, \bot, \bot)$$

Iteration number three moves the information in the second clause to point 5 where a new **if** object is added. Since the groundness of `E` depends on the groundness of `A` which already is ground, `E` becomes ground as well:

$$(\begin{bmatrix} \mathtt{X} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{X,Y} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}) \end{bmatrix},$$

$$\begin{bmatrix} \mathtt{A,E} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}), \mathbf{if}(\mathtt{E}, \{\mathtt{A}\}) \end{bmatrix}, \bot)$$

In the fourth iteration the **if**-elements in point 5 together with the fact that `C` becomes ground due to the first clause makes all variables in the second clause ground:

$$(\begin{bmatrix} \mathtt{X} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{X,Y} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}) \end{bmatrix},$$

$$\begin{bmatrix} \mathtt{A,E} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}), \mathbf{if}(\mathtt{E}, \{\mathtt{A}\}) \end{bmatrix}, \begin{bmatrix} \mathtt{A,B,C,E} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}), \mathbf{if}(\mathtt{E}, \{\mathtt{A}\}) \end{bmatrix})$$

We will need yet another iteration to realize that this is the fixpoint.

■ Resulting program

What then can we say about the program? We can for instance say that `A` always is ground when calling "`-`". Hence we do not need to dynamically check if `A` is ground. The second function call has only one variable non-ground, `C` so in the call to "`*`" we will only need to check the groundness of `C`. The program could thus be transformed to:

Program 5.19

```
fac(0,1).
fac(A,B) :-
     gfreeze([C], fun_call(A*C,D)),
     fun_call(A-1,E),
     fac(E,C).
```

■ Correctness of the abstract interpretation

This thesis does not include any formal proof of the correctness of the abstract interpretation developed in this chapter. Intuitively, correctness means that the result of our abstract interpretation for a given class of goals properly characterizes the configurations computed at each program point by S-SLD resolution starting in any of the goals in the class (or obtained by the base interpretation of chapter 3 initialized with such goals). To give some intuition that we are actually abstracting away correctly, let's look at a part of the base interpretation and compare it with the information obtained by our abstract interpretation.

One of (infinitely) many ways of calling the factorial predicate is with the first argument bound to 3; that is, the initial state is the equation $\mathtt{X}_0 \doteq \mathtt{3}$. Then one derivation, to point 5, will look as follows if does not use the solved form abstraction.

$$\frac{\dfrac{\langle(\{\mathtt{X}_0 \doteq \mathtt{3}\}, 0, \phi_0)\ ;\ 0\rangle}{\langle(\{\mathtt{A}_1 \doteq \mathtt{X}_0, \mathtt{B}_1 \doteq \mathtt{Y}_0\}, 3, \phi_1) :: (\{\mathtt{X}_0 \doteq \mathtt{3}\}, 2, \phi_0)\ ;\ 1\rangle}}{\langle(\{\mathtt{B}_1 \doteq \mathtt{A}_1 {*} \mathtt{C}_1, \mathtt{A}_1 \doteq \mathtt{X}_0, \mathtt{B}_1 \doteq \mathtt{Y}_0\}, 5, \phi_1) :: (\{\mathtt{X}_0 \doteq \mathtt{3}\}, 2, \phi_0)\ ;\ 2\rangle}$$

If we use the normal form abstraction the derivation will have the following outlook:

$$\frac{\dfrac{\langle(\{\mathtt{X}_0 \doteq \mathtt{3}\}, 0, \phi_0)\ ;\ 0\rangle}{\langle(\{\mathtt{A}_1 \doteq \mathtt{3}, \mathtt{B}_1 \doteq \mathtt{Y}_0\}, 3, \phi_1) :: (\{\mathtt{X}_0 \doteq \mathtt{3}\}, 2, \phi_0)\ ;\ 1\rangle}}{\langle(\{\mathtt{B}_1 \doteq \mathtt{3} {*} \mathtt{C}_1, \mathtt{A}_1 \doteq \mathtt{3}, \mathtt{B}_1 \doteq \mathtt{Y}_0\}, 5, \phi_1) :: (\{\mathtt{X}_0 \doteq \mathtt{3}\}, 2, \phi_0)\ ;\ 2\rangle}$$

In our abstract interpretation we considered calls of the factorial procedure with the first argument being ground. The call considered above belongs to this class. Our abstract information at point 5 is as follows:

$$\begin{bmatrix} \mathbf{ground}(\mathtt{A}) \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}) \end{bmatrix}$$

Consider the first derivation. Them the **if**$(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\})$ item clearly corresponds to the equation $\mathtt{B}_1 \doteq \mathtt{A}_1{*}\mathtt{C}_1$. In case of **ground**$(\mathtt{A})$ it corresponds to the two equations $\{\mathtt{X}_0 \doteq \mathtt{3},\ \mathtt{A}_1 \doteq \mathtt{X}_0\}$. To derive the groundness we will have to "dereference" the variable $\mathtt{A}_1$ yielding the equation $\mathtt{A}_1 \doteq \mathtt{3}$.

Next, consider the second derivation. Now it is clear that **ground**$(\mathtt{A})$ corresponds to the second equation of the top of the goal. It is not so clear that the corresponding equation to **if**$(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\})$ is $\mathtt{B}_2 \doteq \mathtt{3}{*}\mathtt{C}_2$ since we have "replaced" the variable `A` with `3`. We believe that it will probably be easier to prove correctness using the dynamic approach and relate it to the finite depth and solved form abstraction.

Chapter 6

Computing the fixpoint

One part of this work was to develop a prototype implementation of the abstract interpretation described in the previous chapter. The implementation was done using the dynamic approach[1], on a subset of GAPLog containing user-defined predicates and some predefined arithmetic functions.

In [Nil92] some iteration strategies were mentioned and discussed, namely the *Jacobi* and *Chaotic* iteration strategies. There is also a variant of the Jacobi iteration strategy mentioned, called the *Gauss-Seidel* iteration strategy. The objective of our work was to implement and compare these three strategies and to compare the experimental results with the complexity analysis in [Nil92].

During the implementation a fourth iteration strategy was invented. This iteration strategy was not implementable (in Prolog) since it is based on parallelism. The strategy however turned out to be previously described in [CC93], under the name *asynchronous iteration*.

■ implementation problems

The implementation was done in Prolog, and one of the problems with Prolog is the lack of fast data structures. The domain element was implemented as a 2-3-tree, which has an access time of $O(log\ n)$ where n is the number of program points. It would however be desirable to be able

[1]The static approach was invented later as the prototype implementation of the analyzer turned out not to include "passing if-objects between clauses".

to simply reset the domain element and not have to rebuild (parts of) the tree every time one computes new parts of the domain. Moreover, a domain element implemented as an array would have a constant access time. We also believe that an implementation based on the the static approach might have been more efficient.

6.1 Jacobi iteration

The Jacobi iteration strategy is a simple and naive approach to the problem of computing the least fixpoint. We simply compute a new domain element until we (eventually) reach the fixpoint. The iteration scheme is as follows:

```
X ← (⊥, ..., ⊥)
while X ≠ Δ(X)
   do
      X ← Δ(X)
   od
return X
```

6.2 Gauss-Seidel iteration

The difference between the Jacobi and the Gauss-Seidel iteration strategies, is the obvious improvement of using the new values of each D_i when computing the next. The Gauss-Seidel has the same iteration scheme as the Jacobi with the difference that the while Jacobi iteration strategy computes a new domain element using the following equations:

$$\begin{array}{rcl} \mathrm{X}_1^{k+1} & = & \Delta_1(\mathrm{X}_1^k, \mathrm{X}_2^k, \ldots, \mathrm{X}_{n-1}^k, \mathrm{X}_n^k) \\ \mathrm{X}_2^{k+1} & = & \Delta_2(\mathrm{X}_1^k, \mathrm{X}_2^k, \ldots, \mathrm{X}_{n-1}^k, \mathrm{X}_n^k) \\ & \vdots & \\ \mathrm{X}_n^{k+1} & = & \Delta_n(\mathrm{X}_1^k, \mathrm{X}_2^k, \ldots, \mathrm{X}_{n-1}^k, \mathrm{X}_n^k) \end{array}$$

the Gauss-Seidel iteration strategy uses the following:

$$\begin{array}{rcl} \mathrm{X}_1^{k+1} & = & \Delta_1(\mathrm{X}_1^k, \mathrm{X}_2^k, \ldots, \mathrm{X}_{n-1}^k, \mathrm{X}_n^k) \\ \mathrm{X}_2^{k+1} & = & \Delta_2(\mathrm{X}_1^{k+1}, \mathrm{X}_2^k, \ldots, \mathrm{X}_{n-1}^k, \mathrm{X}_n^k) \\ & \vdots & \\ \mathrm{X}_n^{k+1} & = & \Delta_n(\mathrm{X}_1^{k+1}, \mathrm{X}_2^{k+1}, \ldots, \mathrm{X}_{n-1}^{k+1}, \mathrm{X}_n^k) \end{array}$$

The idea is to use new information as soon as possible. The Jacobi iteration scheme just computes a new domain-element with respect to to the old one, while the Gauss-Seidel inserts the new information in the vector as it is computed and use it when computing information in the next point.

6.3 Chaotic iteration

The third iteration strategy tested was the chaotic iteration strategy. The idea behind this strategy is to make use of the structure of the programs being analysed. The two strategies mentioned above spend a lot of time computing program points which do not change their values because the points they depend on have not changed their values.

We introduce a set of operations Ω_i which compute a (possibly) new value for exactly one program point:

$$\Omega_i(\mathrm{X}) \quad = \quad (\mathrm{X}_1, \ldots, \mathrm{X}_{i-1}, \Delta_i(\mathrm{X}), \mathrm{X}_{i+1}, \ldots, \mathrm{X}_n)$$

Now we will try to select a sequence of Ω_i-operations that computes the same fixpoint as the Jacobi iteration strategy computes. This will be done by using the information in the call graph. The two possible results of applying a certain Ω_i to X are:

(i) A new X is computed. That is: the new X differs from the old in exactly one program point, say i.

(ii) The new X computed is the same as the old X.

Consider (i). This means that the only program points that possibly will be affected and hence interesting to compute, are the program points related to i via the call graph. As described in chapter 3 we can partition all program points into two sets: Either a program point is a *calling point* of a body atom, or it is an *exit point* of a clause. In the former case the affected points are the entry points of the clauses invokable by the atom to right of the point, *plus* the success point of the atom. In the latter, the possibly affected points are the success points of all atoms which are able to invoke the clause. We will make use of the following function, *depend_on*, to compute a set of program points affected by a change in one particular point:

Definition 6.1 *depend_on*

$$\begin{array}{rcl} depend\_on({}_\bullet\alpha) & = & \{\alpha_\bullet\} \cup \{{}_\bullet C \mid \alpha \rightsquigarrow C\} \\ depend\_on(C_\bullet) & = & \{\alpha_\bullet \mid \alpha \rightsquigarrow C\} \end{array}$$

□

Remark 6.2 Note that the second set in the first rule will be empty if α is an equation. □

We next introduce a stack, q, to keep track of affected program points. We also need some tools to update the stack. Note that the *push*-operation before pushing the new point, first removes all occurrences of the new point on the stack[2].

$$\begin{array}{lcl} top(p_1 :: p_2 :: \ldots :: p_n) & = & p_1 \\ pop(p_1 :: p_2 :: \ldots :: p_n) & = & p_2 :: \ldots :: p_n \\ push(p, p_1 :: p_2 :: \ldots :: p_n) & = & p :: rem(p, p_1 :: \ldots :: p_n) \end{array}$$

Now, given these primitives we can define the *update* function using *depend_on* to compute new points.

Definition 6.3 *update*

$$\begin{array}{lcl} update({}_\bullet\alpha, \overline{q}) & = & push({}_\bullet C_n, \ldots push({}_\bullet C_1 push(\alpha_\bullet, \overline{q}))) \\ & & \alpha \text{ not equation} \\ update(C_\bullet, \overline{q}) & = & push(\alpha_{n\bullet}, \ldots push(\alpha_{1\bullet}, \overline{q})) \end{array}$$

□

Remark 6.4 Again note that when α is an equation, the only program point affected is the success point of the equation. □

Finally the chaotic algorithm for compute the LFP(Δ) will then be defined thus:

Algorithm 6.5 *Chaotic iteration*

$\mathrm{X} \leftarrow (\perp, \ldots, \perp)$
$q \leftarrow 0$

[2] $rem(a, B)$ removes all occurrences of a in B.

```
while q ≠ ∅ do
    i ← top(q)
    q ← pop(q)
    if X ≠ Ω_i(X) then
        X ← Ω_i(X)
        q ← update(i, q)
        fi
    od
return X
```

An example

Consider the program implementing factorial again. The chaotic algorithm then behaves in the following way (Again we compute the **if** objects as we visit each program point for the first time.):

$$(\bot, \bot, \bot, \bot, \bot, \bot, \bot)$$

$$q \;=\; 0$$

The first Ω applied is Ω_0 which computes the information given by the user. The stack is updated with the entry points affected by the fact that the information at point 0 changed.

$$(\begin{bmatrix} \mathtt{X} \\ \emptyset \end{bmatrix}, \bot, \bot, \bot, \bot, \bot, \bot)$$

$$q \;=\; 3 :: 2 :: 1$$

Since α_3 is an equation, the only change on the stack is to add point 4.

$$(\begin{bmatrix} \mathtt{X} \\ \emptyset \end{bmatrix}, \bot, \bot, \begin{bmatrix} \mathtt{A} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}) \end{bmatrix}, \bot, \bot, \bot)$$

$$q \;=\; 4 :: 2 :: 1$$

Again the atom is an equation, so the only change on the stack is to pop 4 and push 5.

$$(\begin{bmatrix} \mathtt{X} \\ \emptyset \end{bmatrix}, \bot, \bot, \begin{bmatrix} \mathtt{A} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}) \end{bmatrix}, \begin{bmatrix} \mathtt{A,E} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}), \mathbf{if}(\mathtt{E}, \{\mathtt{A}\}) \end{bmatrix}, \bot)$$

$$q \;=\; 5 :: 2 :: 1$$

This time a calling point of an atom which has two "daughters" related via $\rightsquigarrow$. So 5 is popped and 3 and 2 plus the success point of 5 are pushed on the stack.

$$(\begin{bmatrix} \mathtt{X} \\ \emptyset \end{bmatrix}, \bot, \bot, \begin{bmatrix} \mathtt{A} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}) \end{bmatrix}, \begin{bmatrix} \mathtt{A,E} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}), \mathbf{if}(\mathtt{E}, \{\mathtt{A}\}) \end{bmatrix}, \bot,)$$

$$q \;=\; 3 :: 2 :: 6 :: 1$$

Since computing point 3 does not change the value of that point, 3 is just popped from the stack without adding anything instead.

$$(\begin{bmatrix} \mathtt{X} \\ \emptyset \end{bmatrix}, \bot, \bot, \begin{bmatrix} \mathtt{A} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}) \end{bmatrix}, \begin{bmatrix} \mathtt{A,E} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}), \mathbf{if}(\mathtt{E}, \{\mathtt{A}\}) \end{bmatrix}, \bot)$$

$$q \;=\; 2 :: 6 :: 1$$

Now 2 is an exit point of a clause, so this affects two success points, 1, and 6.

$$(\begin{bmatrix} \mathtt{X} \\ \emptyset \end{bmatrix}, \bot, \begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}) \end{bmatrix}, \begin{bmatrix} \mathtt{A,E} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}), \mathbf{if}(\mathtt{E}, \{\mathtt{A}\}) \end{bmatrix}, \bot)$$

$$q \;=\; 6 :: 1$$

Point 6 will be changed, and since that point is an exit point of a clause the atoms calling the clause, 6 and 1, will possibly be affected and are hence pushed on the stack.

$$(\begin{bmatrix} \mathtt{X} \\ \emptyset \end{bmatrix}, \bot, \begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}) \end{bmatrix},$$

$$\begin{bmatrix} \mathtt{A,E} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}), \mathbf{if}(\mathtt{E}, \{\mathtt{A}\}) \end{bmatrix}, \begin{bmatrix} \mathtt{A,B,C,E} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}), \mathbf{if}(\mathtt{E}, \{\mathtt{A}\}) \end{bmatrix},)$$

$$q \;=\; 6 :: 1$$

However, point 6 will not be changed so the element remains the same. The only action to take is to pop 6 from the stack.

$$(\begin{bmatrix} \mathtt{X} \\ \emptyset \end{bmatrix}, \perp, \begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}) \end{bmatrix},$$

$$\begin{bmatrix} \mathtt{A,E} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}), \mathbf{if}(\mathtt{E}, \{\mathtt{A}\}) \end{bmatrix}, \begin{bmatrix} \mathtt{A,B,C,E} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}), \mathbf{if}(\mathtt{E}, \{\mathtt{A}\}) \end{bmatrix},)$$

$$q = 1$$

Finally we compute point 1 and the algorithm terminates. The domain element obtained is the same as shown in the former chapter.

$$(\begin{bmatrix} \mathtt{X} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{X,Y} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \mathtt{A} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}) \end{bmatrix},$$

$$\begin{bmatrix} \mathtt{A,E} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}), \mathbf{if}(\mathtt{E}, \{\mathtt{A}\}) \end{bmatrix}, \begin{bmatrix} \mathtt{A,B,C,E} \\ \mathbf{if}(\mathtt{B}, \{\mathtt{A}, \mathtt{C}\}), \mathbf{if}(\mathtt{E}, \{\mathtt{A}\}) \end{bmatrix},)$$

$$q = nil$$

What can we say about the chaotic iteration algorithm, then? As shown above the algorithm used 9 calls to an Ω_i. The Jacobi uses $5 \times 7 = 35$ while The Gauss-Seidel uses $3 \times 5 = 15$.

6.4 Theoretical and empirical results

To compare the three methods we use the number of calls to Ω. The informal complexity analysis in [Nil92] assigns the three methods the following complexities: Jacobi $\in O(n^2)$, Gauss-Seidel $\in O(n^2)$ and finally Chaotic $\in O(cn)$, where n is the number of program points, and c is a constant.

The empirical tests were done using four different programs:

- The factorial program shown in this thesis, containing 7 program points and 2 calls to external functions.
- A program (`typesettab/4`) for computing typesetting-information, containing 21 program points and 4 calls to external functions.

- A DCG for computing binary numerals (`knuth/1`), containing 32 program points and 8 calls to external functions.

- A program for computing fast Fourier transforms (`fft/2`), containing 110 program points and 17 calls to external functions.

The test seems to confirm both the informal complexity analysis as well as the "benchmark test" done in [Nil92]. The figure below shows the number of calls to an Ω_i for each iteration method.

The result is about what we expected. The Chaotic iteration has a complexity of about $O(2n)$ where n is the number of points. The Jacobi and Gauss-Seidel have as expected a considerably worse complexity. In fact the Jacobi iteration is so slow that it cannot be of any practical interest.

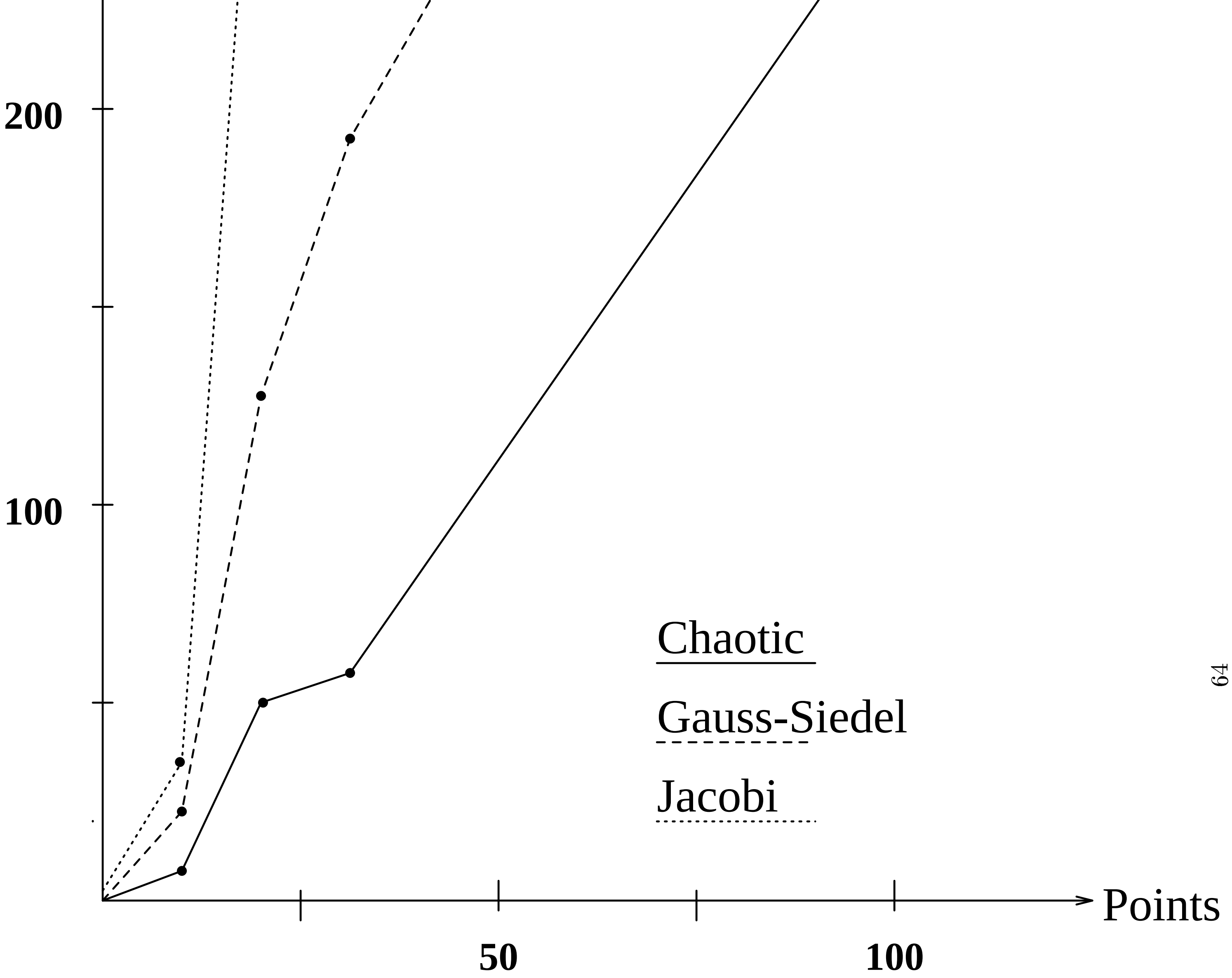

Figure 6.1: Benchmark results

Chapter 7

Conclusions and Future work

We have developed two abstract interpretations, based on the framework by Ulf Nilsson in [Nil92]. The first of them allows for inferring groundness properties of Prolog programs. The second, an extension of the former, applies to GAPLog programs.

The main contributions of the thesis are

The domain A simple domain was developed for inferring groundness properties of GAPLog programs.

A prototype implementation A prototype implementation was done for a subset of GAPLog containing only user defined predicates.

Benchmark A couple of benchmark tests were performed to verify both the informal complexity analysis as well as the practical tests done in [Nil92].

■ Conclusions

It is clearly possible to analyze logic programs with external functions using abstract interpretation.

The framework by Ulf Nilsson turned out to be a simple and elegant tool for constructing abstract interpretations for logic programs *with* and *without* equality.

Computing abstract interpretation for programs with delays is expensive, even with simple domains. The time to analyze a program, even one with relatively few program points, is unpleasantly high. For instance the time for analyzing the `fft/2` which was the largest program tested is 33 seconds using the chaotic iteration method. This could for instance depend on the following things:

- The analyzer also spend a lot of time recomputing the **if** items.
- The **if** items were also involved in the first version of the domain, so an equality check on the domain elements was quite complex.
- The domain element was implemented as a 2-3-tree which has an access time of $log(n)$, where n is the number of program points. When computing a new value for a program point, parts of the tree had to be reconstructed. One improvement would be to use a data structure with constant access time, for instance an array.

In any case: *Efficient algorithms are of great importance when computing abstract interpretations.*

Now, this kind of analysis is not necessarily uninteresting. Although there is no point using this time-wasting data flow analysis while *developing* GAPLog programs, a compilation *with* abstract interpretation of the final version of the program is a possibility and clearly motivates this work.

■ Future work

There are a number of ways both to complete and extend this work. Worth mentioning are the following:

Correctness proof This work does not include a formal proof of the abstract interpretation developed in chapter 5. To complete the work, the Galois connection $\langle \alpha \, ; \, \gamma \rangle$ has to be defined.

The Equations The analysis could be improved if we allow **if** objects to be passed between clauses. Then it would be possible to conclude groundness of `X` at the success point of the goal in the following program (with no variables ground in the goal):

```
:- p(X,Y), Y = 3.

p(A,B) :-
     A = B * B
```

Move equational atoms within clauses In many programs one could reduce the number of dynamic tests by changing the ordering of the body atoms in some clauses. For instance the conclusions drawn based on the result of analysing the factorial program are not optimal. If we exchange the places of the two last atoms, we will end up with a program where no variables have to be frozen. It would be desirable to develop machinery to determine how one can perform such moves.

Implementation It would be desirable to implement the static approach to see if the analysis has more realistic performance. Another interesting topic would be to investigate the possibility of computing the fixpoint on parallel computer architectures.

Implementation (again) The prototype implementation was done for user defined predicates. A real implementation would have to deal with predefined predicates as well. A way of describing the groundness behaviour of these predicates has to be developed.

Bibliography

[AH87] Abramsky, S. and Hankin, C. editors *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[AKLN87] Aït-Kati, H. Lincoln, and P. Nasr, R. Le Fun: Logic, Equations, and functions. In *Proc. 4th IEEE International Symposium on Logic Programming*, pp 17-23, San Fransisco, 1987.

[BGL+87] Bosco, P.G. Giovannetti, E. Levi, G. Moiso, C. and Palamidessi, C. A complete semantic characterization of K-LEAF, a logic language with partial functions. In *Porc. 4th IEEE Internat. Symposium on Logic Programming,* pp. 318–327, San Fransisco, 1987.

[BM88] Bonnier, S. and J. Małuszyński. Towards a clean amalgamation of logic programs with external procedures. In *5th Int. conf. and symp. on logic programming*, pp. 311-326, MIT Press, 1988.

[Bon89] Bonnier, S. *Horn clause logic with external procedures: towards a theoretical framework*, Licentiate thesis 197, Dept. of Computer and Information Science, University of Linköping, Sweden, 1989.

[Bon90] Bonnier, S. *Horn clause logic with external procedures: towards a formal framework*, Research report, Dept. of Computer and Information Science, University of Linköping, Sweden, 1990.

[Bon92] Bonnier, S. *A Formal Basis for Horn Clause Logic with External Polymorphic procedures* PhD thesis 276, Dept. of Computer and Information Science, University of Linköping, Sweden, 1992.

[Boy91a] Boye, J. *Operational completeness of logic programs with external procedures*, Master's thesis LiTH-IDA-Ex-9104, Linköping University, 1991.

[Boy91b] Boye, J. S-SLD-resolution — An operational semantics for logic programs with external procedures. In *Proc. PLILP'91*, LNCS 528, pp. 383–393, Springer Verlag, 1991.

[Bru87] Bruynooghe, M. *A framework for the abstract interpretation of logic programs.* Report CW62, Katholieke Universiteit, Leuven, 1987.

[Bä86] Bäckström, K-J. *Diskret matematik*, Studentlitteratur, 1986.

[CC93] Cousot, P. and Cousot, R. *Abstract interpretation and Application to Logic Programs*, Article in Journal of Logic Programming special issue on Abstract Interpretation 1993.

[Coh86] Cohen, S. The APPLOG Language In DeGroot, D. and Lindstrom, G. editors. *Logic programming, functions, relations and equations.* Prentice Hall, 1986.

[Fri85] Fribourg, L. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 172-184, Boston, 1985.

[GM86] Goguen J.A. and Meseguer J. *Eqlog: Equality, Types and Generic Modules for Logic Programming* In Logic Programming, Functions, Relations and Equations, pp. 295-363. Prentice Hall, 1986

[Han92] Hanus, M. *An Abstract Interpretation Algorithm for Residuating Logic Programs*, Research report, Max-Planck-Institut für Informatik, Germany, 1992.

[JLM84] Jaffar, J., J-L. Lassez and M. Maher. A theory of complete logic programs with equality. *Journal of logic programming*, 1(3), pp. 211-223, 1984.

[JS87] Jones, N.D. and H. Søndergaard. A semantics-based framework for the abstract interpretation of Prolog. In Abramsky and Hankin editors, *Abstract interpretation of declarative languages*, pp. 123-142, Ellis Horwood, 1987.

[KK91] Kågedal, A and F. Kluźniak. *Enriching Prolog with S-unification*, Research report LiTH-IDA-R-91-12, University of Linköping, Sweden, 1991.

[Kle52] Kleen, S.C. *Introduction to mathematics.* North-Holland Publ. Co, Amsterdam-New York, 1952.

[Kom82] Komorowski, J. QLOG—The Programming Environment for Prolog inLISP. In Clark, K. and Tärnlund, S-Å.editors. *Logic Programming.* Prentice-Hall, 1982.

[Kåg92] Kågedal, A. *Logic Programming with External Procedures: an Implementation*, Lic thesis 326 Dept. of computer and information science, University of Linköping, Sweden, 1992.

[MB+93] Maluszynski, J. Bonnier, S. Boye, J. Kågedal, A. Kluźniak, F. Nilsson, U. Logic Programs with External Procedures. In K.R. Apt, J.W. de Bakker and J.J.M.M. Rutten editors *Current trends in logic programming languages, integration with functions, constraints and objects*

[Mel87] Mellish C. Abstract interpretation of Prolog programs. In Abramsky and Hankin editors, *Abstract interpretation of declarative languages*, pp. 181-198, Ellis Horwood, 1987.

[MH84] Mellish, C. and Hardy, S. Integrating Prolog in the Poplog Environment. In Campbell, J. editor. *Implementations of Prolog.* Ellis Horwood, 1984.

[MM82] Martelli, A. Montanari U *An efficient unification algorithm.* ACM transactions on programming languages and systems 4(2), pp. 258-282, 1982.

[MR92] Moreno-Navarro, J.J. and Rodríguez-Artalejo, M. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming,* Vol. 12, pp. 191-223, 1992.

[Nil92] Nilsson, U. *Abstract interpretations & abstract machines: contributions to a methodology for the implementation of logic programs*, PhD thesis 265, Dept. of Computer and Information Science, University of Linköping, Sweden, 1992.

[Nil89a] Nilsson, U. *A Systematic Approach to Abstract Interpretation of Logic Programs.* Licentiate Thesis 196, Dept. of Computer and Information Science, University of Linköping, Sweden, 1989

[Nil89b] Nilsson, U. Towards a Framework for the Abstract Interpretation of Logic Programs. in *Programming Language Implementation and Logic Programming 88,* pages 68–82, Springer-Verlag, 1989.

[Nil90] Nilsson, U. *Systematic semantic approximations of logic programs.* In Deransart and Małuszyński editors, *Proc. PLILP'90*, LNCS 456, pp. 293-306, Springer Verlag, 1990.

[NM90] Nilsson, U. and Małuszyński, J. *Logic, Programming, and Prolog*, John Wiley and Sons, 1990.

[RS82] Robinsson, J. and Sibert, E. LOGLISP: Motivation, Design and Implementation. In Clark, K. and Tärnlund, S-Å.editors. *Logic Programming.* Prentice-Hall, 1982.

[Tar55] Tarski, A. *A Lattice Theoretical Fixpoint Theorem and its Applications.* Pacific J. Math, 5:285–309, 1955.

Appendix

■ The S-unification algorithm

Algorithm 7.1 (S-unification)
Initially, let $U_0 = U$ and $i = 0$. Repeatedly do the following: Select any equation $u \doteq t$ in U_i such that one of Rules 1 to 9 applies. If no such equation exists then stop with U_i as result. Otherwise perform the corresponding action, that is: stop with failure or construct U_{i+1} from U_i and increment i by 1.

1. u and t are identical. Remove $u \doteq t$.

2. u is a variable which occurs elsewhere in U_i and t is a constructor term distinct from u. If u occurs in t then stop with failure, otherwise replace all other occurrences of u by t (leaving $u \doteq t$ unchanged).

3. u is a non-variable and t is a variable. Replace $u \doteq t$ by $t \doteq u$.

4. u is of the form $c_1(u_1 \dots u_n)$ and t is of the form $c_2(t_1 \dots t_n)$ where c_1 and c_2 are constructors. If c_1 and c_2 are distinct then stop with failure, otherwise replace $u \doteq t$ by $u_1 \doteq t_1, \dots, u_n \doteq t_n$.

5. u and t are both function calls. Let z be a variable not in $var(U_i) \cup var(U)$. Replace $u \doteq t$ by the two equations $z \doteq u$ and $z \doteq t$.

6. u or t has a function call d as proper sub-term. Let z be a variable not in $var(U_i) \cup var(U)$. Replace all occurrences of d in U_i by z, then add the equation $z \doteq d$.

7. u is a function call and t is a constructor term. Replace $u \doteq t$ by $t \doteq u$.

8. u is a constructor term and t is a function call which occurs elsewhere in U_i. Replace all other occurrences of t by u (i.e. leave $u \doteq t$ unchanged).

9. t is a ground function call. Replace $u \doteq t$ by $u \doteq s$, where $t \doteq s$ is in $\mathcal{E}$.□

When given a set U of equations to unify, the algorithm transforms U by applying the rules (in any order) until no rule is applicable. Depending on the result of the transformation, it then gives the following result:

1. $\emptyset$, if the transformation ends in failure

2. $\{\sigma_{U'}\}$, if the transformation ends in a solved form U'.

3. "don't know", if neither 1 nor 2 applies.

■ The `gfreeze/2` predicate

The `gfreeze/2` predicate is implemented by means of the `freeze/2` predicate. The latter takes a list as its first argument and a call to an external function as its second, but it only waits until the first variable in that list is ground before calling the function.

```
gfreeze(V,G) :- var(V), !, freeze(V,gfreeze(V,G)).
gfreeze(A,G) :- atomic(A), !, call(G).
gfreeze([H|T],G) :- !, gfreeze(H,gfreeze(T,G)).
gfreeze(T,G) :- functor(T, _, Arity), gfreeze_(Arity,T,G).

gfreeze_(0, _T, Goal):-
    !,
    call(Goal).
gfreeze_(ArgNo, T, Goal) :-
    arg(ArgNo, T, Arg),
    NextArgNo is ArgNo - 1,
    gfreeze(Arg, gfreeze_(NextArgNo, T, Goal).
```