Abstract Interpretation, Bug Detection and Bug Diagnosis in Normal Logic Programs

by

Lunjin Lu

A thesis submitted to
the Faculty of Science
the University of Birmingham
for the degree of Doctor of Philosophy

School of Computer Science
The University of Birmingham
Birmingham B15 2TT
The United Kingdom

September, 1994

Synopsis

This thesis reports our research into abstract interpretation, bug detection and bug diagnosis in normal logic programs. Abstract interpretation is an approach for static program analysis. Bug detection and bug diagnosis are two main activities in program debugging. The objective of this work is to develop an automated bug detection method for normal logic programs and to improve the declarative bug diagnosis method for normal logic programs; in pursuit of this objective, abstract interpretation has turned out to be useful in bug detection.

A least fixed-point generic abstract semantics is developed for forward analysis of normal logic programs and is specialised for inferring mode and type information.

A generic abstract domain and a generic abstract unification algorithm are developed and specialised for inferring mode and type information both of which are inferred together with sharing and aliasing information.

Bug detection is accomplished by a combination of static checking and program testing. In static checking, a program is checked for the satisfaction of its intended properties. An intended property is either a set of assertions associated with points in the program or a set of success patterns for procedures in the program. The result of abstract interpretation is used both in static checking and in program testing.

In respect of bug diagnosis, a declarative diagnoser is developed. The diagnoser first builds a tree to model the execution history of a bug symptom and then diagnoses by searching the tree. The diagnoser is generic in that it can be used with different search strategies. It also has a smaller search space than those reported in the literature.

Acknowledgements

My Ph.D. studies have been supported by the Sino-British Friendship Scholarship Scheme.

I am in debt to Dr. Peter Greenfield and Dr. Salleh Syed-Mustaffa for their kind supervision, useful discussion and sometimes insightful criticism during the last four years. Dr. Greenfield supervised me for the first three years before he left for France. He continued helping me afterwards up to the completion of this thesis. Dr. Syed-Mustaffa supervised me for the last year. I am also in debt to Professor Jorge Bocca who co-supervised me during the last year and has been a member of my thesis group. Without their help, I would not be able to have completed this thesis. However, I myself am responsible for any mistakes in this thesis since the work has been independently done.

I owe my gratitude to Mr. Bob Hendley for his useful comments at the meetings of my thesis group of which he is a member. Of course, his help has not been confined to those meetings. I am also grateful to Dr. Peter Hancox for his help.

Finally, I thank the School of Computer Science which has given me much support throughout the last four years.

Contents

List of Figures

Chapter 1

Introduction

Logic programming originates from the discovery that a subset of predicate logic can be given an operational interpretation, which is embodied in the programming language Prolog. Unlike an imperative program that describes how to solve a problem, a logic program describes what a problem is. A logic program can be understood declaratively as a collection of axioms from which theorems can be derived. With a logic programming language, the programmer is relieved from thinking how his programs are executed and thus can devote his time and ingenuity to understanding problems to be solved.

The declarativeness of logic programming helps reduce errors in programming because there is no need for the programmer to translate his understanding of a problem into an algorithm. However, this does not guarantee that a logic program will be free of bugs in the first instance. Firstly, the programmer may make mistakes in understanding the problem and therefore write a program which describes a different problem from the one that was intended to be solved. Secondly, logic programming languages also include non-logical constructs such as *cut*. These constructs, when not used properly, compromise declarativeness because programs containing these constructs need to be understood operationally. Thirdly, current implementations of logic programming languages do not fully relieve the programmer from the operational aspects of their programs. For instance, some logic programming systems usually require that the programmer use negation as failure safely.

Logic programming has been widely used in areas such as database systems, theorem proving, expert systems, natural language processing, parallel and concurrent computing, and so on. However, in comparison to the widespread application of logic programming, less attention appears to have been given to the problem of development of logic programs. The programming environments for logic programming languages often resemble those for other programming languages [143] and do not take into account the unique features of logic programming. They do not give satisfactory support for program debugging during which bugs in programs are detected, diagnosed and corrected. There is little support for bug detection. The programmer needs to run the program on sample inputs and makes judgements on the correctness of the program from its outputs. Breakpointing, single stepping and state inspection are the usual facilities provided for bug diagnosis. These facilities make it necessary for the programmer to understand the execution of the program and therefore undermine declarativeness.

We report our research into logic program debugging. The language that we have chosen in our research is normal logic programs [74] - a subset of the first order predicate logic. The objective of this work is to develop an automated bug detection method for normal logic programs and to improve the declarative bug diagnosis method for normal logic programs; in pursuit of this objective, abstract interpretation has turned out to be useful in bug detection.

A *bug* is an error in a program that causes the program to deviate from its expected behaviour. The purpose of program debugging is to remove bugs from programs. It is generally agreed that program debugging is the process of detecting, diagnosing and correcting bugs in programs [110].

Bug detection is the activity of judging if the actual behaviour of a program is that which was intended. A difference between the actual behaviour and the intended behaviour is called a *bug symptom*. The behaviour may be the execution time of a procedure invocation, the termination of a procedure invocation, the output value of a procedure called with a certain input value, and so on.

Bug diagnosis is the activity of finding the bug that is responsible for a given bug symptom. Bug diagnosis comprises bug localisation and bug identification. *Bug localisation* identifies as small a program part as possible that causes the bug symptom to manifest itself while *bug identification* classifies the bug. Bug localisation has been the main focus of efforts to automate bug diagnosis. The reason for this is that bug localisation has been considered the most difficult and therefore the most expensive activity in bug diagnosis [120]. If a bug has been sufficiently localised, it should be easy to identify the bug.

Bug correction is the activity of modifying the buggy part of the program in an attempt to make it behave as intended. Unless the bug detected is a consequence of a design flaw, the bug correction activity is relatively simple [120]. Based on this observation, we have focused on bug detection and bug diagnosis.

There are many kinds of symptom that a logic program may exhibit. The program may produce a wrong result (an *inconsistency symptom*), fail to produce a correct result (an *insufficiency symptom*), fall into an infinite loop (a *non-termination symptom*) or fail to respect mode or type specifications for some procedures, and so on.

Example 1.0.1 The following is a buggy quicksort program in Prolog. Natural numbers are represented in terms of 0 and the successor function s/1. For instance, the natural number 3 is represented as s(s(s(0))).

```
qs([X|L],L0) :-
    pt(L,X,L1,L2),
    qs(L1,L3),
    qs(L2,L4),
    ap([X|L3],L4,L0). %ap(L3,[X|L4],L0).
qs([],[]).

pt([X|L],Y,L1,[X|L2])  :-
                      %le(Y,X),
    pt(L,Y,L1,L2).
pt([X|L],Y,[X|L1], L2) :-
    le(Y,X),          %gt(Y,X)
    pt(L,Y,L1,L2).
pt([],X,[],[]).
```

```
ap([X|L1],L2,[X|L3]) :-
    ap(L1,L2,L2). %ap(L1,L2,L3)
ap([],L,[]).      %ap([],L,L).

le(0,0).
le(0,s(X))    :- le(0,X).
le(s(X),s(Y)) :- le(X,Y).

gt(s(0),0).
gt(s(X),0)    :- gt(X,0).
gt(s(X),s(Y)) :- gt(X,Y).
```

The intended interpretation for *qs*(Li,Lo) is that Li and Lo are lists of natural numbers and Lo is sorted and contains the same elements as Li. The intended interpretation for *pt*(Li,N,Lo1,Lo2) is that Li, Lo1 and Lo2 are lists of natural numbers, N is a natural number, Lo1 contains those elements of Li which are less than or equal to N and Lo2 contains those elements of Li which are greater than N. The intended interpretation for *ap*(L1,L2,L3) is that L1,L2 and L3 are lists and L3 is the concatenation of L1 and L2. The intended interpretation for *le*(N1,N2) is that N1 and N2 are natural numbers and N1 is less than or equal to N2. The intended interpretation for *gt*(N1,N2) is that N1 and N2 are natural numbers and N1 is greater than N2.

The program contains a number of buggy literals. A buggy literal is indicated by a comment beginning with a %. The comment also contains the correct version of the buggy literal. Whether the program will exhibit a bug symptom depends on how it is queried. Here are three queries to and results from the program.

```
| ?- qs([s(0),0],L).

no
| ?- qs([s(0)],L).

L = [s(0)|_6946]

yes
| ?- qs([s(0)],[s(0)]).

yes
| ?-
```

qs([s(0),0],L) is an insufficiency symptom because *qs*([s(0),0],[0,s(0)]) is intended and the program fails finitely on it. *qs*([s(0)],[s(0)|_6946]), where _6946 is a variable, is an inconsistency symptom because it is produced by the program and is unwanted. (It is a convention that a variable without a mnemoric name is written as a positive integer preceded by an _.) When executed on *qs*([s(0)],[s(0)]), the program exhibits no symptom because *qs*([s(0)],[s(0)]) is both intended by the programmer and produced by the program. ∎

Example 1.0.2 The intended interpretation for *double*(X,Y) in the following program is that Y=2×X. *double*(s(0),Y) is a non-termination symptom because the execution of the program on *double*(s(0),Y) will never stop.

```
double(0,0).
double(s(X),s(s(Y))) :-
        double(s(X),Y). % double(X,Y)
```

∎

Example 1.0.3 The following program has two procedures: *ap*/3 and *rev*/2 for list processing. The intended interpretation for *ap*(L1,L2,L3) is that L3 is equal to the result of linking L1 and L2 by putting the first element of L2 after the last element of L1. The intended interpretation for *rev*(L1,L2) is that L1 and L2 contain the same elements but in reverse order. The program has associated with it a number of comments indicating locations in the program called program points. We call the literal following a program point the literal at that program point. For instance, the literal at *u* is *ap*(T0,H,L).

```
rev([],[]).
rev([H|T],L) :-
        rev(T,T0),
            % point u
        ap(T0,H,L).
        % ap(T0,[H],L)
```

```
ap([],Xs,Xs). % point v
ap([Y|Ys],Zs,[Y|Zs0]) :-
              % point w
        ap(Y,Zs,Zs0).
```

The programmer would expect *ap*/3 to be called with its first and second arguments being lists. In the case of the first clause for *ap*/3, Xs should be a list when the execution of the program reaches point *v*. Because the second literal in the body of the second clause for *rev*/2 is buggy, the execution of the program on *rev*([s(0)],Lo) will violate the programmer's expectation because Xs in the first clause for *ap*/3 is instantiated to s(0) rather than a list when the execution reaches *v*. ∎

Inconsistency and insufficiency symptoms are readily characterised declaratively whilst the others are better described operationally as shall be shortly shown. The operational semantics of normal logic programs that we consider is the SLDNF-resolution via the left-to-right computation rule [74]. Figure 1.1 illustrates a rough classification of bug symptoms. We consider the detection and the diagnosis of both declarative and operational symptoms.

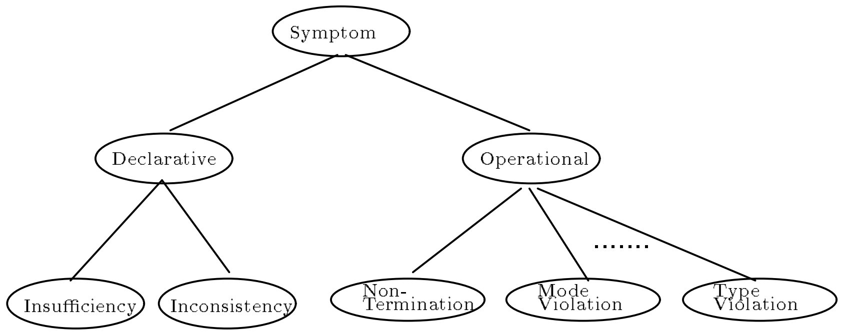


Figure 1.1: Bug Symptoms: A Classification

1.1 Bug Detection

The detection of declarative symptoms is accomplished by means of a combination of static program analysis and program testing. In order to detect declarative symptoms, it is necessary to be able to decide what theorems are intended and what theorems are unwanted. The judgement of a theorem being intended or unwanted is provided by an entity, called an oracle, that is external to the debugging system. If the program has an executable specification, the oracle can be automated. The role of the oracle is usually played by the programmer. The following example shows that static program analyses such as type and mode analyses can help the detection of declarative symptoms.

Example 1.1.1 Consider the program in example 1.0.1. One property of the intended interpretation for *qs*(Li,Lo) is that if Li is a list of natural numbers and the program succeeds on *qs*(Li,Lo) then Lo is a list of natural numbers. Type analysis would show that this property may be violated by suggesting that Lo is not necessarily a list of natural numbers. ∎

In general, static program analysis cannot definitely predict if a program would have declarative symptoms or what the declarative symptoms would be. They can only predict that the program is likely to have declarative symptoms. We use program testing to detect declarative symptoms. In the above example, the program would be tested on *qs*/2 with its first argument being a list of natural numbers. Program testing would detect the inconsistency symptom *qs*([s(0)],[s(0)|_6946]).

The detection of operational symptoms is more involved and different kinds of operational symptom need to be considered separately. We consider non-termination, violations of the weaker safeness condition for the negation as failure rule, mode violations and type violations.

The detection of non-termination is in general unsolvable. The solution in [122] that is also adopted in our work is to impose a limit on the maximal amount of certain resources that one execution of the program may take. When an execution of the program on an input exceeds the limit, it is halted and is suspected of non-terminating. We impose a limit on the number of nested procedure calls. The execution of the program in example 1.0.2 on *double*(s(0),Y) will exceed any finite limit on the number of nested procedure calls.

The detection of mode (type) violation is performed with a set of mode (type) assertions associated with some program points in the program. These assertions specify properties that the program states, at these program points, must satisfy. A program state is an assignment (a substitution) of values from the domain of discourse to the variables in the program. An assertion associated with a program point is a description of the set of admissible program states at the program point. Whenever the execution reaches the program point, the program state must satisfy the assertion.

Given a program and a set of assertions associated with it, one would expect to detect operational symptoms by comparing, at each program point, the set of program states satisfying the associated assertions with the set of program states that actually occur at the program point. But, this is impractical because the sets of both actual and admissible program states for a program point are usually of infinite size and not computable in finite time. We also solve this problem by combining static program analysis and program testing.

Given a program and a set of mode (type) assertions associated with it, an invariant on the set of actual program states for every program point is derived by the static mode (type) analysis. If the assertion associated with a program point is implied by the invariant for the same program point then the assertion will be satisfied during all the executions of the program. If this holds for every program point that is associated with an assertion, then the program is correct with respect to the set of mode (type) assertions associated with it. Otherwise, it is possible that some executions of the program may violate mode (type) assertions associated with the program. In this case, a set of test data is generated on which the program is tested. Whenever the execution reaches a program point with which an assertion is associated, the current program state is checked against the assertion. If the program state does not satisfy the assertion then an operational symptom has been detected.

Example 1.1.2 Consider the program in example 1.0.3. Suppose that the program will be only executed on *rev*/2 with its first argument being a list of natural numbers. Type analysis will indicate that Xs in the first clause for *ap*/3 would be instantiated to a natural number rather than a list whenever the control of execution is transferred to v from u. The same analysis indicates that execution will not reach v from w. At this point, we would infer that the literal at u is buggy. However, we cannot be sure about this because type analysis does not tell us if the execution of the program would definitely reach u or if the first clause for $ap/3$ would be called at u. Therefore, we test the program on $rev/2$ with its first argument being a list of natural numbers and find that *rev*([s(0)],L) will cause the violation of the type assertion at v that Xs must be a list. ∎

In summary, bug detection is accomplished by means of a combination of static program analysis and program testing. The role of static program analysis in bug detection is to detect possible violations of certain expected properties of a program whilst that of program testing is to confirm these violations with specific bug symptoms.

1.2 Program Analysis

The approach to static program analysis that we use in our work is abstract interpretation [19]. In abstract interpretation, program analyses are viewed as program executions over non-standard data domains. A collecting semantics is first defined for a program. Then an approximation of the collecting semantics is calculated by simulating over a non-standard data domain the computation of the collecting semantics over the standard data domain. Although there has been much work on the application of abstract interpretation to the analyses of logic programs [9, 21, 57, 60, 61, 63, 64, 65, 87, 89, 94, 101, 102, 135, 137, 138], the results are not suitable for our purposes because they are usually compilation-oriented and, more importantly, they do not deal explicitly with negation as failure.

We have developed two generic abstract semantics for forward analyses of normal logic programs via the left-to-right computation rule. They both derive run-time properties of programs. The difference between them is that the first one derives more information than the second one. These abstract semantics are specialised for mode and type analyses.

Mode and type analyses are used together with program testing to detect bug symptoms. The discovery of bug symptoms leads the program to be diagnosed.

1.3 Bug Diagnosis

Bug diagnosis is accomplished declaratively. Traditional tracers are not suitable for logic program diagnosis because they compromise the declarativeness of logic programming languages. Although the programmer can focus on the declarative semantics of the program when programming, they must know how the program is executed when carrying out a diagnosis. Since different tracers may use different execution models, the programmer must have a thorough knowledge of the execution model supported by the tracer he uses.

The declarative diagnosis approach analyses the history of execution of the program which exhibits unexpected behaviour in order to isolate an erroneous segment of this execution history. The part of the program corresponding to the erroneous segment is then localised. The erroneous segment is isolated by repeatedly selecting and inspecting segments from the execution history. The selection of segments is automated while inspection of the segments is accomplished by using an oracle which judicially makes judgement on selected segments.

We use a tree to model the execution history for a bug symptom. The diagnosis is accomplished by searching the tree. The selection of a segment of the execution history is the selection of a sub-tree.

In respect of declarative symptoms, the execution history for an inconsistency symptom is modelled by a tree similar to a proof tree and that for an insufficiency symptom by a tree similar to an SLDNF-tree. We have developed a declarative diagnoser that is generic in the sense that it supports different strategies for searching the tree such as top-down, bottom-up, divide-and-conquer and top-down zooming strategies. The generality allows for the development of new search strategies. The diagnoser also has a smaller search space than those reported in the literature. This is achieved by using the acquired knowledge of the program to prune the search space (the tree) before it is searched.

The diagnosis of an operational symptom is performed by searching a tree which models the derivation from the initial goal to where the operational symptom is detected. The completed computations are abstracted as a single node. The diagnosis also involves the oracle that, in addition to answering queries about the theorems of the program, may also be asked to decide whether or not a program state at a program point is admissible.

1.4 Prototype Debugger

We have so far outlined our work in static program analysis, bug detection and bug diagnosis in normal logic programs. A prototype debugger supporting these activities has been developed. It has five main components as illustrated in figure 1.2:

- The *program analyser* comprises a generic abstract interpreter and a number of abstract unification algorithms, each for a particular analysis. The analyses supported are type [10, 52, 57, 83] and mode [10, 15, 25, 83, 95, 130] analyses. The program analyser is used to derive invariants about program states or an approximation of each procedure in the program.

- The *bug detector* compares the result from the program analyser with the oracle's intention, generates test data and runs the program on the test data in a controlled way. The *abstract bug detector* compares the result from the program analyser with the oracle's intention. The *test data generator* generates test data using the results from the *abstract bug detector* and the *tester* runs the program on the test data so as to detect bug symptoms.

- The *diagnoser* localises bugs responsible for symptoms.

- The *corrector* is an interface for the user to modify the program. Since we are mainly concerned with detection and diagnosis, the corrector is very simple.

- The *oracle interface* is an interface between the debugger and the oracle.

In respect of declarative symptoms, given a program and a kind of program property, the program analyser computes an abstract success set for each procedure in the program. The bug detector then compares the actual abstract success set with the intended abstract success set for the same procedure, uses the discrepancies between them, called abstract bug symptoms, to generate test data and runs the program on the test data to detect a set of bug symptoms. The bug diagnoser then localises the bugs that are responsible for these bug symptoms. Finally, the bug corrector corrects the bugs localised by the bug diagnoser.

In respect of operational symptoms, given a program and a set of mode (type) assertions associated with it, the program analyser derives the invariants for the program. The bug detector then compares the invariants with the assertions, uses the discrepancies between them to generate test data and runs the program on the test data to detect a set of bug symptoms. The bug diagnoser then localises the bugs that are responsible for these bug symptoms. Finally, the bug corrector corrects the bugs localised by the bug diagnoser.

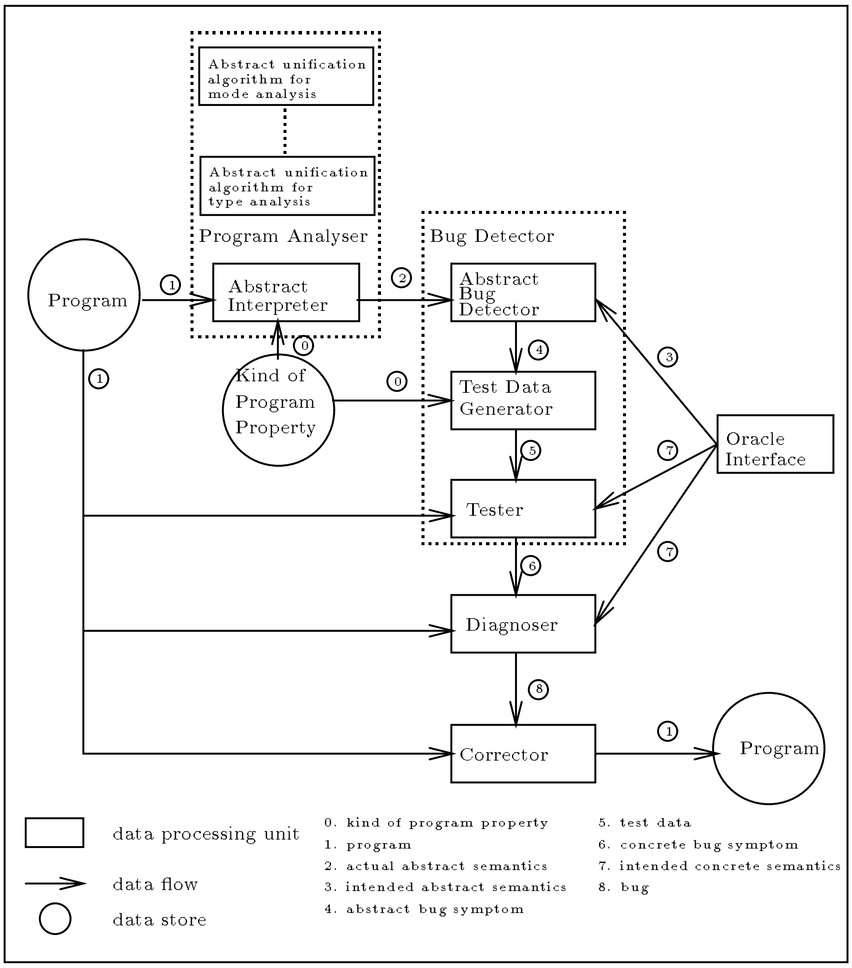


Figure 1.2: Prototype Debugger: Dataflow Diagram

1.5 Overview of the Thesis

The remainder of the thesis is organised as follows. Chapter 2 contains preliminary concepts on abstract interpretation, logic programming and program debugging. Chapter 3 reviews related work on logic programming debugging, as well as program debugging in general, and on abstract interpretation of logic program. In Chapter 4, we present two fixed-point generic abstract semantics for the abstract interpretation of normal logic programs. These generic abstract semantics deal explicitly with negation as failure. Chapter 5 specialises our generic abstract semantics to perform mode and type analyses. Chapter 6 uses the results of these analyses in bug detection. Chapter 7 presents our diagnosers for declarative and operational symptoms. In Chapter 8, we summarise our work and discuss possible extensions to our work. Appendix A contains proofs of lemmas and theorems appearing in the thesis that were considered too long to appear in the body of the thesis as well as some basic results on unification.

Chapter 2

Preliminaries

This chapter contains some preliminary concepts from logic programming, program debugging and abstract interpretation. Section 2.1 introduces the mathematical foundations of abstract interpretation. Section 2.2 covers some basic concepts in logic programming. Section 2.3 defines symptoms and bugs in logic programs.

2.1 Abstract Interpretation

Abstract interpretation [19] is a program analysis methodology for statically deriving run-time properties of programs. The derived program properties are then used by other program processors such as compilers, partial evaluators, etc. Program analyses are viewed as program executions over non-standard data domains. Cousot and Cousot first laid the solid mathematical foundations for abstract interpretation [19, 20]. Their idea is to define a collecting semantics for a program which associates with each program point the set of the storage states that are obtained whenever the execution reaches that point. Then an approximation of the collecting semantics is calculated by simulating over a non-standard data domain the computation of the collecting semantics over the standard data domain. The standard data domain and the interpretation of the program on the standard data domain are called *concrete* whilst the non-standard data domain and the interpretation of the program on the non-standard data domain are called *abstract*.

The semantics of a program is given by an interpretation. For instance, let the value of the variable X be 4 and the value of the variable Y be 5, then the integer 20 is given as the semantics of the expression $X \times Y$ by the standard interpretation which interprets a variable identifier as the value of the variable and $\times : int \times int \mapsto int$ as integer multiplication where int

denotes the set of integers.

When a program is analysed, the values of the variables in the program are not usually available. However, some information about these variables is available from which useful information can be inferred. For example, if we know that both variables X and Y will have positive integers as their values then we can safely conclude that expression $X \times Y$ will have a positive integer as its value. This conclusion is given by an abstract interpretation which replaces $\times$ with $\hat{\times} : sign \times sign \mapsto sign$ where $sign$ denotes the domain of signs which consists of $\hat{n}$, $\hat{z}$ and $\hat{p}$. A negative integer has sign $\hat{n}$, 0 has sign $\hat{z}$ and a positive integer has sign $\hat{p}$. $\hat{\times} : sign \times sign \mapsto sign$ is defined in the following.

$$\begin{array}{lll} \hat{n} \mathbin{\hat{\times}} \hat{n} = \hat{p} & \hat{n} \mathbin{\hat{\times}} \hat{z} = \hat{z} & \hat{n} \mathbin{\hat{\times}} \hat{p} = \hat{n} \\ \hat{z} \mathbin{\hat{\times}} \hat{n} = \hat{z} & \hat{z} \mathbin{\hat{\times}} \hat{z} = \hat{z} & \hat{z} \mathbin{\hat{\times}} \hat{p} = \hat{z} \\ \hat{p} \mathbin{\hat{\times}} \hat{n} = \hat{n} & \hat{p} \mathbin{\hat{\times}} \hat{z} = \hat{z} & \hat{p} \mathbin{\hat{\times}} \hat{p} = \hat{p} \end{array}$$

Domain int is called the concrete domain while domain $sign$ is called the abstract domain. A constant in int is called a concrete constant and a constant in $sign$ is called an abstract constant. Function $\times$ is called a concrete function and function $\hat{\times}$ is called an abstract function.

If an expression contains a concrete constant then it is replaced by an abstract constant. For example, the expression 4×3 is interpreted as $\hat{p}$ by first replacing 4 with $\hat{p}$ and 3 with $\hat{p}$ and $\times : int \times int \mapsto int$ with $\hat{\times} : sign \times sign \mapsto sign$ as defined above.

We have shown that an abstract interpretation of an expression consists in replacing a concrete constant with an abstract constant and replacing a concrete function with an abstract function. Now consider the abstract interpretation of the expression 4+(-3). By replacing 4 with $\hat{p}$ and -3 with $\hat{n}$, we do not know how to assign a sign to expression 4+(-3). We introduce another sign $\top$ to represent this situation. $\top$ corresponds to the set of integers. Let $sign^* = sign \cup \{\top\}$. Now, the abstract function $\hat{+} : sign^* \times sign^* \mapsto sign^*$ is given below.

$$\begin{array}{lllll|lllll|lllll|lllll} \top & \hat{+} & \top & = & \top & \top & \hat{+} & \hat{n} & = & \top & \top & \hat{+} & \hat{z} & = & \top & \top & \hat{+} & \hat{p} & = & \top \\ \hat{n} & \hat{+} & \top & = & \top & \hat{n} & \hat{+} & \hat{n} & = & \hat{n} & \hat{n} & \hat{+} & \hat{z} & = & \hat{n} & \hat{n} & \hat{+} & \hat{p} & = & \top \\ \hat{z} & \hat{+} & \top & = & \top & \hat{z} & \hat{+} & \hat{n} & = & \hat{n} & \hat{z} & \hat{+} & \hat{z} & = & \hat{z} & \hat{z} & \hat{+} & \hat{p} & = & \hat{p} \\ \hat{p} & \hat{+} & \top & = & \top & \hat{p} & \hat{+} & \hat{n} & = & \top & \hat{p} & \hat{+} & \hat{z} & = & \hat{p} & \hat{p} & \hat{+} & \hat{p} & = & \hat{p} \end{array}$$

This example shows that abstract interpretation is imprecise. While the expression 4+(-3) actually has sign $\hat{p}$ it is interpreted to have sign $\top$ which is less precise than $\hat{p}$. A partial order $\preceq$ can be introduced to compare the precision of abstract constants. $x \preceq y$ denotes that x is more precise than y. For the above example, we have

$$\hat{n} \preceq \top \quad \hat{z} \preceq \top \quad \hat{p} \preceq \top \quad \hat{n} \preceq \hat{n} \quad \hat{z} \preceq \hat{z} \quad \hat{p} \preceq \hat{p}$$

Although the information given by the abstract interpretation of a program is less precise than the information extracted directly from the concrete interpretation of the program, abstract interpretation is useful because the concrete interpretation of the program may not be computable in finite time since the concrete domain is usually infinite.

2.1.1 Complete Lattice

Let S, S_1, S_2 be sets. The *power-set* $\wp(S)$ of S is the set of all the subsets of S. $\wp(S) = \{X | X \subseteq S\}$. The *Cartesian product* $S_1 \times S_2$ of S_1 and S_2 is the set of all the tuples with the first components in S_1 and the second components in S_2. $S_1 \times S_2 = \{< s_1, s_2 > \ | s_1 \in S_1 \wedge s_2 \in S_2\}$.

Example 2.1.1 Let $S_1 = \{a, b\}$ and $S_2 = \{1, 2, 3\}$. Then

$$\wp(S_1) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$$

$$S_1 \times S_2 = \{< a, 1 >, < a, 2 >, < a, 3 >, < b, 1 >, < b, 2 >, < b, 3 >\}$$

where $\emptyset$ is the empty set. ∎

A *binary relation* R on S is a subset of $S \times S$. $< x, y > \in R$ is denoted as xRy and $< x, y > \notin R$ is denoted as $x \not R y$. R is *reflexive* iff xRx for every $x \in S$. R is *transitive* iff for every $x, y, z \in S$, xRy and yRz imply xRz. R is *anti-symmetric* if, for every $x, y \in S$, xRy and yRx implies $x = y$. R is *symmetric* if, for every $x, y \in S$, $xRy \leftrightarrow yRx$. Note that a relation R may be neither symmetric nor anti-symmetric.

An *equivalence relation* R on S is a reflexive, symmetric and transitive relation on S. Let $x \in S$. $[x]_R \stackrel{def}{=} \{y \in S \mid xRy\}$. $[x]_R$ is called an *equivalence class* of R. The set of equivalence classes of R is denoted as $S_{/R} \stackrel{def}{=} \{[x]_R \mid x \in S\}$. We have $S = \bigcup S_{/R}$ and, for any $x, y \in S$, $xRy \leftrightarrow [x]_R = [y]_R$, i.e., a member of one equivalence class is related, by R, to every member of the same equivalence class but not to any member of any other equivalence class.

A *partial order* $\sqsubseteq$ on S is a reflexive, anti-symmetric and transitive relation on S. A *poset* $< S, \sqsubseteq >$ is a tuple where S is a set and $\sqsubseteq$ is a partial order on S.

Example 2.1.2 Let N be the set of natural numbers, $\leq$ be "less than or equal to" and $<$ be "less than". Then $\leq$ is a partial order on N and hence $< N, \leq >$ is a poset. $<$ is not a partial order on N because it is not reflexive. ∎

A partial order $< S, \sqsubseteq >$ can be illustrated by a Hasse diagram [109] when S is finite. A member in S corresponds to a node. If $a \sqsubseteq b$ then a is at a lower position than b. $a \sqsubseteq b$

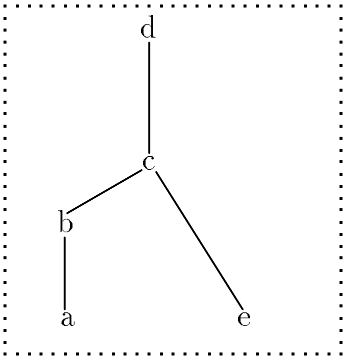


Figure 2.1: Hasse Diagram: An Example

corresponds to an edge between a and b iff $a \sqsubseteq b$ cannot be deduced by using the transitivity and the reflexivity of $\sqsubseteq$.

Example 2.1.3 Let $S = \{a, b, c, d, e\}$ and

$$\sqsubseteq \; = \left\{ \begin{array}{c} <a,a>, <a,b>, <a,c>, <a,d>, \; <b,b>, <b,c>, <b,d>, \\ <c,c>, <c,d>, \; <d,d>, \; <e,c>, <e,d>, <e,e> \end{array} \right\}$$

Then figure 2.1 is the Hasse diagram for $< S, \sqsubseteq >$.

$a \sqsubseteq b$ can be deduced from the above diagram since a and b are linked and a is at a lower place than b. $a \sqsubseteq a$ and $a \sqsubseteq d$ can be deduced by using the reflexivity and the transitivity of $\sqsubseteq$ respectively. ∎

Let $< S, \sqsubseteq >$ be a poset, $X \subseteq S$ and $u, v \in S$. u is an *upper bound* of X if $x \sqsubseteq u$ for every $x \in X$. An upper bound u of X is the *least upper bound* of X if $u \sqsubseteq v$ for every other upper bound v of X. The least upper bound of X is unique if it exists and is denoted as $\sqcup X$. $\sqcup\{x_1, x_2, \ldots, x_k\}$ is sometimes written as $x_1 \sqcup x_2 \sqcup \cdots \sqcup x_k$. $\sqcup\{x|P(x)\}$ is sometimes written as $\sqcup_{P(x)} x$. Similarly, u is a *lower bound* of X if $u \sqsubseteq x$ for every $x \in X$. A lower bound u of X is the *greatest lower bound* of X if $v \sqsubseteq u$ for every other lower bound v of X. The greatest lower bound of X is unique if it exists and is denoted as $\sqcap X$. $\sqcap\{x_1, x_2, \ldots, x_k\}$ is sometimes written as $x_1 \sqcap x_2 \sqcap \cdots \sqcap x_k$. $\sqcap\{x|P(x)\}$ is sometimes written as $\sqcap_{P(x)} x$.

Let $< S, \sqsubseteq >$ be a poset. $\perp \in S$ is an *infimum* of $< S, \sqsubseteq >$ if $\perp \sqsubseteq x$ for every $x \in S$. Not every poset has an infimum. A poset has a unique infimum when it has one. A *supremum* of $< S, \sqsubseteq >$ is defined dually.

A *complete lattice* $< S, \sqsubseteq>$ is a poset such that every $X \subseteq S$ has a least upper bound and a greatest lower bound. A complete lattice has a unique infimum and a unique supremum. We will write a complete lattice $< S, \sqsubseteq>$ as $< S, \sqsubseteq, \bot, \top, \sqcap, \sqcup >$ when it is necessary to make explicit the infimum $\bot$, the supremum $\top$, the greatest lower bound operator $\sqcap$ and the least upper bound operator $\sqcup$.

Example 2.1.4 Let $S = \{a, b, c, d, e\}$ and R be the partial order in example 2.1.3. $< S, R >$ is not a complete lattice since it does not have an infimum. ∎

Definition 2.1.5 Let $< S_1, \sqsubseteq_1>$ and $< S_2, \sqsubseteq_2>$ be posets. Define

$$< S_1, \sqsubseteq_1> \otimes < S_2, \sqsubseteq_2> \stackrel{def}{=} < S, \sqsubseteq>$$

where $S = S_1 \times S_2$ and $\sqsubseteq$ is defined as follows. For any $x_1, y_1 \in S_1$ and $x_2, y_2 \in S_2$,

$$< x_1, x_2 > \sqsubseteq < y_1, y_2 > \stackrel{def}{=} x_1 \sqsubseteq_1 y_1 \wedge x_2 \sqsubseteq_2 y_2$$

$\sqsubseteq$ compares one component independently of the other and is therefore said to be *componentwise*. This generalises to an arbitrary finite number of posets $\{< S_\imath, \sqsubseteq_\imath> \mid \imath \in I\}$ where I is a finite set of distinct indices and we use the following notation.

$$\bigotimes_{\imath \in I} < S_\imath, \sqsubseteq_\imath> \stackrel{def}{=} < S_{i_1}, \sqsubseteq_{i_1}> \otimes < S_{i_2}, \sqsubseteq_{i_2}> \otimes \cdots$$

∎

The construction of a complete lattice using $\otimes$ is illustrated by figure 2.2.

Lemma 2.1.6 Let $\{< S_\imath, \sqsubseteq_\imath, \bot_\imath, \top_\imath, \sqcap_\imath, \sqcup_\imath > \mid \imath \in I\}$ be complete lattices. Then $< S, \sqsubseteq, \bot, \top, \sqcap, \sqcup >= \bigotimes_{\imath \in I} < S_\imath, \sqsubseteq_\imath, \bot_\imath, \top_\imath, \sqcap_\imath, \sqcup_\imath >$ is a complete lattice with, letting $x_{i_k}, y_{i_k} \in D_{i_k}$ for any $i_k \in I$,

$$\begin{array}{rcl}
\bot & = & < \bot_{i_1}, \bot_{i_2}, \cdots > \\
\top & = & < \top_{i_1}, \top_{i_2}, \cdots > \\
< x_{i_1}, x_{i_2}, \cdots > \sqcap < y_{i_1}, y_{i_2}, \cdots > & = & < x_{i_1} \sqcap_{i_1} y_{i_1}, x_{i_2} \sqcap_{i_2} y_{i_2}, \cdots > \\
< x_{i_1}, x_{i_2}, \cdots > \sqcup < y_{i_1}, y_{i_2}, \cdots > & = & < x_{i_1} \sqcup_{i_1} y_{i_1}, x_{i_2} \sqcup_{i_2} y_{i_2} \cdots, >
\end{array}$$

PROOF: See p. 115 in [116]. ∎

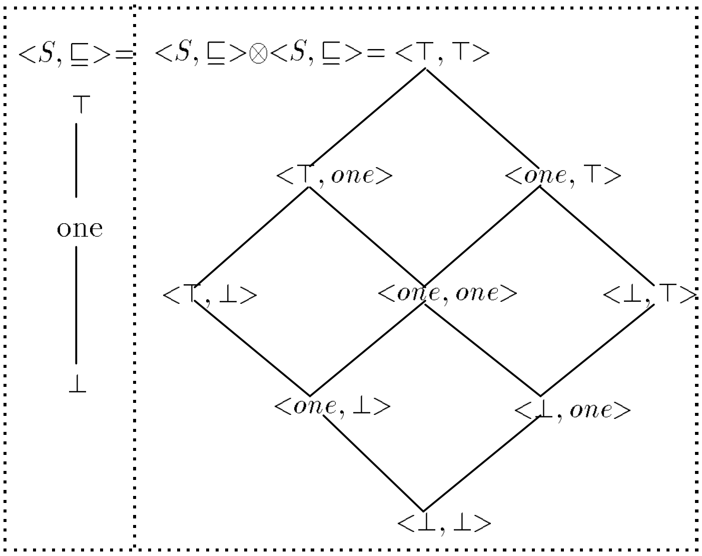


Figure 2.2: Complete Lattices: Component-wise Composition

Let D and $\bar{D}$ be sets. A *total function* f from D to $\bar{D}$ is a subset of $D \times \bar{D}$ such that, for every $d \in D$, there is one and only one $\bar{d} \in \bar{D}$ such that $< d, \bar{d} > \in f$. $< d, \bar{d} > \in f$ is denoted as $\bar{d} = f(d)$. If $f(x) = E(x)$ for any $x \in D$ then f is sometimes written as $\lambda x \in D.E(x)$ when it is not necessary to give it a name. The set of all the total functions from D to $\bar{D}$ is denoted as $D \mapsto \bar{D}$.

Let $f \in D \mapsto \bar{D}$ and $g \in \bar{D} \mapsto \tilde{D}$. We use $g \cdot f$ to denote the *composition of two functions* f and g. $g \cdot f \stackrel{def}{=} \lambda x \in D.g(f(x))$.

Definition 2.1.7 Let D be a set and $< \bar{D}, \bar{\sqsubseteq} >$ be posets. Define

$$f \mathrel{\tilde{\sqsubseteq}} g \stackrel{def}{=} \forall x \in D.(f(x) \mathrel{\bar{\sqsubseteq}} g(x))$$

$\tilde{\sqsubseteq}$ compares two functions f and g by comparing $f(x)$ and $g(x)$ according to $\bar{\sqsubseteq}$ for every point $x \in D$. So, $\tilde{\sqsubseteq}$ is said to be *point-wise*. ∎

Lemma 2.1.8 Let D be a set, $< \bar{D}, \bar{\sqsubseteq}, \bar{\bot}, \bar{\top}, \bar{\sqcap}, \bar{\sqcup} >$ a complete lattice and $\tilde{\sqsubseteq}$ be defined in the definition 2.1.7. Then $< D \mapsto \bar{D}, \tilde{\sqsubseteq} >$ is a complete lattice with its infimum $\tilde{\bot}$, its supremum $\tilde{\top}$, its greatest lower bound operator $\tilde{\sqcap}$ and its least upper bound operator $\tilde{\sqcup}$ being the following.

$$\tilde{\bot} = \lambda x \in D.\bar{\bot}$$

$$\tilde{\top} = \lambda x \in D.\bar{\top}$$

$$
\begin{aligned}
f \,\tilde{\sqcap}\, g &= \lambda x \in D.(f(x) \,\bar{\sqcap}\, g(x)) \\
f \,\tilde{\sqcup}\, g &= \lambda x \in D.(f(x) \,\bar{\sqcup}\, g(x))
\end{aligned}
$$

PROOF: See p. 118 in [116]. ∎

Let $< D, \sqsubseteq, \perp, \top, \sqcap, \sqcup >$ and $< \bar{D}, \bar{\sqsubseteq}, \bar{\perp}, \bar{\top}, \bar{\sqcap}, \bar{\sqcup} >$ be complete lattices, and $f \in \bar{D} \mapsto D$. f is *monotonic* if $f(\bar{x}) \sqsubseteq f(\bar{y})$ for any $\bar{x}, \bar{y} \in \bar{D}$ such that $\bar{x} \,\bar{\sqsubseteq}\, \bar{y}$.

Let $< D, \sqsubseteq>$ be a complete lattice and $f \in D \mapsto D$. $x \in D$ is a *fixed-point* of f if $x = f(x)$. x is the *least fixed-point*, denoted by *lfp* f, of f if $x \sqsubseteq y$ for each fixed-point y of f. *lfp* $f = f \uparrow \beta$ for some ordinal β [129] where

$$
f \uparrow \beta \stackrel{def}{=} \begin{cases} \sqcup\{f \uparrow \beta' \mid \beta' < \beta\} & if\ \beta\ is\ a\ limit\ ordinal \\ f(f \uparrow (\beta - 1)) & if\ \beta\ is\ a\ successor\ ordinal \end{cases}
$$

2.1.2 Abstract Interpretation

We now formalise the notion of abstract interpretation according to the ideas given by [19]. The idea of having an element in an abstract domain $< \bar{D}, \bar{\sqsubseteq} >$ as a description of an element in a concrete domain $< D, \sqsubseteq>$ is formalised by a monotonic function from $\bar{D}$ to D, called a *concretisation function* γ.

We say an element d in D is *approximated* by an element $\bar{d}$ in $\bar{D}$ if $d \sqsubseteq \gamma(\bar{d})$. There might well be a number of elements that approximate d. Some are more precise than others with respect to $\bar{\sqsubseteq}$.

If $\gamma \in \bar{D} \mapsto D$ and $\gamma' \in \tilde{D} \mapsto \bar{D}$ are concretisation functions then $\gamma \cdot \gamma' \in \tilde{D} \mapsto D$ is a concretisation function.

The notion of abstract interpretation can also be formalised by using an abstraction function from the concrete domain to the abstract domain, or a Galois connection between the abstract domain and the concrete domain or a relation between the concrete and the abstract domains [85].

A fixed-point interpretation of a program is the least fixed-point of a function associated with the program on a semantic domain, often a complete lattice. The following theorem shows how the least fixed-point of a monotonic function on one complete lattice can be approximated by the least fixed-point of another monotonic function on another complete lattice.

Theorem 2.1.9 If $< D, \sqsubseteq >$ and $< \bar{D}, \bar{\sqsubseteq} >$ are complete lattices, F a monotonic function on $< D, \sqsubseteq >$, $\bar{F}$ a monotonic function on $< \bar{D}, \bar{\sqsubseteq} >$, γ is a monotonic function from $\bar{D}$ to D and $\forall \bar{d} \in \bar{D}.(F \cdot \gamma(\bar{d}) \sqsubseteq \gamma \cdot \bar{F}(\bar{d}))$ then $lfp F \sqsubseteq \gamma(lfp \bar{F})$.

PROOF: See [87]. ∎

2.2 Logic Programming

We assume that the reader is familiar with the terminology used in logic programming [74]. Let $\mathcal{L}$ be a first order language with function symbol set Σ and predicate symbol set Π which is disjoint from Σ. Let $\mathcal{VAR}$ be a denumerable set of variables. $\mathcal{TERMS}$ denotes the set of *terms* that can be built from Σ and $\mathcal{VAR}$. $\mathcal{ATOMS}$ denotes the set of *atoms* constructible from Π, Σ and $\mathcal{VAR}$. The negation of an atom A is denoted as $\neg A$. A *literal* is either an atom or the negation of an atom.

Let t be a term or an atom. Define $h(t)$ to be the *height* of the tree representing t as follows

$$h(t) \stackrel{def}{=} \begin{cases} 1 + \max\{h(t_i) \mid 1 \leq i \leq n\} & if\ t = f(t_1, \cdots, t_n) \\ 0 & otherwise \end{cases}$$

Let θ and σ be *substitutions*. $\sigma \circ \theta$ denotes the composition of σ and θ. A substitution θ is called *idempotent* if $\theta \circ \theta = \theta$. $dom(\theta)$ denotes the *domain* of θ. Define $Sub \stackrel{def}{=} \{\theta \mid \theta\ is\ a\ substitution\}$. An *expression* is a term, an atom, a literal and so on. The set of variables in an expression E is denoted as $vars(E)$. For an expression E and a substitution θ, $E\theta$ denotes the *instance* of E under θ. An expression E' is an instance of another expression E if $E' \equiv E\theta$ for some substitution θ where $A \equiv B$ denotes that A is syntactically identical to B. $E\sigma\theta$ is defined as $(E\sigma)\theta$. Let θ be a substitution and $\mathcal{V} \subseteq \mathcal{VAR}$. $\theta \upharpoonright \mathcal{V} \stackrel{def}{=} \{X/t \in \theta \mid X \in \mathcal{V}\}$ is the *restriction* of θ to $\mathcal{V}$. The convention is that $\circ$ binds stronger than $\upharpoonright$. For instance, $\eta \circ \sigma \upharpoonright V$ is equal to $(\eta \circ \sigma) \upharpoonright V$.

Two substitutions σ and θ are *equivalent modulo renaming* if there are two renaming substitutions δ and ρ such that $\sigma = \theta \circ \delta$ and $\theta = \sigma \circ \rho$. We write $\sigma \cong \theta$ to denote that σ and θ are equivalent modulo renaming. We will not distinguish those substitutions that are equivalent modulo renaming. $\cong$ is naturally extended to expressions. Let E_1 and E_2 be two expressions. $E_1 \cong E_2$ if there are two renaming substitutions δ and ρ such that $E_1 = E_2\delta$ and $E_2 = E_1\rho$.

An *equation* is a formula of the form $l = r$ where l and r are terms or atoms. The set of all equations is denoted as $\mathcal{EQ}$. Let $E \in \wp(\mathcal{EQ})$. E is in *solved form* if (a) l is a variable for each $(l = r) \in E$, (b) $(l = r) \in E \wedge (l' = r') \in E \rightarrow l \neq l'$ and (c) a leftside variable does not occur in the rightside of any equation in E. There is a bijection between an idempotent substitution and a set of equations in solved form. The set of equations in solved form corresponding to an idempotent substitution θ is denoted as $eq(\theta)$.

A substitution σ is a *unifier* for a set of equations $E \in \wp(\mathcal{EQ})$ if $l\sigma \equiv r\sigma$ for every equation $l = r$ in E. If a set of equations E has a unifier then E is said to be *unifiable*. A unifier σ for a set of equations E is a *most general unifier* for E if, for any other unifier θ for E, there is a substitution η such that $\sigma = \theta \circ \eta$. The unification of a set of equations is decidable and the most general unifiers for a set of equations are equivalent modulo renaming. Therefore, there is a function mgu from $\wp(\mathcal{EQ})$ to $\{fail\} \cup Sub$ that, given a set of equations E, either returns a most general unifier for E if E is unifiable or returns *fail* otherwise. We define $eq(fail) \stackrel{def}{=} fail$. We sometimes write $mgu(\{l = r\})$ as $mgu(l, r)$ when the set of equations to be unified is a singleton.

A *normal clause* is a formula of the form $H \leftarrow L_1, \cdots, L_n$ where H is an atom, L_i for each $1 \leq i \leq n$ is a literal. H is called the *head* of the clause and $L_1, \cdots, L_n$ the *body* of the clause. If $n = 0$ then $H \leftarrow L_1, \cdots, L_n$ is called a fact and written as $H \leftarrow$. A *normal goal* is a formula of the form $\leftarrow L_1, \cdots, L_n$ with L_i for each $1 \leq i \leq n$ being a literal. A *normal (logic) program* $\mathcal{P}$ is a set $\{C_\imath \mid \imath \in \aleph_C\}$ of normal clauses where $\aleph_C$ is a finite set of distinct natural numbers. Let $m[\imath]$ denote the number of literals in the body of clause $C_\imath$. We write $C_\imath$ as $H_\imath \leftarrow L_{(\imath,1)}, L_{(\imath,2)}, \cdots, L_{(\imath,m[\imath])}$.

There might be an infinite number of possible initial goals on which a program is to be executed. For the time being, the set of possible initial goals is denoted as $\mathcal{G} \stackrel{def}{=} \{< G_k, \Theta_k > | k \in \aleph_G\}$ where $\aleph_G$ is a finite set of distinct natural numbers such that $\aleph_G \cap \aleph_C = \emptyset$. G_k for each $k \in \aleph_G$ is a normal goal and Θ_k is a set of substitutions θ_k such that $dom(\theta_k) \subseteq vars(G_k)$. Each $G_k\theta_k$ with $\theta_k \in \Theta_k$ is an initial goal. Let $m[k]$ be the number of literals in G_k. We write G_k as $\leftarrow L_{(k,1)}, L_{(k,2)}, \cdots, L_{(k,m[k])}$.

Let $\aleph \stackrel{def}{=} \aleph_C \cup \aleph_G$. Let $\mathcal{P}_i$ refer to C_i for $i \in \aleph_C$ and to G_i for $i \in \aleph_G$. Define $\mathcal{V}_i \stackrel{def}{=} vars(\mathcal{P}_i)$.

Let $i \in \aleph$. We annotate $\mathcal{P}_i$ with $m[i] + 1$ *program points*, point (i, j) to the left of $L_{(i,j)}$ for $1 \leq j \leq m[i]$ and point $(i, m[i] + 1)$ to the right of $L_{(i,m[i])}$. $entry(i) \stackrel{def}{=} (i, 1)$ is called the *entry point* of $\mathcal{P}_i$ and $exit(i) \stackrel{def}{=} (i, m[i] + 1)$ the *exit point* of $\mathcal{P}_i$.

We denote by $\mathcal{N}_{\mathcal{P},\mathcal{G}}$ the set of all the program points annotated with $\mathcal{P}_i$ for all $i \in \aleph$. Let $p = (i, j)$ be a program point. $p[1] = i$ denotes the index to the clause or the initial goal to which p belongs. $p[2] = j$ denotes the position of p in the clause or the initial goal. So, $p = (p[1], p[2])$. We define two partial functions $\lambda p.p^+$ and $\lambda p.p^-$ over the set of all the program points.

$p^+ = (p[1], p[2] + 1)$ is defined for each p such that $p[2] \leq m[p[1]]$ and $p^- = (p[1], p[2] - 1)$ is defined for each p satisfying $2 \leq p[2] \leq m[p[1]] + 1$. p^+ is the program point to the right of p if p^+ exists and p^- is the program point to the left of p if p^- exists.

Let $p \in \mathcal{N}_{\mathcal{P},\mathcal{G}}$. B_p denotes the atom in literal L_p. If L_p is positive then $L_p \equiv B_p$. If L_p is negative then $L_p \equiv \neg B_p$.

Example 2.2.1 The following normal logic program will be used in several examples in the thesis. The intended interpretation for $member(X, L)$ is that X is a member of list L. The intended interpretation for $diff(X, L, K)$ is that either X is a member of list L or X is a member of list K but that X is not both a member of L and a member of K.

$$\begin{aligned}
C_1 &\equiv diff(X, L, K) \leftarrow member(X, L), \neg member(X, K) \\
C_2 &\equiv diff(X, L, K) \leftarrow member(X, K), \neg member(X, L) \\
C_3 &\equiv member(X, [X|L]) \leftarrow \\
C_4 &\equiv member(X, [H|L]) \leftarrow member(X, L)
\end{aligned}$$

Suppose that the set of initial goals is described by $\{< G_5, \Theta_5 >\}$ with $G_5 \equiv \leftarrow diff(X, Y, Z)$ and Θ_5 is the set of substitutions θ such that $X\theta$ is a variable, and both $L\theta$ and $K\theta$ are ground terms. Then, $\aleph_C = \{1, 2, 3, 4\}$ and $\aleph_G = \{5\}$. $\mathcal{N}_{\mathcal{P},\mathcal{G}}$ contains 11 program points. $L_{(1,1)} = member(X, L)$ and $L_{(1,2)} = \neg member(X, K)$. $\mathcal{V}_1 = \{X, L, K\}$ and $\mathcal{V}_5 = \{X, Y, Z\}$. ∎

2.3 Program Debugging

An error that makes a program exhibit some unexpected behaviour is called a bug while the unexpected behaviour that a bug causes is called a bug symptom. A buggy logic program may exhibit many kinds of bug symptom. It may produce a wrong answer, fail to produce a correct answer, fall into looping, call procedures with arguments of wrong types, or violate the weaker safeness condition for the negation as failure rule, etc. The first two kinds of bug symptom are called declarative since they are related to the declarative semantics of logic programs whilst others are called operational since they are related to the operational semantics of logic program.

First consider declarative symptoms. Let $\mathcal{P}$ be a logic program. We define the success set of $\mathcal{P}$ as

$$SS(\mathcal{P}) \stackrel{def}{=} \{A \mid A \text{ is an atom and } \epsilon \text{ is a computed answer for } P \cup \{\leftarrow A\}\}$$

$SS(\mathcal{P})$ is the set of the theorems that are provable from the axioms in $\mathcal{P}$. Let $A \in SS(\mathcal{P})$; then $\forall A$ is provable from $\mathcal{P}$. Let $vars(A) = \{X_1, \cdots, X_k\}$. $\forall A$ denotes $\forall X_1. \cdots \forall X_k.A$. $SS(\mathcal{P})$ is closed under instantiation; that is, if $A \in SS(\mathcal{P})$ then $A\theta \in SS(\mathcal{P})$ for any substitution θ. The success set $SS(\mathcal{P})$ possibly contains non-ground atoms and describes the operational behaviour of logic programs more precisely than the success set defined in terms of ground atoms [132]. A model-theoretic counterpart of $SS(\mathcal{P})$ for definite programs is given in [42].

Definition 2.3.1 Let $\mathcal{P}$ be a logic program and $\mathcal{M}$ an interpretation.

(1) An atom A is an inconsistency symptom of $\mathcal{P}$ w.r.t. (with respect to) $\mathcal{M}$ if A is invalid in $\mathcal{M}$ and $A \in SS(\mathcal{P})$.

(2) An atom A is an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ if A is satisfiable in $\mathcal{M}$ and $\mathcal{P}$ finitely fails on A. ∎

The definition of an inconsistency symptom A of $\mathcal{P}$ w.r.t. $\mathcal{M}$ can be strengthened by requiring A to be unsatisfiable in $\mathcal{M}$ and that of an insufficiency symptom A of $\mathcal{P}$ w.r.t. $\mathcal{M}$ can be strengthened by requiring A to be valid in $\mathcal{M}$. By a declarative symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$, we refer to an inconsistency or insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

Definition 2.3.2 Let $\mathcal{P}$ be a logic program and $\mathcal{M}$ an interpretation.

(1) A clause instance $A \leftarrow W$ is an inconsistent clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$ if $A \leftarrow W$ is an instance of a clause of $\mathcal{P}$ and $A \leftarrow W$ is invalid in $\mathcal{M}$.

(2) An atom A is an uncovered atom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ if A is valid in $\mathcal{M}$ and, for every clause $A' \leftarrow W$ of $\mathcal{P}$ such that A and A' unify with mgu θ, $W\theta$ is unsatisfiable in $\mathcal{M}$. An atom A is an incompletely covered atom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ if an instance of A is an uncovered atom of $\mathcal{P}$ w.r.t. $\mathcal{M}$. ∎

In the literature, the notion of $A \leftarrow W$ being an inconsistent clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$ is often strengthened by requiring $A \leftarrow W$ to be unsatisfiable in $\mathcal{M}$. By a declarative bug of $\mathcal{P}$ w.r.t. $\mathcal{M}$, we refer to an inconsistent clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$ or an uncovered atom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ or an incompletely covered atom of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

If there is a declarative symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ then there is a declarative bug of $\mathcal{P}$ w.r.t. $\mathcal{M}$ [43, 44, 73, 74, 144]. Therefore, given a declarative symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$, a declarative diagnoser for logic programs searches for a declarative bug of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

Definition 2.3.3 Let $\mathcal{P}$ be a normal program and A be an atom. A is said to be a non-termination symptom of $\mathcal{P}$ if an execution of $\mathcal{P}$ on A never stops. ∎

If A is a non-termination symptom of $\mathcal{P}$, there is an SLDNF-derivation of $\mathcal{P} \cup \{\leftarrow A\}$ such that either SLDNF recurs on itself indefinitely or there is an atom A' such that $\neg A'$ is selected within a recursive invocation of SLDNF and there is an infinite SLDNF-derivation of $\mathcal{P} \cup \{\leftarrow A'\}$.

Definition 2.3.4 Let $\mathcal{P}$ be a normal program and A be an atom. A is said to be an unsafe symptom of $\mathcal{P}$ if an execution of $\mathcal{P}$ on A violates the weaker safeness condition for the negation as failure rule. ∎

We characterise a class of operational symptoms to which both mode and type violations belong by attaching to each program point a set of admissible substitutions. The oracle is required to decide whether a substitution is admissible at a program point. Let p be a program point and σ a substitution. We say that $< p, \sigma >$ is admissible if σ is admissible at p. We say that $< p, \sigma >$ is inadmissible if σ is inadmissible at p.

Definition 2.3.5 Let $\mathcal{P}$ be a program and A be an atom. A is an inadmissibility symptom of $\mathcal{P}$ if the execution of $\mathcal{P}$ on A reaches a program point p with a current substitution θ such that $< p, \sigma >$ is inadmissible. ∎

The detection of inadmissibility symptoms requires assertions to be attached to program points. Let p be a program point and $\mathcal{A}(X_1, \cdots, X_k)$ an assertion where $X_1, \cdots, X_k$ are variables occurring in the clause or in the initial goal to which p belongs. The attachment of $\mathcal{A}(X_1, \cdots, X_k)$ to p dictates that, whenever the execution of the program reaches p with the value of X_j being t_j for $1 \leq j \leq k$, $\mathcal{A}(t_1, \cdots, t_k)$ must be true. A mode assertion dictates the instantiation modes of the values of the variables $X_1, \cdots, X_k$ whilst a type assertion dictates the types of the values of the variables $X_1, \cdots, X_k$. The substitutions satisfying $\mathcal{A}(X_1, \cdots, X_k)$ should be a superset of those substitutions which are admissible at the program point p. The association of an assertion $\mathcal{A}$ with a program point p is denoted as $\mathcal{A}@p$.

Definition 2.3.6 Let $\mathcal{P}$ be a normal program, $\mathcal{M}$ an interpretation, p and q two program points and σ and θ two substitutions. $(< p, \sigma >, < q, \theta >)$ is said to be a diverging invocation of $\mathcal{P}$ w.r.t. $\mathcal{M}$ if (a) $< p, \sigma >$ is admissible, and (b) if $C_{q[1]}$ is used to satisfy $L_p\sigma$ then execution reaches q with θ being the current substitution, $L_{(q[1],1)}\theta, \cdots, L_{q^-}\theta$ are valid in $\mathcal{M}$, and $< q, \theta >$ is inadmissible. ∎

It is well known that the problem of non-termination is undecidable [122]. The programmer could know that a program did not terminate only if they had waited for an infinite amount of time. This does not prevent the programmer from debugging such a program since the execution of a non-terminating program will eventually exhaust the space or the time or some other resource allocated to running it. However, even if the execution exhausts one of its allocated resources, it still does not mean that the program is non-terminating. It may be that the program is not sufficiently efficient in using the resource or that there is not enough of the resource. We detect non-termination symptoms by setting a limit on the number of nested procedural calls. An atom A is suspected to be a non-termination symptom of $\mathcal{P}$ if the execution of $\mathcal{P}$ on A exceeds the limit. The diagnoser for non-termination may fail to find a bug since A is not necessarily a non-termination symptom of $\mathcal{P}$.

Let $\mathcal{P}$ be a normal program and $\mathcal{M}$ be an interpretation. We shall show in Chapter 7 that if there is an inadmissibility symptom of $\mathcal{P}$ then there is a diverging invocation of $\mathcal{P}$ and $\mathcal{M}$ or a declarative bug of $\mathcal{P}$ w.r.t. $\mathcal{M}$. The first case is related to the operational semantics and the second to the declarative semantics.

Chapter 3

Related Work

This chapter surveys the main approaches to bug detection and bug diagnosis. The main approaches used in existing systems are first summarised, exemplified through typical systems and then compared to justify the choice of the particular approaches. We also survey related work on abstract interpretation, mode and type analyses in logic programs.

3.1 Detection Approaches

This section reviews the main bug detection approaches that have been implemented in existing debugging systems. It covers debugging systems that support other programming paradigms as well as the logic programming paradigm because there been comparatively little research devoted to bug detection in logic programs.

3.1.1 Program Understanding Approach

The program understanding approach [1, 58] creates a problem decomposition for the specification that best matches the program and interprets the mismatches between them as bugs.

The specification is given at a different level of formalisation in different systems. In PROUST (PROgram Understander for STudents [58]), the specification is informally given as a list of goals that the program must achieve and the objects that the program must manipulate. In MYCROFT [48], the specification is formally given as a set of axioms that the program must satisfy. In IPA (Intelligent Program Analysis [113]), the specification is available as a program generation model from which correct programs implementing the specification can be derived.

These systems use knowledge of programming and knowledge of language constructs to create a problem decomposition of the specification that best matches the program.

Program understanding can be problem-driven or code-driven. Problem-driven understanding starts with the specification, forms a hypothesis of the overall structure of the program and refines the hypothesis to the level of code at which the hypothesised structure can be verified against the program. Code-driven understanding starts from the program and forms a more abstract description of what the individual parts of the program do and then what the entire program does.

PROUST, an intelligent tutoring system for novice PASCAL programmers, exemplifies the program understanding approach. Given a specification consisting of a list of goals that the program must achieve and a list of objects that the program must manipulate, PROUST hypotheseses possible goal decompositions that relate the goals in the specification to the plans that achieve them and matches these goal decompositions with the program code in order to find a goal decomposition that best matches the program. The mismatches between the best matching decomposition and the program code are interpreted as bugs.

PROUST uses its knowledge of programming plans in order to understand the program. A programming plan is a procedure or strategy for realising intentions in code where the key elements have be abstracted and represented [59]. For example, the following "Sentinel Process-Read Plan" is in the knowledge base of PROUST.

```
CONSTANT: ?Stop
VARIABLE: ?New
Template:
    initinput:
        subgoal Input(?New)
    mainloop:
        WHILE ?New <> ?Stop DO
        BEGIN
            Process: ?#
            Next:
                subgoal Input(?New)
        END
```

To find the best matching goal decomposition, PROUST searches the space of program interpretations – the interpretations of program meaning. Each state in the space has three components. One component is a list of the goals whose implementations within the program have yet to be determined. Another component is a partial goal decomposition. The other component is a match between the plans of the partial decomposition and the program code. The initial state consists of the list of goals in the specification, an empty goal decomposition and an empty match between the goal decomposition and the program.

At any state PROUST selects one goal from the goal list and then selects a plan from a list of plans that implement this goal. The structure of the plan is matched with the program code and subgoals in the plan are added to the goal list. State transition occurs when PROUST either selects a goal or selects a plan or matches the plan template with the code or transforming a goal.

3.1.2 Program Verification Approach

The program verification approach [80, 92, 93] tries to prove that the program satisfies the specification. Failure in doing so indicates the possible existence of bugs.

The specification in existing systems is either given as example programs [92, 93] that implement the specification or given as a pair of preconditions and postconditions for each subprogram [80]. These systems use knowledge of program constructs to prove that the program satisfies the specification. To deal with the diversity in implementing the specification, the systems that use example programs as the specification usually require that a number of example programs are given and use program transformation to deal with minor variations [92, 93]. These program transformations are small semantics-preserving transformations. For instance, the expression $X \leq Y$ may be transformed to the expression $Y \geq X$ in order to make it easier to compare the program with the specification.

TALUS [92, 93] exemplifies the program verification approach. It is intended for debugging student programs in Lisp. The specification is given as several stored programs implementing different algorithms for the problem task. TALUS first uses heuristic knowledge to identify the algorithm that best matches the student program and maps the functions and the variables in the student program to those in the corresponding stored program. The result is that each student function is paired with a stored function. Then the functions in each pair are checked for their computational equivalence. TALUS first uses symbolic execution to identify all the conditions that must hold in order for the two functions to be equivalent. Then a Boyer-Moore theorem-prover [7] is used to prove these conditions. Symbolic execution used in TALUS is precise. For instance, the symbolic execution of the expression $X + X + 2 * X$ gives the expression $4 * X$ as output. In the debugging scenario given in [92], TALUS has to establish the equivalence between the following student function

```
(DEFUN FLAT (ANS TR)
   (IF (ATOM TR) ANS
       (FLAT (FLAT ANS (CDR TR)) (CAR TR))))
```

and the following stored function

```
(DEFUN FLAT (ANS TR)
```

```
(IF (ATOM TR) (CONS TR ANS)
    (FLAT (FLAT ANS (CDR TR)) (CAR TR))))
```

Using its knowledge of program constructs, TALUS determines that the following two conditions must be true in order for the two functions to be computationally equivalent:

```
(IMPLIES (ATOM TR) (EQUAL (CONS TR ANS) ANS))
```

and

```
(IMPLIES (NOT (ATOM TR))
         (EQUAL (FLAT (FLAT ANS (CDR TR)) (CAR TR))
                (FLAT (FLAT ANS (CDR TR)) (CAR TR))))
```

The Boyer-Moore theorem prover is used to prove the above two conditions. The theorem prover succeeds in proving the second condition and fails in proving the first one. The failure in proving the first condition detects a bug in the student function.

3.1.3 Well-Formedness Checking Approach

The well-formedness checking approach [127, 141] does not use a specification; rather it applies a standard to all programs, in the sense that all programs are checked against the same specification. The program is considered buggy if it violates a syntactic rule, a rule of programming discourse [123] or a pragmatic rule [141].

PHENARETE [141] exemplifies the well-formedness checking approach. When given the following version of the REVERSE function in Lisp

```
(DEFUN REV L1 L2 COND ULL L22 A1 T RVE A1 ONS CRA A1 A2)
```

PHENARETE first uses its knowledge of the construction and correction of common program constructs to correct the program to the following syntactically correct program.

```
(DEFUN REV (L1 L2)
    (COND ((NULL L2) L1)
          (T (REV L1 (CONS (CAR L1) L2)))))
```

PHENARETE then uses its knowledge of standard Lisp functions to determine that the program does not terminate. The knowledge of standard Lisp functions enables PHENARETE to infer that the function accesses the elements of list L1 and so L1 should be broken by CDR and tested by NULL for terminating recursion. PHENARETE then proposes two possible improved versions of the function:

```
(DEFUN REV (L1 L2)
    (COND ((NULL L2) L1)
          ((NULL L1) L2)
           (T (REV (CDR L1) (CONS (CAR L1) L2)))))
```

and

```
(DEFUN REV (L1 L2)
    (COND ((NULL L1) L2)
          (T (REV (CDR L1) (CONS (CAR L1) L2)))))
```

Since PHENARETE does not use a specification, it cannot decide that the second function is more appropriate than the first function. This illustrates the limitations of debugging systems that do not use specifications.

3.1.4 Program Property Analysis Approach

The program property analysis approach [75, 100] works with a specification that specifies a set of properties that the program must satisfy. The program is checked against these properties. Any violation of these program properties suggests the possibility of the existence of bugs. Program property analysis systems such as NUDE (NU-Prolog Debugging Environment [100]) and Apropos (Analyzer for Prolog Programs Of Students [75]) do not actually detect bugs; they can only give warnings to the programmer who bears the responsibility for detecting bugs. Because of the inherent impreciseness of the program analysis techniques that they use, these systems can either only give warning against the possible existence of bugs or use other approaches to confirm the existence of bugs.

NUDE [100] exemplifies this approach. NUDE consists of a set of program analysis tools for debugging NU-Prolog programs. The programmer may specify the mode or the type for each predicate in the program and use the appropriate tools to check if the program is consistent with the mode or the type specification. For example, if the specification has the following type definition, which is given in a regular type language [146]:

```
?- type $odd -> {$_.[], $_.$_.$odd}
?- type $even -> {[],$_.$_.$even}
```

NUDE will detect that the clause

```
query :- odd(X), even(X).
```

has a type error, because the type unification of the types \$odd and \$even is empty.

3.1.5 Program Testing Approach

The program testing approach compares the input-output relation of the program with that of the specification. Since there are usually an infinite number of input-output pairs, the comparison is only done on a set of sample data called test data. The generation of the test data is a subject of intensive research in the imperative programming paradigm, but little has been done in the logic programming paradigm.

The Deductive Debugging System [30, 31, 70] exemplifies the testing approach. It debugs definite logic programs and works with an executable specification that is also a definite logic program. To detect bugs, the Deductive Debugging System tests each predicate of the program. It first runs the specification of each predicate to obtain a correct input-output pair, then uses only the input value to run the the program to be debugged. If the execution fails, goes into a loop or returns an incorrect output value, then the program has at least one bug.

The Deductive Debugging System generates alternative test data through backtracking. For any non-trivial program, this testing process does not terminate unless some mechanism is used to force a termination. However this would imply that the predicate may not have been tested thoroughly to ensure confidence in the correctness of the program.

BIP [5] uses sample input-output pairs as specification. The program to be debugged is checked with respect to this set of input-output pairs. If the program returns an incorrect output value for any input value in the specification, the program is deemed to have bugs. Given the fact that the input-output pairs are specified by the specification, BIP does not do much in automating bug detection except for executing the program on the sample inputs.

3.1.6 Comparison of the Detection Approaches

The program understanding approach has the advantage of detailed analysis of the program. It not only proves the correctness of the program, but also understands the problem decomposition process which the programmer went through when writing the program. Thus it can interpret the possible errors in the problem decomposition process.

The program understanding approach is subject to two fundamental limitations: prohibitive computational cost and incompleteness of the knowledge of programming plans. To understand the program, PROUST must search a very large space of program interpretations. The size of the space is exponential to the number of programming plans [134]. The number of plans that such a system would require is very large. The Programmer's Apprentice [139] effort has revealed that there are probably several thousand important program clichés – abstract versions of programming plans. The corresponding number for the more concrete programming plans is even larger. Like many other systems which rely only on their knowledge bases which are very often incomplete, program understanding systems can only understand what they know about. For example, if a plan is not in its knowledge base, then PROUST is at a loss as how to interpret the corresponding code.

The attraction that the program verification approach has is its sound mathematical and logical foundation. However, this approach requires that the specification is given in a rigid formalism. For example, PUDSY (Program Understanding and Debugging SYstem [80]) requires the specification to be given as pre-conditions and post-conditions in the first order logic while TALUS requires several implementations of the specification. The verification approach is also subject to heavy computational cost. Inductive systems such as PUDSY must search through a large base of program schemata while deductive systems like TALUS must deal with the diversity of the possible implementations of the same specification.

The well-formedness checking approach is computationally much less expensive than the program understanding approach and the program verification approach. However, because this approach does not use the specification, a system that uses this approach can only give warnings and delegate the responsibility of detecting bugs to the programmer.

Like the well-formedness checking approach, the program property analysis approach also gives warnings at a relatively low cost. The difference between the well-formedness checking approach and the program property analysis approach lies in that the well-formedness checking approach warns against form-related bugs while the program property analysis approach against semantics-related bugs.

Program testing costs less than program understanding and program verification, but more than well-formedness checking and program property analysis. The thorough testing approach exemplified by the Deductive Debugging System is computational costly and theoretically impossible [53, 91]. The sample testing exemplified by BIP does not generate test data at all.

We have so far reviewed the main bug detection approaches. Each of these approaches has its advantages and disadvantages. Ideally we would like to make use of their advantages and avoid their disadvantages; that is, we would like to automate bug detection with a high reliability without paying a very high computational cost. We detect bugs by combining the program property analysis approach and the program testing approach. The program is first checked, by means of program analysis, against a set of assertions associated with it and then tested on a set of test data should it fail to satisfy the set of assertions associated with it.

3.2 Diagnosis Approaches

This section reviews the main bug diagnosis approaches that have been used in existing debugging systems.

The studies on human debugging process have shown that the programmer extracts and analyses various information in generating and evaluating their hypotheses on bug locations [49, 134]. The debugging systems have been developed to support information extraction and analysis. The level of support for information extraction and analysis varies between the systems. The information filtering approach reduces the quantity of information that the programmer has to analyse. The declarative approach requires the programmer to provide part of the intended interpretation for the program. The knowledge-based approach reasons about the bug symptom, the program and the specification to localise bugs.

3.2.1 Information Filtering Approach

The information filtering systems extract related information from the trace of the bug symptom, that is, the execution history [36, 39, 127] of the bug symptom. The programmer then analyses the information to identify the part of the trace to examine further.

Information filtering systems vary by the level at which the trace is abstracted, ranging from tracers that provide commands for the programmer to navigate through the trace to the systems that present the top-level behaviour of the bug symptom to the programmer.

Essential to an information filtering system is an execution model of programs. Many models have been proposed for use in debugging logic programs [11, 38, 108, 111]. These models present the programmer with information at different levels of detail. For example, the box model [11] does not allow the programmer to access information about unification while the model used in the Coda system [108] makes information about unification available.

In Byrd's box model of Prolog [11], there are four ports at which the execution of a call may be inspected: Call, Exit, Redo and Fail. The Call port corresponds to the first time that the call is executed. If the execution is successful, then the Exit port is visited and later calls executed. The Redo port is visited on backtracking if the remaining calls fail. Here Prolog attempts to find alternative solution paths for the call by visiting the Redo port. If none is found, then the Fail port is visited and backtracking proceeds to earlier goals. If there are more solutions, then the Exit port is visited again.

Debugging facilities in many Prolog implementations are based on Byrd's execution model. These debuggers are often called *de facto* standard debuggers [108]. These debuggers provide the programmer execution traces of Prolog programs. The programmer can inspect the process of the execution of Prolog programs either by tracing step by step or by setting breakpoints. However, such debuggers do not provide the the debugging information to the programmer at an appropriate level of detail. On the one hand, the programmer has no access to the step of clause matching. Therefore, no information is provided about failed pattern matches between clauses and calls. It is sometimes difficult to see the relationship between the program trace and the program clauses. On the other hand, these debuggers usually overwhelm the programmer with the size of the program trace.

There have been many attempts to improve the way in which the debuggers provide information to the programmer.

Plummer [108] extends Byrd's four port execution model into an execution model which models the behaviour of a Prolog program in terms of clauses rather than calls as is the case in Byrd's model. The Coda system [108] provides more detailed information than the standard debugger by allowing the programmer to interact with the process of clause matching and displaying the program trace in terms of clauses rather than calls. Coda allows the programmer to set watch points not only on procedures but also on particular invocations.

Both Byrd's and Plummer's models provide too much information for the programmer. Most of the information presented is not relevant to the programmer's debugging purposes. Providing too much information for the programmer not only prolongs the debugging task but may also overwhelm the programmer. A good debugger should provide the programmer with relevant information while abstracting away irrelevant information.

The way in which the information is presented to the programmer is equally important in debugging. Presented with program trace in text lines, the programmer does not find it easy to relate the information to the bug. Eisenstadt and Brayshaw [40] present an AORTA (And/OR Tree, Augmented) model of Prolog execution and a graphical debugger for Prolog programming. The debugger is able to provide both a close-up view of program execution for beginners and a long-distance view for experts.

PTP (Prolog Trace Package [38, 39]) shows the clause being tried with several types of failure and presents the programmer with an overview of execution. The programmer can selectively ask for more detailed views of the execution to see what has happened in the trace and focus on the execution of a particular call. The following scenario given in [39] should help give an impression of the information filtering by PTP. PTP is invoked when the call *qsort*([2,1,3],R) fails erroneously. PTP extracts the clause used to resolve the call and all failed subcalls.

```
PTP: qsort([2,1,3],R).
  1: ? qsort([2,1,3],_275)
 38: - qsort([2,1,3],_275) {2}
 SUBGOAL FAILURES:
 7/16: split([3],2,_184,_185)
 3/36: split([1,3],2,_184,_185)
 1/39: qsort([2,1,3],_275)
```

The programmer then analyses this information and may decide to look in detail at how a particular subcall fails.

PRESET [127] generates an explanation for the bug symptom from the execution history and presents it to the programmer. The programmer identifies the part of the explanation that is suspicious and asks for further explanation.

Suppose we have the following program which contains a bug in the predicate *even*/1.

```
f1(X,Y) :- even(X), !, Y is X/2.
f1(X,X).
even(X) :- 0 is X mod 3.
```

The call f1(4,X) having the wrong answer f1(4,4), the interaction between the programmer and PRESET is as follows.

```
why f1(4,4).
Matched: f1(X,X).
Backtracking caused. Explain(y/n)? y

1st matched: f1(X,Y) :- even(X), !, Y is X/2.
even(4) fails.  Explain(y/n) y

Matched: even(X) :- 0 is X mod 3.
0 is 4 mod 3 fails.  (is: System Predicate) ok?
```

PTP [38, 39] extracts more detailed information than PRESET. It abstracts a top-level behaviour of the execution history and presents this top-level behaviour to the programmer. The programmer then identifies the suspicious part of the top-level behaviour to *zoom in*, that is, to observe that part of the behaviour in more detail. TPM (Transparent Prolog Machine [40]) enhances PTP by presenting the execution trace graphically.

Instead of fixing the level of detail at which the execution history is abstracted, Opium [36] provides the programmer with a meta-language to program their information extraction methods and their diagnosis strategies. The programmability makes it possible for the experienced programmer's diagnosis expertise to be implemented. Several diagnosis strategies have been implemented in Opium [34, 35, 41].

3.2.2 Knowledge-Based Approach

The knowledge-based approach incorporates expertise that the programmer uses in debugging. For example, if the output of a sorting procedure is in the wrong order, the programmer may hypothesise that the fault is probably in the operator of the comparison statement [120]. FALOSY (FAult LOcalization SYstem [118]) uses its debugging knowledge to make such a hypothesis. Within its knowledge base is a set of fault models that associate output discrepancies with possible causes. Each fault model lists the fault symptoms under which it is applicable, hypotheses about possible faults, and other possibly related fault models. A fault model in FALOSY's knowledge base follows.

```
(master_file_priming_read_error
   (triggers (Invalid first master_record)
             (Invalid first new master_record)
             (Invalid master_record_count))
   (expected_defects missing master_file_priming_read)
   (related_faults insert_error))
```

FALOSY [118] works with a manually prepared discrepancy list and the program source code. The following discrepancy list indicates that both the first master and new master records contain invalid data, and that the number of master records processed is not equal to the number of records known to be in the master file.

```
(discrepancy_list
    (invalid (first master_record))
    (invalid (first new_master_record))
    (ne (master_record_count (card master_file))))
```

Given the above discrepancy list, FALOSY first selects a fault-driven diagnosis tactic and triggers the *master_file_priming_read_error* fault model. So, FALOSY hypothesises that the priming read statement for the master file is missing.

3.2.3 Declarative Approach

The declarative approach utilises the trace of bug symptoms and the intended interpretation for the program in order to localise bugs. The agent responsible for providing the intended interpretation is called the oracle and is usually the programmer. The oracle only needs to know the intended interpretation for the program and does not need to know the operational semantics of the program. This relieves the oracle from the execution details.

The principle of declarative diagnosis is as follows. If a computation of a program component gives an incorrect result and all the sub-computations it invokes compute correct results, then the code of the program component is buggy.

The following buggy insertion sort program has an incorrectness symptom: it computes *isort*([2,1,3],[2,3,1]) which is not in the intended interpretation.

```
isort([X|Xs],Ys) :- isort(Xs,Zs),insert(X,Zs,Ys).
isort([],[]).

insert(X,[Y|Ys],[Y|Zs]) :- Y>X,insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys]) :- X<=Y.
insert(X,[],[X]).
```

The application of the incorrectness bug diagnosis procedure *fp*/2 in [122] results in the following.

```
|?- fp(isort([2,1,3],[2,3,1]),C).
query: isort([],[])? y.
query: insert(3,[],[3])? y.
query: isort([3],[3])? y.
query: insert(1,[],[1])? y.
query: insert(1,[3],[3,1])? n.
C = insert(1,[3],[3,1]) :- 3>1, insert(1,[],[1]).
yes
```

The oracle is queried five times and each time is asked to tell if a particular element is in the intended interpretation for the program. The diagnosis is an instance of a buggy clause in the program.

3.2.4 Comparison of the Diagnosis Approaches

An information filtering system allows the programmer to navigate the trace of the bug symptom freely. This might be the only advantage that this approach has over the other approaches.

The support provided by filtering systems is limited. Because the programmer is responsible for identifying any suspicious operational behaviour of the bug symptom, he must know the operational semantics as well as the declarative semantics of the specification. The level of detail at which the trace is abstracted is not necessarily suitable to the problem at hand. This may make it difficult to identify the suspicious behaviour.

Knowledge-based diagnosis systems not only localise the bug but also give the identification of the bug. The ability of identifying the bug stems from their knowledge bases.

A system that implements the knowledge based approach is subject to a fundamental limitation: the incompleteness of its knowledge base. The limited amount of knowledge implies a limited ability in localising bugs. Given a discrepancy which cannot be matched with any fault model, FALOSY will not be able to make any hypothesis about the possible cause of the discrepancy. Furthermore, a system that uses the knowledge based approach requires a more detailed classification of bug symptoms by the programmer. This implies more human involvement.

The declarative approach has the advantage that the oracle only needs to know the intended interpretation for the program and does not need to know the operational semantics of the program. This is important, especially for logic programming languages, for the following reasons. Firstly, when programming, the programmer needs only to specify the logic of the problem the program is supposed to solve. The control of the program is accomplished by the implementation of the logic programming language. If the programmer had to know the execution of their programs during debugging, they would not benefit from the advantages that the logic programming paradigm is supposed to provide. Secondly, some implementation optimisations, say tail recursion optimisations and folding/unfolding transformations, make the execution of the program difficult to understand. The same program produces different traces under different implementations. Thirdly, parallel programming languages make the flow of control difficult to predict. Different executions of the same code can produce different traces.

The declarative approach localises bugs to clause level, which is coarser than the knowledge based approach. For example, FALOSY pinpoints a particular statement in a function. However, the coarseness of bug diagnosis does not hinder its use in logic program debugging because the size of a clause is usually smaller than that of a function in other programming paradigms. When used in other programming paradigms, the declarative approach can be used together with other approaches. For example, the declarative approach has been used in [45] together with program slicing [140]. Program slicing as it is in imperative programming is a technique which, given a variable X at a location L in a procedure P, finds all the statements in P which are relevant to X at L. A statement is relevant to X at L if it accesses (reads or writes) a variable Y and a change in Y's value leads to a change in X's value at L or vice verse. The size of a clause in a logic program is much smaller than a procedure or a function in an imperative or functional or object-oriented program. Therefore, program slicing as a bug localisation technique is not as much helpful for logic programs as for other programs.

The major problem with the declarative approach is the large quantity of queries that the programmer has to answer. There might be a large number of procedure invocations during the execution of the bug symptom. Although the programmer is queried on only part of these calls, the number of queries may still be overwhelming.

Based on the above discussion, we would like to take advantage of the declarative approach and to avoid its shortcomings. In other words, when debugging, the oracle is only responsible for verifying or providing a reasonable number of elements in the intended interpretation for the program.

3.3 Declarative Diagnosis of Logic Programs

Declarative program diagnosis is an interactive process in which a declarative diagnoser acquires part of the intended interpretation for the program from the oracle and compares it with the actual interpretation of the program. Recently, there has been much research into declarative logic program diagnosis [33, 43, 44, 72, 74, 121, 122, 144]. Throughout this section, $\mathcal{P}$ is a definite program, $\mathcal{M}$ the intended interpretation for $\mathcal{P}$, A_w an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ and A_m an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$. We shall omit references to $\mathcal{P}$ and $\mathcal{M}$ when we talk about a symptom, a bug, validity, satisfiability, success and failure. For instances, a symptom will mean a symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$, validity will mean validity in $\mathcal{M}$ and success will mean success with $\mathcal{P}$.

The task of a declarative diagnoser is to search for a declarative bug, given a declarative symptom [74]. Many declarative diagnosers for logic programs have been proposed. Shapiro [121, 122] developed the algorithmic debugging method and exemplified the method through pure Prolog. Ferrand [43] adapted the algorithmic debugging method for definite logic programs. Lloyd [73, 74] presented a declarative diagnoser for arbitrary logic programs. The diagnoser is a meta-program which makes it easy to improve its performance by adding control information as meta-calls; and in particular a top-down diagnoser was derived in this way. Yan [144] improved the top-down diagnoser by reorganising its control information.

Each of these declarative diagnosers for logic programs has an inconsistency diagnosis procedure and an insufficiency diagnosis procedure. The inconsistency diagnosis procedure is called with $\mathcal{P}$, $\mathcal{M}$ and A_w as its inputs and returns a declarative bug. The insufficiency diagnosis procedure is called with $\mathcal{P}$, $\mathcal{M}$ and A_m as its inputs and returns a declarative bug. When describing an inconsistency diagnosis procedure or an insufficiency diagnosis procedure, we may rewrite it as a piece of Prolog code to which $\mathcal{P}$ and $\mathcal{M}$ are implicit inputs.

3.3.1 Inconsistency Diagnosis Procedures

The simplest inconsistency diagnosis procedure is just a rewriting of the stronger version of the definition 2.3.2.(1).

$$bug((A \leftarrow B)) \leftrightarrow clause(A, B), valid(B), unsatisfiable(A).$$

This is equivalent to the following rewriting of the Prolog code in [99].

```
%N.0 in Naish 91
```

```
inconsistency((A:-B)) :-
        clause(A,B),
        valid(B),
        unsatisfiable(A).
```

unsatisfiable(A) is true iff A is unsatisfiable and *valid*(B) is true iff B is valid. The implementation of *unsatisfiable*/1 and *valid*/1 varies from one diagnoser to another.

N.0 can be used to test if a clause instance is inconsistent. However, it is of little use as an inconsistency diagnosis procedure. Firstly, it must be applied to all clause instances, that is, it may have an infinite search space. Secondly, it does not relate the bug it returns to any inconsistency symptom. All the inconsistency diagnosis procedures in the literature restrict their search spaces to a subset of the set of clause instances.

The bottom-up inconsistency diagnosis procedures L.1 in [74] and F.1 in [43] search a space that consists of the clause instances that are used in all SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_w\}$. Each of the procedures can be rewritten as follows.

```
% inconsistency diagnosis procedure L.1 (F.1) in Lloyd 87 (Ferrand 87)
inconsistency((A,B),C) :-
        inconsistency(A,C).
inconsistency((A,B),C) :-
        inconsistency(B,C).
inconsistency(A,C) :-
        clause(A,B),
        inconsistency(B,C).
inconsistency(A,(A:-B)) :-
        \+ A=(_,_),
        unsatisfiable(A),       % to get an unsatisfiable instance
        clause(A,B),
        valid(B).
```

The clause instances are generated by the first three clauses and tested by the fourth clause. L.1 and F.1 search in a bottom-up manner. A clause instance $C\theta$ used to resolve with a goal G to derive another goal G^1 is tested only after all the clause instances that are used to resolve with G^1 have been tested. The input to `inconsistency/2` is carried by its first argument and the output from `inconsistency/2` by its second argument. When called with A_w, `inconsistency/2` first applies its third clause to generate the first clause instance of an SLD-derivation of $\mathcal{P} \cup \{\leftarrow A_w\}$, then applies its first and second clauses to select an atoms from the body of the clause instance and then again applies its third clause to generate the second clause instance of the SLD-derivation. `inconsistency/2` continues this process until the SLD-derivation cannot be extended. At that time, `inconsistency/2` tests the consistency of the last clause instance used in the SLD-derivation. After backtracking, `inconsistency/2` always tries to extend the SLD-derivation by using alternative clause instances before testing the consistency of the last untested clause instance used in the SLD-derivation.

While Ferrand [43] is only concerned with the theoretical aspects of declarative diagnosing, Lloyd [74] gives the following specifications for the procedures *unsatisfiable*/1 and *valid*/1. *unsatisfiable*(A) succeeds with all the unsatisfiable instances of A. *valid*(B) succeeds with all the valid instances of B. In practice, *unsatisfiable*(A) and *valid*(B) are implemented by means of queries to the oracle. When B is a conjunctive formula, it is broken down into its atomic components and the oracle is queried about these atoms. When *unsatisfiable*(A) (*valid*(A)) is invoked, the oracle is asked if A has an unsatisfiable (valid) instance and if so, the oracle is asked to provide a substitution θ such that $A\theta$ is unsatisfiable (valid). On backtracking, the oracle is asked if A has other unsatisfiable (valid) instances that have not been provided and if so, the oracle is asked to provide another substitution θ' such that $A\theta'$ is unsatisfiable (valid). If A has N unsatisfiable (valid) instances, then at most $N+1$ *yes/no* questions and N *instance* questions can be asked about A depending on how often backtracking to *unsatisfiable*(A) (*valid*(A)) occurs.

Top-down inconsistency diagnosis procedures L.2 in [74] and Y.1 in [144] further restrict their search spaces. They are equivalent to the following code.

```
% inconsistency diagnosis procedure L.2 (Y.1) in Lloyd 87 (Yan 90)
inconsistency((A,B), C) :-
        unsatisfiable(A),      % to get an unsatisfiable instance
        inconsistency(A,C).
inconsistency((A,B), C) :-
        unsatisfiable(B),
        inconsistency(B,C).
inconsistency(A, C) :-
        clause(A,B),
        call(B),               % to constrain search space
        unsatisfiable(B),
        inconsistency(B,C).
inconsistency(A, (A:-B)) :-
        \+ A=(_,_),
        unsatisfiable(A),      % to get an unsatisfiable instance
        clause(A,B),
        valid(B).
```

A clause instance in the search space of L.2 and Y.1 satisfies two conditions. Firstly, it is a clause instance that is used in any SLD-derivation of $\mathcal{P} \cup \{\leftarrow A_w\}$. Secondly, its head is an inconsistency symptom. The second condition constrains its search space to the set of all the clause instances that are used in all successful SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_w\}$.

The single stepping inconsistency diagnosis procedure S.1 in [121, 122] constrains its search space to the set of all the clause instances that can be used in all the SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_w\}$ via the left-to-right computation rule. It can be rewritten as below. The call *call(A)* in the second clause of S.1 fixes its computation rule and therefore constrains its search space.

```
% single stepping inconsistency diagnosis procedure S.1 in Shapiro 83
inconsistency((A,B),C) :-
        inconsistency(A,C).
inconsistency((A,B),C) :-
        call(A),            % control information
        inconsistency(B,C).
inconsistency(A, C) :-
        clause(A,B),
        inconsistency(B,C).
inconsistency(A, (A:-B)) :-
        clause(A,B),
        unsatisfiable(A),   % to get an unsatisfiable instance
        valid(B).
```

The original code in [121, 122] has extra control information that constrains itself to find only one inconsistent clause instance. The original code is quite different in form from the above code. We have transformed it to make its logic clearer.

S.1 assumes that whenever a call A is solved with a substitution θ, $A\theta$ is ground. It uses a ground oracle to implement *unsatisfiable*/1 and *valid*/1. The oracle is only asked if a ground atom is valid. Since the atom is ground, it is either unsatisfiable or valid. The groundness assumption is restrictive and the subsequent inconsistency diagnosis procedures do not use this assumption.

S.1 simulates Prolog's execution of $\mathcal{P}$ on A_w. Whenever a call A is solved with a substitution θ, S.1 asks the oracle if $A\theta$ is valid. If $A\theta$ is valid, S.1 continues to execute the remaining calls. Otherwise, an inconsistent clause instance has been found. Let $C = A \leftarrow B_1, B_2, \cdots, B_n$ be the clause instance used to solve the call A. According to the way that S.1 works and the implementation of the oracle, $(B_1, B_2, \cdots, B_n)\theta$ is already known to be valid. So, $C\theta$ is an inconsistent clause instance.

The above inconsistency diagnosis procedures search a space larger than necessary. In fact, an inconsistent clause instance must have been used in any successful derivation of $\mathcal{P} \cup \{\leftarrow A_w\}$. Therefore, we can constrain the search space to the set of the clause instances used in one successful derivation of $\mathcal{P} \cup \{\leftarrow A_w\}$.

The bottom-up inconsistency diagnosis procedure SS.1 in [125], the top-down inconsistency diagnosis procedure SS.2 in [125] and the divide-and-query inconsistency diagnosis procedure S.2 in [122] restrict their search spaces to the set of the clause instances that are used in one SLD-refutation of $\mathcal{P} \cup \{\leftarrow A_w\}$. That is, each of them searches a space that consists of all the clause instances that are used in one proof tree for $\mathcal{P}$ and A_w. They first construct a proof tree for $\mathcal{P}$ and A_w and then use different strategies to search it for an inconsistent clause instance.

We shall call the inconsistency diagnosis procedure SS.1 (SS.2) the bottom-up (top-down) tree search inconsistency diagnosis procedure so as to distinguish it from the bottom-up inconsistency diagnosis procedures L.1 and F.1 (top-down inconsistency diagnosis procedures L.2 and Y.1).

SS.1 traverses the proof tree in post order. When SS.1 visits a node A, it asks the oracle if A is valid. It stops at the first node A where the oracle answers with *no*, implying A is invalid. Because SS.1 traverses the proof tree in post order, the oracle must have answered with *yes* to all the queries about its children $B_1, B_2, \cdots, B_n$, implying that B_i is valid for $1 \leq i \leq n$. So, $A \leftarrow B_1, B_2, \cdots, B_n$ is an inconsistent clause instance. SS.1 is equivalent to the following.

```
% inconsistency diagnosis procedure SS.1 in Sterling and Shapiro 86
inconsistency((A,B),C) :-
        inconsistency(A,C).
inconsistency((A,B),C) :-
        inconsistency(B,C).
inconsistency(A, C) :-
        clause(A,B),
        call(B),
        !,                      % to constrain the search space
        inconsistency(B,C).
inconsistency(A, (A:-B)) :-
        clause(A,B),
        unsatisfiable(A),       % to get an unsatisfiable instance
        valid(B).
```

The number of the queries imposed by SS.1 is proportional to the number of the nodes in the proof tree in the worst case where the only inconsistency symptom in the proof tree is the root node.

Notice the difference between S.1 and SS.1. S.1 has a larger search space than SS.1 and therefore will usually ask more questions when diagnosing the same inconsistency symptom. For example, let $\mathcal{M} = \{p(1), \neg p(2), p(3), \cdots\}$ and $\mathcal{P}$ be the following program.

```
p(X) :-
        f(Y),b(X,Y),c(X).
p(X) :-
        a(X),d(X).
a(1).
a(2).
f(1).
f(2).
f(3).
b(1,3).
b(3,3).
c(1).
c(3).
d(1).
d(3).
d(2). %extra inconsistent clause
```

$p(2)$ is an inconsistency symptom and $d(2) \leftarrow$ an inconsistent clause instance. The successful SLD-derivation of $\mathcal{P} \cup \{\leftarrow p(2)\}$ is $\leftarrow p(2), \leftarrow (a(2), d(2)), \leftarrow d(2), \Box$ where $\Box$ is the empty goal. Before this successful SLD-derivation of $\mathcal{P} \cup \{\leftarrow p(2)\}$ is constructed, Prolog has pursued an unsuccessful SLD-derivation of $\mathcal{P} \cup \{\leftarrow p(2)\}$: $\leftarrow p(2), \leftarrow (f(2), b(2, Y), c(2)), \leftarrow (b(2, Y), c(2))$. So, the validity of $f(2)$ is queried by S.1. The validity of $f(2)$ is not queried by SS.1 because $f(2) \leftarrow$ is not used in the successful SLD-derivation of $\mathcal{P} \cup \{\leftarrow p(2)\}$.

Starting at the root of the proof tree, SS.2 [125] checks if there is an invalid atom among its children $\{B_1, B_2, \cdots, B_n\}$. If none of them is invalid, $A_w \leftarrow B_1, B_2, \cdots, B_n$ is an inconsistent clause instance. If there is an invalid atom, say B_i, among $\{B_1, B_2, \cdots, B_n\}$, SS.2 recurs on B_i since B_i is an inconsistency symptom. The worst case number of queries imposed by SS.2 is $b \times d$, where b is the branching factor of the proof tree and d is the depth of the proof tree. If we rewrote SS.2, it would have the same form as that for L.2 and Y.1 except that it would include an extra ! after $call(B)$ in its third clause to commit itself to one proof tree for $\mathcal{P}$ and A_w.

The divide-and-query inconsistency diagnosis procedure S.2 [122] uses the following property of a proof tree for $\mathcal{P}$ and A_w. If a node A in it is valid, then the subtree rooted at node A can be trimmed because it cannot contribute to the diagnosis. Starting with a proof tree for $\mathcal{P}$ and A_w, S.2 selects a node A in it, and asks the oracle if A is valid. If the oracle answers *yes*, then it trims the subtree rooted at A, otherwise it trims the other part of the proof tree. S.2 repeats the process until the height of the tree becomes 0. The only node left, A, and its children $B_1, B_2. \cdots, B_n$ in the initial proof tree correspond to an inconsistent clause instance: $A \leftarrow B_1, B_2, \cdots, B_n$.

Ideally, S.2 should select the node that splits the remaining tree into two halves to achieve the optimal worst case number of queries: $\log_2 n$. However, it is rarely possible to divide the tree in half. Usually, S.2 splits the tree into two parts which are nearly equal in size. If the tree has n nodes and branching factor b, then the worst case number of queries is $b \times log_2 n$ [74]. Plaisted [107] generalises the idea of S.2 by first transforming a proof tree into a binary tree and then reduces its search space by at least 1/3 at each step.

SS.1, SS.2 and S.2 use a ground oracle to decide if a ground atom A is valid. This implies that A_w and the clause instances in the proof tree for $\mathcal{P}$ and A_w are ground too. However, the proof tree may contain nodes which are not ground. When a node A in the proof tree is not ground, the oracle is asked to instantiate A to a ground instance, preferably an unsatisfiable ground instance [125].

The above inconsistency diagnosis procedures require the oracle to provide unsatisfiable instances of invalid atoms. The inconsistency diagnosis procedure D.1 proposed by Drabent *et al.* in [33] relieves the oracle from doing so. It works in a similar way to SS.2 except that it does not require the proof tree to be ground and it only asks the oracle if an atom is valid. The result that D.1 returns may contain some instance that is valid as well as some instance that is invalid.

Procedure	*Author(s)*	*Space*	*Strategy*	*Query*
L.1	Lloyd [74]	1	a	ii
F.1	Ferrand [43]	1	a	-
S.1	Shapiro [121, 122]	2	a	iii
L.2	Lloyd [74]	3	b	ii
Y.1	Yan [144]	3	b	ii
SS.1	Sterling and Shapiro [125]	4	a	iii
SS.2	Sterling and Shapiro [125]	4	b	iii
D.1	Drabent et al. [33]	4	b	i
S.2	Shapiro [122]	4	c	iii

Figure 3.1: Inconsistency Diagnosis Procedures: Summary

An inconsistency diagnosis procedure has three essential features: search space, search strategy and the type of queries. The search space that an inconsistency diagnosis procedure searches is either (1) the set of the clause instances that are used in all SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_w\}$, or (2) the set of the clause instances that are used in all the SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_w\}$ via the left-to-right computation rule, or (3) the set of the clause instances that are used in all successful SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_w\}$, or (4) the set of the clause instances that are used in one successful SLD-derivation of $\mathcal{P} \cup \{\leftarrow A_w\}$. The search strategy used by an inconsistency diagnosis procedure is either (a) bottom-up, or (b) top-down, or (c) binary. Each time the oracle is queried, he is asked either (i) a yes/no question about whether an atom A is valid, or (ii) a yes/no question about whether an atom A is valid, followed by an instance question requesting an unsatisfiable instance of A if A is not valid, or (iii) a yes/no question about whether an atom A is valid, followed by an instance question requesting a ground unsatisfiable instance of A if A is not valid.

Different combinations of these features result in different performances in terms of the number of queries and the preciseness of the output. We summarise the inconsistency diagnosis procedures by figure 3.1.

The search space and the search strategy influence the number of queries. The top-down and the binary strategies are better than the bottom-up strategy in terms of the worst case number of queries. So, the top-down or the binary strategy is a better choice than the bottom-up strategy. Lloyd has shown that neither the top-down nor the binary strategy is better than each other with respect to the worst case number of queries [74]. So, the choice should be left to the oracle.

The type of queries influences the preciseness of the output. The inconsistency diagnosis procedures [33, 98] that impose queries of type (i) return an inconsistent clause instance. The clause instance is too general because it may contain a valid instance. The inconsistency diagnosis procedures [74, 43, 144] that impose queries of type (ii) return an inconsistent clause instance that is unsatisfiable. The result delivered by the inconsistency diagnosis procedures [121, 122, 125] that impose queries of type (iii) is the most specific, i.e. a ground unsatisfiable clause instance. The oracle should be given the right to choose the type of queries he would like to answer.

3.3.2 Insufficiency Diagnosis Procedures

The simplest insufficiency diagnosis procedure is just a rewriting of definition 2.3.2.(2).

$$bug(uncovered(A)) \leftrightarrow valid(A), \forall B(clause(A, B) \rightarrow unsatisfiable(B)).$$

This can be transformed into the following.

$$bug(uncovered(A)) \leftrightarrow valid(A), \neg\exists B(clause(A, B) \wedge satisfiable(B)).$$

The above is equivalent to the following Prolog code in [99]. Notice that a formula is satisfiable iff it has a valid instance.

```
% N.0 insufficiency diagnosis procedure (Naish 91)
bug(uncovered(A)) :-
        valid(A),
        \+ (clause(A,B),valid(B)).
                       % instead of satisfiable(B)
```

N.0 can be used to test that an atom in the extended Herbrand base [66] is an uncovered atom. It is of little use as an insufficiency diagnosis procedure. It must be called with all the atoms in the extended Herbrand base. It does not relate the bug it returns to any insufficiency symptom. However, it may serve as a specification for insufficiency diagnosis procedures. The search space of the above code is the extended Herbrand base. All the insufficiency diagnosis procedures in the literature restrict their search spaces to a subset of the extended Herbrand base.

The following code is equivalent to the bottom-up insufficiency diagnosis procedures L.1 in [74]

```
% insufficiency diagnosis procedure L.1  in Lloyd 87
insufficiency((A,B),R) :-
        insufficiency(A,R).
insufficiency((A,B),R) :-
        insufficiency(B,R).
insufficiency(A,R) :-
        clause(A,B),
        insufficiency(B,R).
insufficiency(A,atom(A)) :-
        valid(A),
        \+ (clause(A,B),valid(B)).
```

L.1 searches a space that consists of the selected atoms in all SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_m\}$. The selected atoms are generated by the first three clauses and tested by the fourth clause. L.1 searches the space in a bottom-up manner in the following sense. If $\leftarrow W^1$ is derived from $\leftarrow W$ using clause C and *mgu* θ, then an atom in $\leftarrow W$ is tested only after all atoms in $\leftarrow W^1$ have been tested.

The following insufficiency diagnosis procedure given in [43] restricts its search space to the set of the valid instances of the selected atoms in all SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_m\}$. The validity is ensured by the call *valid(B)* in the third clause.

```
% insufficiency diagnosis procedure F.1 in Ferrand 87
insufficiency((A,B),R) :-
        insufficiency(A, R).
insufficiency((A,B),R) :-
        insufficiency(B,R).
insufficiency(A,R) :-
        clause(A,B),
        valid(B),
        insufficiency(B).
insufficiency(A,atom(A)) :-
        valid(A),
        \+ (clause(A,B),valid(B)).
```

As in the case of L.1, the first three clauses of F.1 serve as a generator and the fourth as a tester. F.1 searches its search space in the same way as L.1 except that F.1 typically has a smaller search space.

The top-down insufficiency diagnosis procedure L.2 in [74] further restricts its search space by ensuring that input to each call to *insufficiency*/2 is a valid failing atom. The search space consists of the valid failing instances of the selected atoms in all SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_m\}$. The validity is ensured by the call *valid(B)* in the third clause.

```
% insufficiency diagnosis procedure L.2 in Lloyd 87
insufficiency((A,B),R) :-
        \+ call(A),
        insufficiency(A,R).
insufficiency((A,B),R) :-
        \+ call(B),
        insufficiency(B,R).
insufficiency(A,R) :-
        clause(A,B),
        \+ call(B),
        valid(B),
        insufficiency(B,R).
insufficiency(A,atom(A)) :-
        valid(A),
        \+ (clause(A,B),valid(B)).
```

The first three clauses of L.2 serve as a generator and the fourth a tester. The combination of \+ and *call* in the first three clauses ensures that the input to recursive calls finitely fails. Given an insufficiency symptom A_m, L.2 first searches for a matching clause. If there is no such clause, then A_m is an uncovered atom. Otherwise, it searches for a valid instance of the body of a matching clause and recurs on that valid instance of the clause body. If no such valid instance exists, A_m is an uncovered atom. L.2 is essentially equivalent to the improved version of the insufficiency diagnosis procedure in [121, 122] which can be rewritten as the following.

```
% insufficiency diagnosis procedure S.1 in Shapiro 83
insufficiency((A,B), R) :-
        \+ call(A),
        insufficiency(A,R).
insufficiency((A,B), R) :-
        call(A),            %equivalent to \+ call(B) in L.2
        insufficiency(B,R).
insufficiency(A, R) :-
        clause(A,B),
        valid(B),
                            %\+ call(B) in L.2 must succeed
        insufficiency(B,R).
insufficiency(A, atom(A)) :-
                            %valid(A) in L.2 must succeed
        \+ (clause(A,B),valid(B)).
```

S.1 is more efficient than L.2 because it avoids the commented out calls by using the fact that the input to each call to *insufficiency*/2 is a valid failing atom.

Insufficiency diagnosis procedure N.1 in [99] has a more restricted search space than L.2 and S.1. The search space is composed of the valid failing instances of the selected atoms in all the SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_m\}$ via any computation rule by which an atom is selected only if all atoms to its left are valid.

```
% insufficiency diagnosis procedure N.1 in Naish 91
insufficiency((A,B),R) :-
        valid(B),
        insufficiency(A,R).
insufficiency((A,B), R) :-
        valid(A),           % to fix computation rules
        insufficiency(B,R).
insufficiency(A,R) :-
        valid(A),
        \+ call(A),
        clause(A,B),
        insufficiency(B,R).
insufficiency(A,atom(A)) :-
        valid(A),
        \+ call(A),
        \+ (clause(A,B),valid(B)).
```

When called with an insufficiency symptom A_m, N.1 searches for a valid instance of A_m. It then searches for a matching clause $A' \leftarrow B_1, B_2, \cdots, B_n$ and corresponding *mgu* θ, and then for a valid failing instance $B_i\theta\eta$ of $B_i\theta$ such that, for all $1 \leq j \leq i-1$, $B_j\theta\eta$ is valid and then recurs on $B_i\theta\eta$. If there are no such matching clause instances, then the valid instance of A_m is an uncovered atom. The behaviour of N.1 is equivalent to the insufficiency diagnosis procedure S.2 that is rewritten below.

```
% insufficiency diagnosis procedure S.2 in Shapiro 83
insufficiency(A,R) :-
        clause(A,B),
        (
                insufficiency1(B,R)
        ->
                true
        ;
                R = atom(A)
        ).

insufficiency1((A,B),R) :-
        valid(A),
        (
                call(A),
                insufficiency1(B,R)
         ;
                \+ call(A),
                insufficiency(A,R)
        ).
insufficiency1(A,R) :-
        valid(A),
        \+ call(A),
        insufficiency(A,R).
```

S.1 and L.2 find a valid instance of the whole clause body before searching for a failing instance of an atom in it while S.2 and N.1 check if a valid instance of an atom fails as soon as the instance is found.

The following insufficiency diagnosis procedure Y.1 [144] is slightly different from N.1.

```
% insufficiency diagnosis procedure Y.1 in Yan 90
insufficiency((A,B),R) :-
        insufficiency(A,R).
insufficiency((A,B),R) :-
        fail(B),
        insufficiency(B,R).
insufficiency(A,R) :-
        valid(A),
        fail(A),
        clause(A,B),
        insufficiency(B,R).
insufficiency(A,R) :-
        valid(A),
        fail(A),
        \+ (clause(A,B),valid(B)).
```

fail(B) states that $\mathcal{P} \cup \{\leftarrow B\}$ has a finitely failed SLD-tree. Note that \+ *call(A)* is different from *fail(B)*. This difference makes the functioning of the second clause of Y.1 different from that of the second clause of N.1. In N.1 a call to *valid(A)* is used to generate a failing instance of B while in Y.1 *fail(B)* is used to get a failing instance of B. The search space of Y.1 is the same as that of N.1.

All the above insufficiency diagnosis procedures search a space larger than is necessary. Each of them searches a subset of the selected atoms in all the SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_m\}$. In fact, an uncovered atom can be found by searching a space that consists of the selected atoms in all the SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_m\}$ via one fair computation rule.

Every insufficiency diagnosis procedure mentioned above requires the input to be an insufficiency symptom; the insufficiency diagnosis procedure D.1 proposed by Drabent *et al.* in [33] only requires its input to be an incompleteness symptom. An incompleteness symptom is an atom which has at least one instance that is an insufficiency symptom. Given an incompleteness symptom A_m, D.1 constructs a data structure called a search forest for A_m. The search forest for A_m is deducible from its SLD-tree and is constructed as follows. For each clause $A' \leftarrow B_1, B_2, \cdots, B_m$ such that A' and A_m unify with *mgu* θ_0, a tree rooted at (B_1, θ_0) is constructed. (B_1, θ_0) has $(B_2, \theta_0\theta_{1,1}), \ldots, (B_2, \theta_0\theta_{1,k})$ as its children where $\theta_{1,1}, \theta_{1,2}, \cdots, \theta_{1,k}$ are the computed answers for $B_1\theta_0$. The tree is extended from each leaf node (B, θ) until $B\theta$ has no computed answer or $B = \Box$.

D.1 queries the oracle, for each node (B, θ), as to whether $B\theta$ has a complete answer set. If $B\theta$ has an incomplete answer set, D.1 either recurs on $B\theta$ if (B, θ) is a branch node or returns $B\theta$ if (B, θ) is a leaf.

The result that D.1 gives is an incompletely covered atom. Compared with the other insufficiency diagnosis procedures for insufficiency diagnosis, the result is less precise. The reason for this is that the information provided by the oracle is not enough to decide if an atom is valid. Notice that the other insufficiency diagnosis procedures ask the oracle to provide valid instances of satisfiable atoms.

Drabent, Nadjm-Tehrani and Małuszynski [33] do not explicitly mention the SLD-tree. We note that each node (B, θ) of the search forest for A_m corresponds to the first atom (the selected atom) of a node $\leftarrow (B\theta, W)$ of the SLD-tree for $\mathcal{P} \cup \{\leftarrow A_m\}$ via the left-to-right computation rule such that $\leftarrow W\eta$ is a node subordinate to $\leftarrow (B\theta, W)$ for some η. The search forest for A_m can be considered as a top-level abstraction of the SLD-tree for $\mathcal{P} \cup \{\leftarrow A_m\}$ via the left-to-right computation rule. The way D.1 works resembles the way PTP [38, 39] works. For this reason, we shall call the search strategy used by D.1 the top-down zooming strategy.

Upon recursion with input $B\theta$, D.1 constructs the search forest for $B\theta$. We argue that the information needed to construct the search forest for $B\theta$ is deducible from the SLD-tree for $\mathcal{P} \cup \{\leftarrow A_m\}$ since the SLD-refutation of $\mathcal{P} \cup \{\leftarrow B\theta\}$ is a subcomputation of the SLD-refutation of $\mathcal{P} \cup \{\leftarrow A_m\}$.

An insufficiency diagnosis procedure also has three essential features: search space, search strategy and the type of queries. The search space that an insufficiency diagnosis procedure searches is either (1) the set of the selected atoms in all SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_m\}$, or (2) the set of valid instances of the selected atoms in all SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_m\}$, or (3) the set of valid failing instances of the selected atoms in all SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_m\}$, or (4) the set of the selected atoms in all the SLD-derivations of $\mathcal{P} \cup \{\leftarrow A_m\}$ via the left-to-right computation rule. The search strategy used by an insufficiency diagnosis procedure is either (a) bottom-up, or (b) top-down, or (c) top-down zooming. Notice the divide-and-query search strategy used for inconsistency diagnosis has not been used for insufficiency diagnosis. Each time the oracle is queried, they are asked either (i) a yes/no question about whether an atom A has a (more) valid instance, followed by an instance question requesting a valid instance of A, or (ii) a yes/no question about whether an answer set for an atom A is complete.

Different combinations of these features also result in different performances in terms of the number of queries and the preciseness of the output. We summarise the insufficiency diagnosis procedures by figure 3.2.

The search space and the search strategy influence the number of queries. The insufficiency diagnosis procedures using a top-down strategy are better than those using a bottom-up strategy in terms of the worst case number of queries because they use information from the oracle to constrain their search. So, a top-down strategy is a better choice than a bottom-up strategy.

Procedure	*Author(s)*	*Space*	*Strategy*	*Query*
L.1	Lloyd [74]	1	a	i
F.1	Ferrand [43]	2	a	-
S.1	Shapiro [121, 122]	3	b	i
L.2	Lloyd [74]	3	b	i
Y.1	Yan [144]	3	b	i
D.1	Drabent et al. [33]	4	c	ii

Figure 3.2: Insufficiency Diagnosis Procedures: Summary

The type of queries influences the preciseness of the output. The only insufficiency diagnosis procedure that imposes queries of type (ii) returns an incompletely covered atom, that is, it has an uncovered instance. It may contain an instance that is not an uncovered atom. The insufficiency diagnosis procedures that impose queries of type (i) return an uncovered atom.

3.3.3 Problems with Declarative Diagnosis

During declarative diagnosis, the oracle is required to provide part of the intended interpretation. Although only the finite part of the intended interpretation is used in a terminated computation, the computation may involve a large number of subcomputations. So, the quantity of the queries may be very large. Reducing the number of queries has been among the main objectives of all the declarative diagnosers.

One way to reduce the number of queries is to devise a more efficient strategy for searching proof trees and SLD-trees. Three strategies, namely, bottom-up, top-down and binary, have been devised for searching proof trees and three strategies, that is, bottom-up, top-down and top-down zooming have been used for searching SLD-trees. Let n be the number of nodes in a tree, b the largest number of branches that the nodes in the tree have and d the depth of the tree. The worst case numbers of nodes that bottom-up, top-down and binary strategies visit are respectively n, $b \times d$ and $b \times log_2 n$. The number of queries that an inconsistency diagnosis procedure imposes on the oracle is proportional to the number of nodes that its search strategy visits. If an insufficiency diagnosis procedure queries about the satisfiability of the conjunction of atoms labelling a node when it visits the node, then the worst case number of queries is also proportional to the number of the nodes that its strategy visits. However, it is much more difficult for the oracle to decide the satisfiability of a conjunction of atoms than to decide the satisfiability of a single atom. The solution used in the literature is to ask the oracle to provide the valid instances of each atom in the conjunction until the satisfiability of the conjunction is decided. This solution drastically increases the number of queries.

Another way to reduce the number of queries is to partially or completely automate the oracle. The methods proposed in the literature include:

1. when diagnosing an insufficiency symptom, using an execution mechanism to obtain a set of solutions [99] of an atom whose satisfiability is to be queried and thus relieving the oracle from providing the instances of the atom in the solution set, and

2. using assertions about the intended interpretation to partially automate the oracle [33]; for example, if there is an assertion dictating that an atom is unsatisfiable, then any query about the validity of its instances will have a negative answer.

During insufficiency diagnosis, the oracle is required to provide all the valid instances of atoms in a conjunction of atoms if the conjunction is unsatisfiable. This is difficult as well as tedious. The insufficiency diagnosis procedure in [33] requires the oracle to judge if an answer set for a call is complete. This solves part of the problem in the sense that the oracle is given a hint concerning the relationship between the answers and the call.

We now conclude our review on declarative diagnosis. The declarative diagnosis approach is preferable for logic programming diagnosis because the programmer does not need to know the operational behaviour of the symptom. The problem with the declarative approach is that the quantity of queries that the oracle has to reply to is typically very large. The methods used in the literature to reduce the quantity of queries are to use efficient search strategies and to partially automate the oracle.

3.4 Abstract Interpretation, Mode and Type Analyses

This section surveys related work on abstract interpretation, mode and type analyses in logic programs.

3.4.1 Abstract Interpretation of Logic Programs

Much work has been done on the abstract interpretation of logic programs; for a comprehensive survey, see [21]. A number of generic abstract semantics have been devised for the abstract interpretation of logic programs [9, 61, 87, 89] and they have found application in the detection of determinacy [24], data dependency analyses [15, 25, 55, 56, 95, 96], mode inference [10, 15, 25, 83, 95, 130], program transformation [117], type inference [10, 52, 57, 83] and proof of termination [133].

Abstract interpretation has been used in both forward and backward analyses of logic programs. A forward analysis [9] approximates the set of substitutions that might occur at each program point given a program and a set of goal descriptions. A backward analysis [4, 86, 87] approximates the set of the atoms that are logical consequences of a program [132].

There are various kinds of abstract interpreter of logic programs. A bottom-up forward abstract interpreter mimics a bottom-up evaluation strategy. A top-down forward abstract interpreter mimics a top-down evaluation strategy. Top-down forward abstract interpreters can be further divided into two sub-classes according to whether or not the underlying top-down evaluation strategy uses memoisation. A fixed-point forward abstract interpreter computes the least fixed-point of a system of simultaneous recurrence equations.

Bottom-up Forward Abstract Interpretation

The abstract interpreter in [61] simulates the bottom-up evaluation based on Alexander Templates (AT) [119]. Given a program and a goal, AT first transforms the program and the goal and then evaluates the transformed program and the transformed goal in a bottom-up manner. Given a program and a goal description, the AT-based abstract interpreter first transforms the program and the goal description in the same way as AT does and then mimics the evaluation phase of AT by replacing concrete unification with an abstract one.

Top-down Forward Abstract Interpretation without Memoisation

A top-down forward abstract interpreter without memoisation [9, 94, 135, 138] approximately executes a goal description by mimicking the underlying top-down evaluation strategy. As an example, we take the top-down forward abstract interpreter in [9].

Given a goal description that is a pair consisting of an atom and an abstract substitution, the top-down forward abstract interpreter in [9] constructs an abstract AND-OR graph to approximate the set of all the intermediate proof trees that may be constructed by SLD via the left-to-right computation rule for all the goals satisfying the goal description. In other words, any intermediate proof tree for any goal satisfying the goal description can be obtained by unravelling the abstract AND-OR graph. An AND-node is a clause head and its child OR-nodes are the atoms in the body of the clause. Every OR-node is adorned with one abstract substitution to the left, called the abstract call substitution, and with another to the right, called the abstract success substitution. The abstract success substitution of an OR-node is the abstract call substitution of its right sibling.

The initial abstract AND-OR graph has one OR-node that is the atom in the goal description and is adorned to the left with the abstract substitution in the goal description. Suppose that the abstract AND-OR graph has been partly constructed. Consider an OR-node A with abstract call substitution β. The abstract interpreter computes the abstract success substitution of the OR-node A as follows. For each clause $C_i \equiv H_i \leftarrow B_{(i,1)}, \cdots, B_{(i,m[i])}$ such that H_i may match with $A\theta$ for some θ satisfying β, the abstract interpreter adds to the OR-node A a child AND-node H_i that has $m[i]$ child the OR-nodes $B_{(i,1)}, \cdots, B_{(i,m[i])}$ and computes the abstract call substitution β^i_{in} of the OR-node $B_{(i,1)}$ - the first child OR-node of AND-node H_i. β^i_{in} approximates the set of the most general unifiers of H_i and $A\theta$ for all θ satisfying β. The abstract interpreter extends the OR-node $B_{(i,1)}$ by recursively applying the same process and extends the OR-node $B_{(i,j+1)}$ in the same way after it has computed the abstract success substitution of the OR-node $B_{(i,j)}$. Eventually, it will have computed the abstract success substitution β^i_{out} of the OR-node $B_{(i,m[i])}$. After computing β^i_{out} for each clause $C_i \equiv H_i \leftarrow B_{(i,1)}, \cdots, B_{(i,m[i])}$ such that H_i may match with $A\theta$ for some θ satisfying β, the abstract interpreter computes the abstract success substitution β' of the OR-node A where β' approximates from above the set of the most general unifiers of $A\theta$ and $H_i\eta$ for all the θ satisfying β and all the η satisfying β^i_{out}.

Since there are recursive calls, Bruynooghe [9] introduces a fixed-point component to the abstract interpreter. Suppose that an OR-node A with abstract call substitution β were to be extended. If A has an ancestor OR-node A' with abstract call substitution β' such that A is a variant of A and β is a variant of β', the abstract interpreter adorns the OR-node A to the right with the infimum abstract substitution and proceeds until the abstract success substitution of the OR-node A' is computed. The abstract interpreter then repeatedly recomputes the part of the AND-OR graph starting from the abstract success substitution of the OR-node A to the abstract success substitution of the OR-node A' by using the abstract success substitution for the OR-node A' as the abstract success substitution for the OR-node A. This fixed-point process finishes when there is no further increase in the value of the abstract success substitution of the OR-node A'. The same fixed-point component is also used to limit the size of the abstract AND-OR graph.

Kemp and Ringwood [67] made an attempt to give a least fixed-point characterisation of

Bruynooghe's work on abstract interpretation. The result is an abstract immediate consequence operator the least fixed-point of which corresponds to an abstract AND-OR tree - an AND-OR graph without circles.

The abstract interpreters in [94, 135, 138] differ from that in [9] only in dealing with recursive calls. Those in [94] and in [138] make use of a memo table and that in [135] uses stream predicates.

Top-down Forward Abstract Interpretation with Memoisation

An abstract interpreter based on an evaluation strategy with memoisation mimics the underlying evaluation strategy with memoisation by replacing concrete substitutions with abstract substitutions and the concrete unification with an abstract unification. For an introduction to evaluation strategies with memoisation, see [137].

The abstract interpreter based on the OLDT-resolution [63, 64, 65] mimics the OLDT-resolution [128]. The left-to-right computation rule is used in the OLDT-resolution. Given a goal that is a pair consisting of a sequence of atoms and a substitution, OLDT-resolution [128] constructs an OLDT structure for the goal. An OLDT structure consists of a search tree, a solution table and an association. Each entry of the solution table has a key and a solution list. The key is an atom and the solution list is a list of atoms that are instances of the key. Each node of the search tree is a pair consisting of a goal and a substitution, and each edge of the search tree is labelled with a substitution. The association is a group of pointers between the nodes of the search tree and the entries in the solution table.

Suppose that the OLDT structure has been partly constructed. Consider a node $< (A, R), \sigma >$ in the search tree where A is an atom, R a sequence of atoms and σ a substitution. If there is an entry in the solution table with a key that is a variant of $A\sigma$ then $< (A, R), \sigma >$ is called a lookup node. Otherwise, $< (A, R), \sigma >$ is called a solution node.

Initially, the search tree has one node that is the pair consisting of the sequence of atoms and the substitution, and both the solution table and the association are empty. OLDT-resolution extends the OLDT structure by extending its lookup nodes and its leaf solution nodes as follows until it cannot be further extended.

If $< (A, R), \sigma >$ is a leaf solution node then OLDT-resolution first adds to the solution table an entry whose key is $A\sigma$ and whose solution list is an empty list that will be filled in later. OLDT-resolution then, for each clause $C_i \equiv H_i \leftarrow B_{(i,1)}, \cdots, B_{(i,m[i])}$ such that H_i and $A\sigma$ unify with *mgu* θ, adds $< (B_{(i,1)}, \cdots, B_{(i,m[i])}, R), \sigma \circ \theta >$ as a child node to node $< (A, R), \sigma >$

and labels the edge from node $< (A, R), \sigma >$ to node $< (B_{(i,1)}, \cdots, B_{(i,m[i])}, R), \sigma \circ \theta >$ with θ. These child nodes will then be extended by the same process. When a unit clause is resolved with a leaf solution node $< (A, R), \sigma >$, the unit clause completes a sub-refutation for $A\sigma$ and also completes sub-refutations for the leftmost atoms of other nodes along the path from $< (A, R), \sigma >$ up to the root of the search tree. Whenever a unit clause is resolved with a leaf solution node, OLDT-resolution updates the solution lists for those keys that correspond to completed sub-resolutions. After the solution list for a key is updated, OLDT-resolution expands those lookup nodes that have pointers pointing to the solution list.

If $< (A, R), \sigma >$ is a lookup node then, OLDT-resolution first adds $< R, \sigma \circ \theta >$ as a child node to $< (A, R), \sigma >$ and labels the edge from $< (A, R), \sigma >$ to $< R, \sigma \circ \theta >$ with θ for each solution $A\sigma\theta$ in the solution list for the key $A\sigma$, and then adds to the association a pointer from lookup node $< (A, R), \sigma >$ to the tail of the solution list for key $A\sigma$. This pointer will be used to add more child nodes to lookup node $< (A, R), \sigma >$ because at the moment that lookup node $< (A, R), \sigma >$ is first extended, some solutions for $A\sigma$ might be unavailable from the solution list and will show up later.

Example Let $\mathcal{P}$ be the following program.

```
arc(a,b).
arc(b,c).
arc(c,a).

path(X,Y) :- arc(X,Y).
path(X,Z) :- arc(X,Y),path(Y,Z).
```

Figures 3.3 and 3.4 illustrate the sequence of OLDT structures created during the OLDT-resolution of $\mathcal{P} \cup \{\leftarrow path(a, A)\}$. In order to make OLDT structures concise, assignments of values to some variables in the substitution part of a node have been omitted. The OLDT-resolution of $\mathcal{P} \cup \{\leftarrow path(a, A)\}$ is completed in 11 steps. The OLDT structure created in i-th step is labelled with i on its upper left corner and the node extended in i-th step is surrounded by dotted box. Step 1 creates the initial OLDT structure. Steps 2,3,5,6,8,9 extend solution nodes while steps 4,7,10,11 extend lookup nodes.

The abstract interpreter based on OLDT closely mimics OLDT-resolution by constructing an abstract OLDT structure for a goal description that is a pair consisting of a sequence of atoms and an abstract substitution. The nodes of the abstract OLDT structure are ordered pairs comprising a sequence of atoms and an abstract substitution instead of a concrete substitution, and the edges of the abstract OLDT structure are now labelled with abstract substitutions instead of concrete substitutions. The key of a solution table entry is now a pair consisting

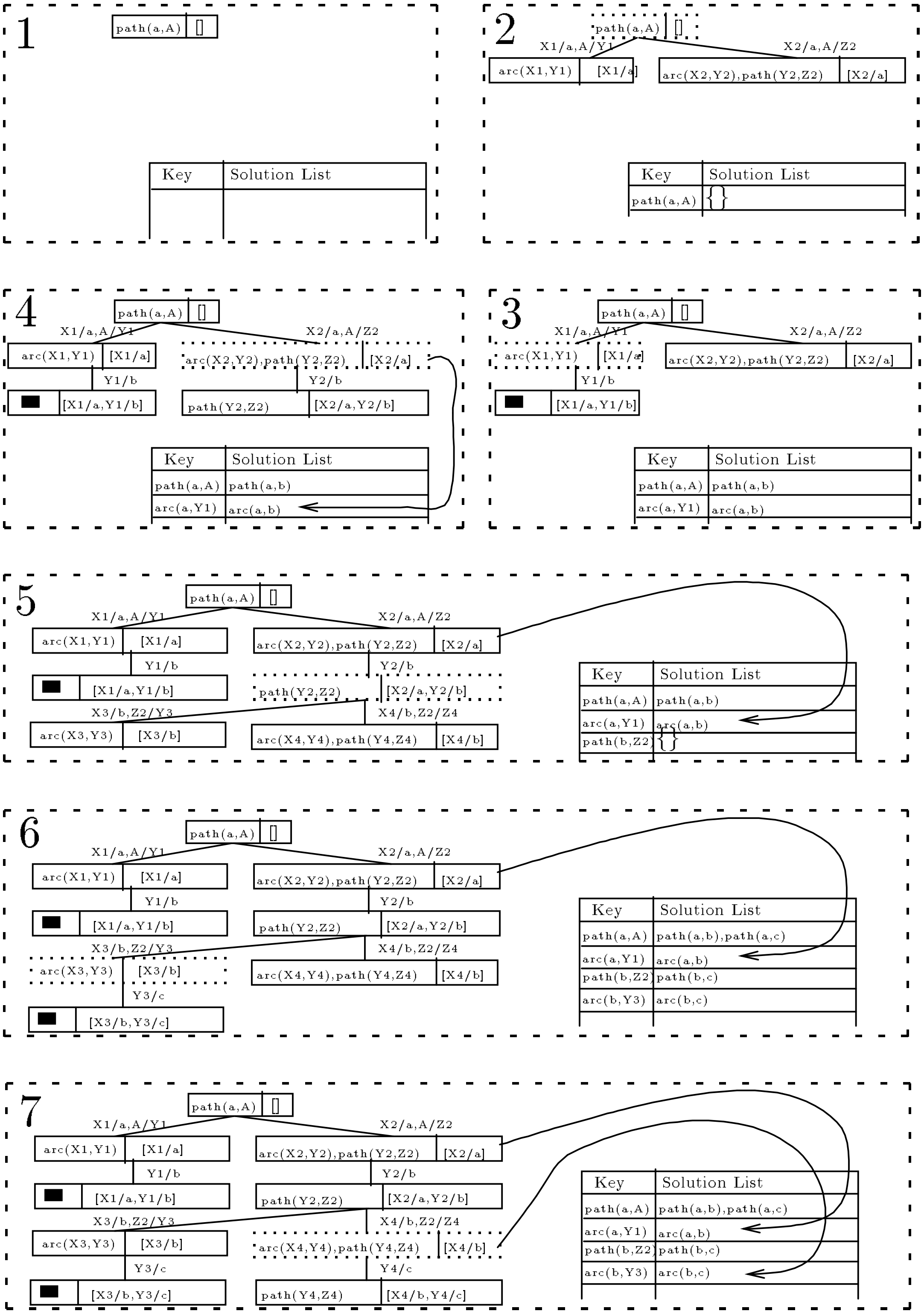


Figure 3.3: OLDT-resolution of $\mathcal{P} \cup \{\leftarrow path(a, A)\}$-part I

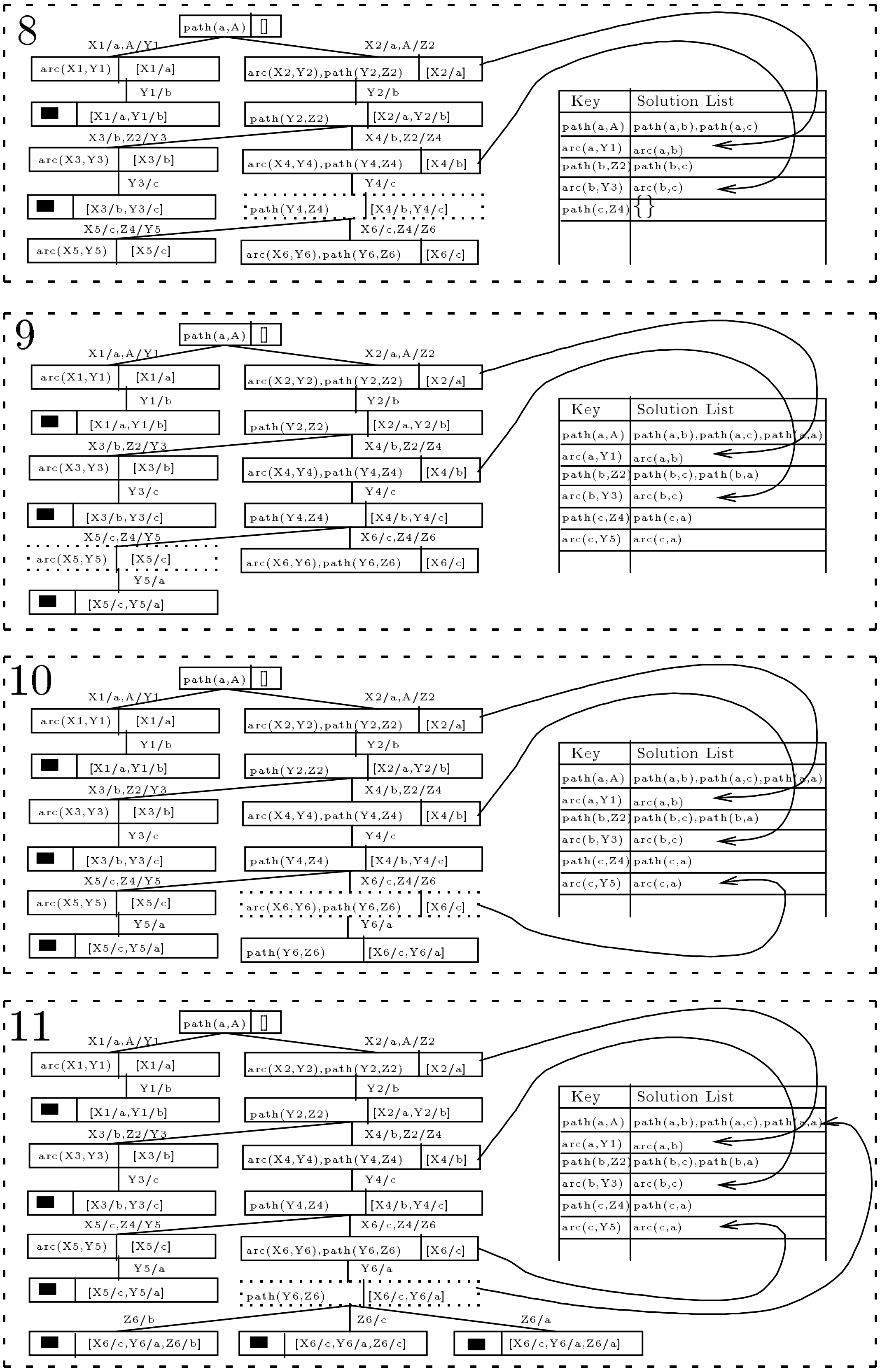


Figure 3.4: OLDT-resolution of $\mathcal{P} \cup \{\leftarrow path(a, A)\}$-part II

of an atom and an abstract substitution, and so is each solution in the solution list for the key. The OLDT-based abstract interpreter mimics OLDT by replacing the concrete unification function with an abstract unification function and the concrete composition function for concrete substitutions with an abstract composition function for abstract substitutions.

Fixed-point Forward Abstract Interpretation

Given a program and a set of goal descriptions that are abstract atoms, the abstract interpreter in [89] derives a system of concrete simultaneous recurrence equations whose least solution approximates the set of all the input atoms and the set of all the output atoms that occur in an intermediate proof tree derivable from the program and any goal satisfying one of the goal descriptions. The system of concrete simultaneous recurrence equations is approximated by a system of abstract simultaneous recurrence equations with each concrete operation being replaced by an abstract operation. Abstract interpretation is done by computing the least fixed-point of the system of abstract simultaneous equations. It is proved by [103] that the collecting semantics for a program and a set of goals defined in [89] is equivalent to the bottom-up computation of the magic-template transformation of the program and the set of goals [112].

Given a program and a set of goal descriptions each of which is a pair consisting of a goal and an abstract substitution, the abstract interpreter in [101, 102] derives a system of concrete simultaneous recurrence equations whose least solution gives to each program point a superset of the set of all the possible substitutions at the program point during the satisfaction of any goal satisfying one of these goal descriptions. A system of abstract simultaneous recurrence equations is derived to approximate the system of concrete simultaneous recurrence equations in the same manner as in [89]. Abstract interpretation is accomplished by computing in an abstract domain the least fixed-point of the system of abstract simultaneous recurrence equations.

The collecting semantics in [13] is the least fixed-point of an abstract immediate consequence operator that is derived from SLD-resolution. The collecting semantics for a definite program $\mathcal{P}$ maps an atom of the form $p(X_1, \cdots, X_n)$ and a set Θ of substitutions into the set of substitutions of the form $\theta \circ \sigma$ where $\theta \in \Theta$ and σ is an answer substitution of $\mathcal{P} \cup \{\leftarrow p(X_1, \cdots, X_n)\theta\}$. Abstract interpretation is accomplished by replacing the concrete operations in the collecting semantics with abstract ones.

The abstract interpreter in [101, 102] collects a set of substitutions for every program point. That in [89] collects the set of input atoms and the set of output atoms. Collecting the set of input atoms corresponds to collecting a set of substitutions for the entry point of each clause, applying each substitution in the set to the head of the clause to obtain a set of atoms for the clause and then putting together the sets of atoms for all the clauses as well as the given set of input atoms. Similarly, collecting the set of output atoms corresponds to collecting a set of substitutions for the exit point of each clause, applying each substitutions in the set to the head of the clause to get a set of output atoms for the clause and putting together the sets of output atoms for all the clauses.

In [60], data are collected only at the entry point of each program clause. The abstract interpreter in [60] is also a generic procedure; the core semantics is augmented with application dependent auxiliary functions that are similar to the abstract operations in [9]. These auxiliary functions operate on abstract domains consisting of appropriate approximations of the collecting semantics.

The existing generic abstract semantics are intended for those analyses whose results are used by compilers. In addition, they are not applicable to normal logic programs since they do not deal explicitly with negation as failure.

3.4.2 Mode Analysis

Multi-directionality is one of the attractive features of logic programming languages. An argument to a procedure can be used as an input or an output or both at the same time. However, it is often the case that a particular procedure or a particular clause is always called with some of its variables bound and some of its variables unbound. Information about the instantiation modes of variables helps the compiler generate efficient code by using different specialised unification algorithms for terms in different instantiation modes [136].

There has been much work into inferring mode information by abstract interpretation [10, 15, 25, 27, 83, 95, 130].

Let us briefly review various abstract domains for representing mode information about a set of substitutions. Let $\mathcal{V} = \{W, X, Y, Z\}$ be the set of the variables of interest and

$$\Theta_e = \left\{ \begin{array}{l} \{W/U,\ X/f(U,V),\ Y/U,\ Z/f(a,b)\}, \\ \{W/V,\ X/g(a,b),\ Y/V,\ Z/g(b,a)\} \end{array} \right\}$$

We shall use Θ_e to show how a set of substitutions is approximated in related work. The names of modes may differ from those in the literature because different names have been used to refer to the same mode in the literature.

Jones and Søndergaard [60], Kanamori [61], Mannila and Ukkonen [82], Nilsson [101] and Warren [137] consider only two modes $\{\mathbf{g}, \mathbf{a}\}$. $\mathbf{g}$ denotes the set of ground terms and $\mathbf{a}$ the set of all terms. Therefore, $\mathbf{g}$ is a more precise description for a ground term than $\mathbf{a}$. An abstract substitution is a function from $\mathcal{V}$ into $\{\mathbf{g}, \mathbf{a}\}$. Θ_e is best approximated by

$$\theta_1^\flat = \{W/\mathbf{a},\ X/\mathbf{a},\ Y/\mathbf{a},\ Z/\mathbf{g}\}$$

Because there are only two mode descriptions, an abstract substitution is often represented as the set of the variables that are mapped to $\mathbf{g}$ by the abstract substitution. For instance, $\theta_1^\flat$ is represented as $\{Z\}$. From $\theta_1^\flat$, one can infer that Z is instantiated to a ground term by each substitution in Θ_e. However, one can neither infer that every substitution in Θ_e instantiates W and Y to the same variable nor infer that some substitution in Θ_e may instantiate W, X and Y to terms that include common variables.

The work reported in [4, 15, 124] introduces into an abstract substitution a symmetric relation to capture possible sharing and (non-)linearity among variables. Let θ be a substitution and X and Y be different variables. If a $X\theta$ and $Y\theta$ contain a common variable then X and Y share under θ. If a variable occurs in $X\theta$ more than once then X is non-linear under θ. The symmetric relation which traces both sharing and linearity is referred to as a pair sharing relation. Notice that the word 'aliasing' has been used in place of the word sharing [15, 27]. We use the word 'aliasing' to refer to the case where two or more variables are instantiated to the same term. An abstract substitution consists of two parts: mode and pair sharing information. The mode information part is a function from $\mathcal{V}$ into $\{\mathbf{g}, \mathbf{a}\}$ and the pair sharing information part is a symmetric relation on $\mathcal{V}$. Θ_e is best approximated by

$$\theta_2^\flat = <\mathcal{I}_2, \mathcal{S}_2>$$

where

$$\mathcal{I}_2 = \{W/\mathbf{a},\ X/\mathbf{a},\ Y/\mathbf{a},\ Z/\mathbf{g}\}$$

and

$$\mathcal{S}_2 = \{(W,X),(X,W),(W,Y),(Y,W),(X,Y),(Y,X)\})$$

Pair sharing information is also used to increase the accuracy of mode analysis. However, because $\{\mathbf{g}, \mathbf{a}\}$ is not fine enough, one is still unable to infer that X and W are instantiated into the same variable by each substitution in Θ_e. The above abstract domain for inferring groundness and pair sharing information is called ASub by Codish *et al.* [17].

Bruynooghe, Janssens, Callebaut and Demoen [10] and Debray and Warren [27] use the modes $\{\mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{a}\}$. The set of terms in mode $\mathbf{e}$ is $\emptyset$ and the set of terms in mode $\mathbf{f}$ is $\mathcal{VAR}$. An abstract substitution is a function from $\mathcal{V}$ into $\{\mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{a}\}$. Θ_e is best approximated by

$$\theta_3^\flat = \{W/\mathbf{f},\ X/\mathbf{a},\ Y/\mathbf{f},\ Z/\mathbf{g}\}$$

$\theta_3^\flat$ approximates Θ_e more precisely than $\theta_1^\flat$ and the mode information in $\theta_2^\flat$. As exemplified in [27], it is now necessary to use sharing information in order to correctly infer mode information. Sharing information is not represented in the abstract substitution $\theta_3^\flat$. In [27], sharing among the variables in each clause is obtained by a local analysis. In [10], sharing is dealt with by the abstract built-in operation for unification.

The work reported in [9], infer both mode and pair sharing information. Mode information is a function from $\mathcal{V}$ into $\{\mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{a}\}$ and pair sharing information is a symmetric relation on $\mathcal{V}$. Θ_e is best approximated by

$$\theta_4^\flat = <\mathcal{I}_4, \mathcal{S}_4>$$

where

$$\mathcal{I}_4 = \{W/\mathbf{f},\ X/\mathbf{a},\ Y/\mathbf{f},\ Z/\mathbf{g}\}$$

and

$$\mathcal{S}_4 = \{(W,X),(X,W),(W,Y),(Y,W),(X,Y),(Y,X)\})$$

Debray [24] also infers pair sharing information together with mode information. One more mode **nf** is introduced to increase accuracy. **nf** represents the set of non-variable terms. The set of terms in mode **nf** is a superset of the set of terms in mode **g** and is a subset of the set of terms in mode **a**. As in [9], pair sharing information is also used to ensure the correctness of mode analysis. The mode information part of an abstract substitution is a function from $\mathcal{V}$ into $\{\mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{nf}, \mathbf{a}\}$. Θ_e is best approximated by

$$\theta_5^\flat = <\mathcal{I}_5, \mathcal{S}_5>$$

where

$$\mathcal{I}_5 = \{W/\mathbf{f},\ X/\mathbf{nf},\ Y/\mathbf{f},\ Z/\mathbf{g}\}$$

and

$$\mathcal{S}_5 = \{(W,X),(X,W),(W,Y),(Y,W),(X,Y),(Y,X)\})$$

Mellish [88, 89] uses one more mode **iu**. The immediate sub-terms of a term in mode **iu** are variables. The set of terms in mode **iu** is a subset of the set of terms in mode **nf**. An abstract substitution is a function from $\mathcal{V}$ into $\{\mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{iu}, \mathbf{nf}, \mathbf{a}\}$ and sharing information is not represented. Θ_e is best approximated by

$$\theta_6^\flat = \{W/\mathbf{f},\ X/\mathbf{nf},\ Y/\mathbf{f},\ Z/\mathbf{g}\}$$

Codish, Dams, Filé, and Bruynooghe [16] infer mode information together with pair sharing information. The mode component can be seen as a function from $\mathcal{V}$ to $\{\mathbf{f}, \mathbf{a}\}$. Cortesi and Filé [18] infer mode information together with sharing, covering and equivalence. A set S_1 of variables covers another set S_2 of variables under a set Θ of substitutions if $\bigcup_{X \in S_1} vars(X\theta) \supseteq \bigcup_{X \in S_2} vars(X\theta)$ for each $\theta \in \Theta$. S_1 and S_2 are equivalent under Θ if $\bigcup_{X \in S_1} vars(X\theta) = \bigcup_{X \in S_2} vars(X\theta)$ for each $\theta \in \Theta$. The abstract domain is called EXP. The mode component can be seen as a function from $\mathcal{V}$ to $\{\mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{o}, \mathbf{a}\}$ where $\mathbf{o}$ represents the set of non-variable, non-ground terms.

Jacobs and Langen [55] proposed an abstract domain, called Sharing, for inferring sharing information. An abstract substitution $\theta^\flat$ is a set of sets of variables in the program. A set $\{V_1, \cdots, V_n\}$ of variables in $\theta^\flat$ says that there may be a substitution θ described by $\theta^\flat$ such that $\bigcap_{1 \leq i \leq n} vars(V_i\theta) \neq \emptyset$. The sharing represented in this way is referred to as set sharing. An abstract substitution also captures groundness information. The absence of a variable from an abstract substitution implies that the variable is ground under any substitution described by the abstract substitution. Θ_e is best approximated by the following abstract substitution.

$$\theta_7^\flat = \{\{W, X, Y\}, \{X\}, \{W, Y\}\}$$

Sundararajan and Conery [126] augmented the Sharing domain with a component representing freeness and another representing non-linearity. An abstract substitution is a triple with its first two components being sets of variables and its third component being a set of sets of variables. The first component contains those variables that are free under all substitutions described by the abstract substitution, the second component contains those variables that are non-linear under some substitutions described by the abstract substitution, and the third component is the same as an abstract substitution in the Sharing domain. Θ_e is best approximated by the following abstract substitution.

$$\theta_8^\flat = <\{W, Y\}, \emptyset, \{\{W, X, Y\}, \{X\}, \{W, Y\}\}>$$

3.4.3 Type Analysis

Prolog is a type-free language. The programmer does not have to specify the types of variables, functions and predicates. This may make it simple to write simple programs. However, it also makes it difficult to debug programs because type errors cannot be detected by Prolog systems. A type error will manifest itself in the form of an inconsistency symptom or an insufficiency symptom rather than an indication of a type violation.

There have been many efforts to augment Prolog with type systems in the forms of type checking systems and type analysis systems. A type checking system [8, 46, 97] enables the programmer to define types and declare types for program objects. It typically provides additional syntax for type definitions and type declarations and a type checker for detecting type inconsistencies. The programmer defines types for function symbols and declares types for arguments of predicates. The type checker checks, clause by clause, the type consistency of a program with respect to some consistency criterion. A clause which satisfies the criterion is called *well-typed*. The program is well-typed iff every clause in it is well-typed. A well-typed program will never call a predicate with arguments of wrong types when executed with well-typed queries [97]. A type checking system imposes an additional overhead on the programmer.

Type analysis systems infer types from the text of a program. Some type analysis systems infer a type for the program and the type is meant to denote a superset of the success set of the program [3, 23, 50, 51, 90, 115, 142, 145, 146]. Other type analysis systems infer types for procedure calls and successes [10, 52, 57, 62, 65].

There are three main approaches for inferring types [51]. The first approach obtains from a program a system of formulae which describes set-theoretic relationships between the atoms appearing in the program. A type is then defined to be a distinguished model of the system of formulae. Examples of this approach are to be found in [3, 23, 90, 115, 142, 145, 146]. The second approach uses an abstract immediate consequence operator to approximate the immediate consequence operator $T_{\mathcal{P}}$ associated with the program $\mathcal{P}$ [132]. A type is defined as a fixed-point of the abstract immediate consequence operator. An example of this approach appears in [145]. Heintze and Jaffar [50, 51] show that the first and the second approaches are equivalent in the sense that a type in one approach has its counterpart in the other. The third approach involves specialising generic abstract semantics by developing abstract domains and their associated abstract operations. Some examples are to be found in [10, 52, 57, 62, 65].

Most type analysis systems infer descriptive types. A descriptive type is a description of a set of terms by an expression in a type language. Different type analysis systems may use different type languages. Most of them use regular types or deterministic regular types [23] in one form or another. A descriptive type is not intuitive. Although the sets of terms described by descriptive types are well-defined, it is difficult for the programmer to understand descriptive types. Horiuchi and Kanamori [52, 62] and Kanamori and Kawamura [65] infer prescriptive types where the types of function symbols are provided by the programmer.

We now review four typical type analysis systems involving different approaches and different kinds of types.

Finding Models for Systems of Formulae

Mishra [90] made the first attempt to infer types for logic programs and is typical of those type analysis systems that infer descriptive types by finding a distinguished model for a set of formulae that captures set-theoretic relationships between the atoms appearing in the program. The type analysis system infers a type for each argument of each predicate in the program. The type of a predicate is the Cartesian product of the types of its arguments and the type of the program is the collection of the types of the predicates in the program. Types are ground regular trees as defined as follows.

A regular tree is either

- the empty set $\emptyset$; or
- a constant; or
- a variable; or
- a tuple $(r_1, \cdots, r_n)$ where $r_1, \cdots, r_n$ are regular trees; or
- $f(r)$ with r being a regular tree and f a function symbol; or
- $r + s$ with r and s being regular trees; or
- $z!g(z)$ where $g(z)$ is a regular tree and z a variable. Here ! is an operator which takes a variable and a regular tree to form another regular tree.

A ground regular tree is a regular tree where variables only occur in the expressions of the form $z!g(z)$. The set of terms denoted by a ground regular tree GRT is $\check{\gamma}^t(GRT)$ where $\check{\gamma}^t$ is defined as follows.

- $\check{\gamma}^t(\emptyset) = \emptyset$;

- $\check{\gamma}^t(c) = \{c\}$;
- $\check{\gamma}^t((r_1, \cdots, r_n)) = \{(t_1, \cdots, t_n) | t_1 \in \check{\gamma}^t(r_1) \wedge \cdots \wedge t_n \in \check{\gamma}^t(r_n)\}$;
- $\check{\gamma}^t(f(r)) = \{f(t) | t \in \check{\gamma}^t(r)\}$;
- $\check{\gamma}^t(r + s) = \check{\gamma}^t(r) \cup \check{\gamma}^t(s)$;
- $\check{\gamma}^t(z!g(z))$ is the least solution of $Z = \{g(z) | z \in Z\}$ with respect to set inclusion $\subseteq$.

For a program whose clauses do not share variables, the analysis system first assigns a type to each argument of each predicate and a type variable to each program variable, then sets up a system of inequalities over regular trees involving these types, and then gives a ground regular tree to each of these types by finding the maximal solution of the system of inequalities.

Approximating the Immediate Consequence Operator $T_{\mathcal{P}}$

Yardeni and Shapiro [145] infer a type for a program $\mathcal{P}$ by computing the least fixed-point of a function that approximates $T_{\mathcal{P}}$. A type is a recursively enumerable set of ground atoms that is tuple-distributive. A set of terms is tuple-distributive if it is equal to the set of terms constructed recursively by permuting each argument position among all terms that have the same function symbol [90]. The tuple-distributive closure $\alpha(S)$ of S is defined as follows.

$$\alpha(S) \stackrel{def}{=} \{f(t_1, \cdots, t_n) | t_i \in \alpha(f_i^{-1}(S))\} \cup \{c | c \text{ is a constant}\}$$

where $f_i^{-1}(S) \stackrel{def}{=} \{t_i \mid f(t_1, \cdots, t_n) \in S\}$ [50].

The type of a program $\mathcal{P}$ is defined as $lfpT_{\mathcal{P}}^{\alpha}$ where $T_{\mathcal{P}}^{\alpha}$ is defined as follows. Let X be a type.

$$T_{\mathcal{P}}^{\alpha}(X) \stackrel{def}{=} \alpha(T_{\mathcal{P}}(X))$$

$lfpT_{\mathcal{P}}^{\alpha}$ is a superset of $lfpT_{\mathcal{P}}$. Yardeni and Shapiro [145] do not give an algorithm for inferring types. Instead, an algorithm for testing the membership of a ground atom in the type of the program $\mathcal{P}$ is given.

Abstract Interpretation with Descriptive Types

Bruynooghe, Janssens, Callebaut, and Demoen [10] and Janssens and Bruynooghe [57] use descriptive types to approximate sets of terms. The descriptive types are graphs, called type graphs. Note that our presentation of type graphs is based on that in [22] which is more concise than those in [10, 57].

Bruynooghe, Janssens, Callebaut, and Demoen [10] use restricted types to approximate sets of ground terms. A restricted type is a 4-tuple $G = <N_t, N_f, r, E>$ where

- N_t is a set of type nodes labelled with different types,
- N_f is a set of functor nodes labelled with functors,
- $N_t \cap N_f = \emptyset$,
- $r \in N_t$, called the root of G, has no incoming arc, and there is a path from r to any other node in G, and
- $E \in \wp(N_t \times N_f) \cup \wp(N_f \times nat \times N_t)$ is a set of arcs, with nat being the set of natural numbers, such that
  - every type node $k \in N_t$ has at least one outgoing arc and if $<k, m_1>, <k, m_2> \in E$ then m_1 and m_2 are functor nodes with distinct labels,
  - each functor node $m \in N_f$ labelled with f/n has n outgoing arcs $<m, i, k_i>$, $1 \leq i \leq n$ where k_i is a type node, and
  - the number of occurrences of the same functor in any acyclic path in G is bounded by a constant.

Let $G = <N_t, N_f, r, E>$ be a restricted type, $k \in N_t$ and $lb(n)$ be the label of n in $N_t \cup N_f$. The set of ground terms denoted by k is the set of ground terms that fold on k. A ground term t is said to fold on k iff

- $t = c$ is a constant, $<k, m> \in E$ and $lb(m) = c$; or
- $t = f(t_1, \cdots, t_n)$, $<k, m> \in E$, $lb(m) = f/n$, $<m, i, k_i> \in E$ and t_i folds on k_i for $1 \leq i \leq n$.

The set of ground terms denoted by a restricted type $G = <N_t, N_f, r, E>$ is

$$\check{\gamma}^t(G) \stackrel{def}{=} \{t \mid t \ folds \ on \ r\}$$

Let $\mathcal{GRAPH}$ be the set of restricted types constructible from Σ and $G, G' \in \mathcal{GRAPH}$. Define $G \sim G' \stackrel{def}{=} \check{\gamma}^t(G) = \check{\gamma}^t(G')$. $\sim$ is an equivalence relation on $\mathcal{GRAPH}$. The set $\mathcal{GRAPH}/_\sim$ of the equivalence classes of $\sim$ is finite. Define $[G]_\sim \check{\sqsubseteq}^t [G']_\sim \stackrel{def}{=} \check{\gamma}^t(G) \subseteq \check{\gamma}^t(G')$. $<\mathcal{GRAPH}/_\sim, \check{\sqsubseteq}^t>$ is a complete lattice.

An abstract substitution approximating a set Θ of ground substitutions θ with $dom(\theta) \subseteq \mathcal{V}$ is a function from $\mathcal{V}$ to $\mathcal{GRAPH}/_{\sim}$ in [10].

Since restricted types can only approximate sets of ground terms, rigid types and integrated types are introduced to approximate sets of terms containing non-ground terms [57]. A rigid type differs from a restricted type in the following way. A type node k may be labelled with a primitive type that denotes a set of terms. For instance, a type node may be labelled with int which denotes the set of integers $\cdots, -1, 0, 1, \cdots$. A type node k labelled with a primitive type T does not have any outgoing arc and any term in the set of terms denoted by T folds on k. There is a special primitive type any denoting $\mathcal{TERMS}$. All other primitive types denote sets of ground terms. $\check{\gamma}^t, \mathcal{GRAPH}, \sim, \mathcal{GRAPH}/_{\sim}, \check{\sqsubseteq}^t$ can be carried over from restricted types to rigid types with the notion of folding adapted as above.

An abstract substitution based on rigid types that approximates a set Θ of substitutions θ with $dom(\theta) \subseteq \mathcal{V}$ has two parts: a type part and an aliasing part. The type part is a function from $\mathcal{V}$ to $\mathcal{GRAPH}/_{\sim}$. Note that the type part also captures the groundness information. A variable is deemed to be instantiated to a ground term by any substitution in Θ if the type part maps the variable to a rigid type not containing any. The aliasing part captures aliasing relations between sub-terms of the terms to which variables in $\mathcal{V}$ are instantiated by substitutions in Θ. The aliasing part is used to improve the accuracy of type analysis.

An integrated type is an enrichment of a rigid type in that an extra primitive type var which denotes $\mathcal{VAR}$ is allowed. Note that any variable in $\mathcal{VAR}$ folds on a type node labelled with var. The primitive type captures freeness information, i.e., if a term is of type var then it is a variable. $\check{\gamma}^t, \mathcal{GRAPH}, \sim, \mathcal{GRAPH}/_{\sim}, \check{\sqsubseteq}^t$ can be carried over from rigid types to integrated types with the above enriched notion of folding.

An abstract substitution based on intergrated types that approximates a set Θ of substitutions θ with $dom(\theta) \subseteq \mathcal{V}$ has three parts: a type part, a pair sharing part, and an aliasing part. The type part and the aliasing part are the same as those for rigid types. The pair sharing part is necessary to ensure the correctness of the type analysis because the type part embodies both freeness and groundness information.

The generic abstract semantics developed by Bruynooghe [9] is specialised for inferring restricted, rigid and integrated types. For each case, algorithms for abstract operations required by the generic abstract semantics are developed and proved to satisfy corresponding safety conditions.

Abstract Interpretation with Prescriptive Types

Horiuchi and Kanamori [52, 62] and Kanamori and Kawamura [65] use prescriptive types to approximate sets of terms. Types are introduced by type definitions.

Types in [62, 65] are defined by type predicates that are unary predicates whose names are the names of types. For instance, type *nat* is defined by the following type predicate.

$$nat(0).$$
$$nat(s(X)) \leftarrow nat(X)$$

The set $\check{\gamma}^t(M)$ of terms denoted by a type M is the set of terms t such that $M(t)$ succeeds, e.g., $\check{\gamma}^t(nat) = \{0, s(0), \cdots, \}$. Type definitions by unary predicates can be equivalently rewritten using a binary predicate $:/2$ which relates the type of a term $f(t_1, \cdots, t_n)$ to the types of $t_1, \cdots, t_n$. For instance, the type *nat* can be equivalently defined as in the following.

$$\begin{array}{rcl} 0 : nat & \leftarrow & \\ s(X) : nat & \leftarrow & X : nat \end{array}$$

It is assumed that there are k type predicates $M_1, \cdots, M_k$ such that $\check{\gamma}^t(M_1), \cdots \check{\gamma}^t(M_k)$ are disjoint. *any* with $\check{\gamma}^t(any) = \mathcal{TERMS}$ is introduced so as to approximate sets of terms not denoted by types in $\{M_1, \cdots, M_k\}$, and *empty* with $\check{\gamma}^t(empty) = \emptyset$ is introduced so that $<\{any, empty, M_1, \cdots, M_k\}, \check{\sqsubseteq}^t>$ is a complete lattice where $M \check{\sqsubseteq}^t M' \stackrel{def}{=} \check{\gamma}^t(M) \subseteq \check{\gamma}^t(M')$.

An abstract substitution approximating a set Θ of substitutions θ with $dom(\theta) \subseteq \mathcal{V}$ is a function from $\mathcal{V}$ to $\{any, empty, M_1, \cdots, M_k\}$.

Horiuchi and Kanamori [52] use polymorphic type definitions by allowing type expressions to be parameterised by type parameters. For instance, the type of a list whose elements are from an unknown type β can be defined as follows where $list/1$ is a type constructor.

$$list(\beta)([\,]).$$
$$list(\beta)([H|T]) \leftarrow \beta(H), list(\beta)(T)$$

Polymorphic type definitions can be equivalently rewritten using the binary predicate $:/2$ in the same way as mono-morphic type definitions. The following is equivalent to the above type definition.

$$\begin{array}{rcl} [\,] : list(\beta) & \leftarrow & \\ [H|T] : list(\beta) & \leftarrow & H : \beta, T : list(\beta) \end{array}$$

Let *any* denote $\mathcal{TERMS}$. Note that $list(nat)$ and $list(list(nat))$ are well-defined type expressions. A *mono-type* is a type that does not contain a type parameter, and a *poly-type* is a type containing at least one type parameter. $list(nat)$ is a mono-type while $list(\beta)$ is a poly-type. There are normally an infinite number of mono-types. Function symbols used in type definitions to denote types such as nat and $list$ are called type constructors. Let $\mathcal{CONS}$ be the set of type constructors. The set $\mathcal{MTYPES}$ of mono-types is the set of ground terms constructible from $\mathcal{CONS}$. $\mathcal{MTYPES}$ is usually an infinite set. Let $:/2$ be augmented with an extra clause $X : any \leftarrow$. The set $\check{\gamma}^t(M)$ of terms denoted by a mono-type M is the set of terms t such that $t : M$ succeeds. $< \{any, empty\} \cup \mathcal{MTYPES},\ \check{\sqsubseteq}^t\ >$ is a complete lattice where $M \check{\sqsubseteq}^t M' \stackrel{def}{=} \check{\gamma}^t(M) \subseteq \check{\gamma}^t(M')$.

An abstract substitution approximating a set Θ of substitutions θ with $dom(\theta) \subseteq \mathcal{V}$ is a function from $\mathcal{V}$ to $\{any, empty\} \cup \mathcal{MTYPES}$.

The generic abstract semantics developed in [65] is specialised for inferring the prescriptive types. An algorithm for abstract unification required by that generic abstract semantics is developed and proved to satisfy its safety conditions.

3.5 Summary

There are five main approaches to bug detection: program understanding, program verification, well-formedness checking, program property analysis and program testing. The program understanding approach and the program verification approach are computationally expensive and they have been used only in tutoring systems. The well-formedness checking approach and the program property analysis approach are computationally feasible but they can only give warnings against the possible existence of bugs. Program testing is less costly in computation than program understanding and program verification but more expensive than program property analysis and well-formedness checking. We aim at automating bug detection with a low computational cost and a high reliability. The program understanding approach and the program verification approach are not suitable since they are computationally costly. The well-formedness checking approach does not detect semantics-related bugs. Among the two other approaches, the program property analysis approach generates warnings against the possible existence of bugs with a lower computational cost and the program testing approach detects bug symptoms with a higher computational cost. We detect bugs by combining the program property analysis approach and the program testing approach. The program is first checked against a set of assertions associated with it by means of program analysis and then tested on a set of test data if it fails to satisfy the set of assertions associated with it.

There are three main approaches for bug diagnosis: the information filtering, knowledge-based and declarative approaches. The information filtering approach compromises the declarativeness of logic programming since it requires the programmer to have the knowledge of the operational semantics of logic programs. In addition, the level of detail at which information is presented to the programmer is not necessarily appropriate to the problem at hand. A system adopting the knowledge-based approach may not be able to locate a bug, for a given bug symptom, due to the incompleteness of its knowledge base. Also, it usually requires the programmer to describe a bug symptom in more detail. A system adopting the declarative approach upholds the declarativeness of logic programming and guarantees to locate a declarative bug for a given declarative symptom. Therefore, the declarative approach is preferable for the diagnosis of logic programs. The main problem with the declarative approach is that the quantity of queries imposed on the oracle is very large. We tackle this problem by making use of the acquired part of the intended interpretation to reduce the size of the search spaces for bug symptoms. For a given bug symptom, the diagnoser that we have developed searches a smaller search space than existing diagnosers. It is also generic in that it can be used with different search strategies.

There are three main kinds of abstract interpreter of logic programs. A bottom-up forward abstract interpreter mimics a bottom-up evaluation strategy. A top-down forward abstract interpreter mimics a top-down evaluation strategy. A fixed-point forward abstract interpreter computes the least fixed-point of a system of simultaneous recurrence equations. The existing generic abstract semantics are intended for those analyses whose results are used by compilers. In addition, they are not applicable to normal logic programs since they do not deal explicitly with negation as failure. In Chapter 4, we shall present a fixed-point generic abstract semantics for the abstract interpretation of normal logic programs. This generic abstract semantics deals explicitly with negation as failure and allows finer analyses than other fixed-point generic abstract semantics. In Chapter 5, we shall specialise the generic abstract semantics for mode and type analyses.

Chapter 4

Generic Abstract Semantics

This chapter derives two generic fixed-point abstract semantics $F^{\flat}_{\mathcal{P},\mathcal{G}}$ and $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ for forward analyses of normal logic programs under the SLDNF-resolution via the left-to-right computation rule. The difference between $F^{\flat}_{\mathcal{P},\mathcal{G}}$ and $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ is that $F^{\flat}_{\mathcal{P},\mathcal{G}}$ is more fin-grained than $F^{\diamond}_{\mathcal{P},\mathcal{G}}$. $F^{\flat}_{\mathcal{P},\mathcal{G}}$ and $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ can be used to perform forward analyses of *normal* logic programs while the existing generic abstract semantics in the literature can only be used to perform forward analyses of *definite* logic programs.

$F^{\flat}_{\mathcal{P},\mathcal{G}}$ is based on a fixed-point collecting semantics that is obtained through two approximations from the operational semantics - SLDNF-resolution. $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ is obtained from the collecting semantics by a further approximation and it can be specialised to perform various analyses under certain sufficient conditions. $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ is derived from $F^{\flat}_{\mathcal{P},\mathcal{G}}$ through one more approximation and it can be specialised to perform various analyses under the same sufficient conditions as $F^{\flat}_{\mathcal{P},\mathcal{G}}$.

Section 4.1 reformulates SLDNF in order to facilitate the derivation of the collecting semantics. Section 4.2 derives the collecting semantics from the operational semantics. Section 4.3 derives $F^{\flat}_{\mathcal{P},\mathcal{G}}$ from the collecting semantics through a further approximation, gives the sufficient conditions for $F^{\flat}_{\mathcal{P},\mathcal{G}}$ to approximate safely the collecting semantics, and analyses its worst case complexity. Section 4.4 derives $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ from $F^{\flat}_{\mathcal{P},\mathcal{G}}$ through one more approximation, and analyses its worst case complexity. In section 4.5, we show how the two generic abstract semantics can be specialised to infer groundness information. Section 4.6 discusses our solution to the problem of forward abstract interpretation of normal logic programs and outlines other plausible solutions and the work required to justify them. Section 4.7 concludes the chapter.

4.1 Operational Semantics

4.1.1 SLDNF

A normal program is executed by means of *SLDNF-resolution* (SLDNF in abbreviation). We shall assume that the left-to-right computation rule is used. We now briefly recall SLDNF. The renamed literals will be written in the form $L\rho$ where L is a literal in a clause or an initial goal and ρ a renaming substitution used to standardise-apart [2].

First consider *SLD-resolution* (SLD in short) for definite logic programs where each literal in the body of every clause is positive. For an initial goal and a definite program, SLD works by repeatedly *resolving* the current goal, initially the initial goal, with a clause in the program. In one resolution step, SLD nondeterministically selects a clause in the program, renames the clause so that it does not have any common variables with the current goal, and derives a new goal by replacing the leftmost literal in the current goal with the body of the renamed clause and applying to the resultant the most general unifier of the head of the renamed clause and the leftmost literal.

If SLD has derived an empty goal, it has successfully computed a *computed answer substitution* to the initial goal. The computed answer substitution is the restriction, to the variables in the initial goal, of the composition of the most general unifiers in the derivation from the initial goal to the empty goal.

With SLD, only positive information can be derived from a program. Now consider SLDNF. SLDNF uses the *negation as failure* rule to derive negative information. SLDNF deals with positive literals in the same way as SLD.

Suppose that the leftmost literal in the current goal is negative. SLDNF first recursively invokes itself with the leftmost literal as the initial goal of the recursive invocation. If the recursive invocation fails then SLDNF removes the leftmost literal from the current goal and continues with the resultant as the new current goal. Otherwise, according to the weaker safeness condition for the negation as failure rule [74], SLDNF either backtracks if the computed answer substitution returned by the recursive invocation is a renaming substitution or otherwise *flounders* [74].

Let the current goal be

$$\leftarrow (L_{(i_j,k)}\rho_j, L_{(i_j,k+1)}\rho_j, \cdots, L_{(i_j,m[i_j])}\rho_j, \cdots)\tau_{(j,k)} \tag{R1}$$

where $dom(\tau_{j,k}) \subseteq vars(L_{(i_j,k)}\rho_j, L_{(i_j,k+1)}\rho_j, \cdots)$ and ρ_j is the renaming substitution used to rename clause C_{i_j}. If $L_{(i_j,k)} \equiv \neg B_{(i_j,k)}$ then SLDNF recursively invokes itself with $\leftarrow B_{(i_j,k)}\tau_{(j,k)}$ as the initial goal of the recursive invocation.

Let $L_{(i_j,k)} \equiv B_{(i_j,k)}$. If there is a clause $C_{i_{j+1}} \equiv H_{i_{j+1}} \leftarrow L_{(i_{j+1},1)}, \cdots, L_{(i_{j+1},m[i_{j+1}])}$ in the program and a renaming substitution ρ_{j+1} such that

$$vars(C_{i_{j+1}}\rho_{j+1}) \cap vars((L_{(i_j,k)}\rho_j, L_{(i_j,k+1)}\rho_j, \cdots)\tau_{(j,k)}) = \emptyset \tag{4.1}$$

and $B_{(i_j,k)}\rho_j\tau_{(j,k)}$ and $H_{i_{j+1}}\rho_{j+1}$ unify, then the new current goal becomes

$$\leftarrow (L_{(i_{j+1},1)}\rho_{j+1}, \cdots, L_{(i_{j+1},m[i_{j+1}])}\rho_{j+1}, L_{(i_j,k+1)}\rho_j, \cdots)\tau_{(j+1,1)} \tag{R2}$$

where

$$\tau_{(j+1,1)} = \tau_{(j,k)} \circ \eta \tag{4.2}$$

and

$$\eta = mgu(H_{i_{j+1}}\rho_{j+1}, B_{(i_j,k)}\rho_j\tau_{(j,k)}) \tag{4.3}$$

Suppose that there is a sub-refutation of

$$\leftarrow (L_{(i_{j+1},1)}\rho_{j+1}, \cdots, L_{(i_{j+1},m[i_{j+1}])}\rho_{j+1})\tau_{(j+1,1)}$$

with the composition of the most general unifiers used in the sub-refutation being θ. Then the next current goal immediately after the sub-refutation is

$$\leftarrow (L_{(i_j,k+1)}\rho_j, \cdots, L_{(i_j,m[i_j])}\rho_j, \cdots)\tau_{(j,k+1)} \tag{R3}$$

where

$$\tau_{(j,k+1)} = \tau_{(j+1,1)} \circ \theta = \tau_{(j,k)} \circ \eta \circ \theta \tag{4.4}$$

4.1.2 VSLDNF

We now propose a variant of SLDNF (VSLDNF in abbreviation) as the operational semantics in order to facilitate the development of the collecting semantics. VSLDNF is equivalent to SLDNF in the sense that, given the same goal and the same program, VSLDNF reaches a program point iff SLDNF reaches the same program point, and the instantiation of the variables in the clause of the program point by VSLDNF is equivalent (modulo renaming) to that by SLDNF. We now formulate VSLDNF and then establish the equivalence between SLDNF and VSLDNF.

Let the current goal be

$$\leftarrow (L_{(i_j,k)}, L_{(i_j,k+1)}, \cdots, L_{(i_j,m[i_j])})\sigma_{(j,k)} \tag{R1'}$$

with $dom(\sigma_{(j,k)}) \subseteq \mathcal{V}_{i_j}$. If $L_{(i_j,k)} \equiv \neg B_{(i_j,k)}$ then VSLDNF recursively invokes itself with $\leftarrow B_{(i_j,k)}\sigma_{(j,k)}$ being the initial goal of the recursive invocation.

Let $L_{(i_j,k)} \equiv B_{(i_j,k)}$. The derivation is suspended and a sub-derivation is started as follows. If there is a clause $C_{i_{j+1}} \equiv H_{i_{j+1}} \leftarrow L_{(i_{j+1},1)}, \cdots, L_{(i_{j+1},m[i_{j+1}])}$ in the program and a renaming substitution ψ_{j+1} such that

$$\mathcal{V}_{i_{j+1}} \cap vars(L_{(i_j,k)}\sigma_{(j,k)}\psi_{j+1}) = \emptyset \tag{4.5}$$

and $B_{(i_j,k)}\sigma_{(j,k)}\psi_{j+1}$ and $H_{i_{j+1}}$ unify then the next current goal becomes

$$(\leftarrow L_{(i_{j+1},1)}, \cdots, L_{(i_{j+1},m[i_{j+1}])})\sigma_{(j+1,1)} \tag{R2'}$$

where

$$\sigma_{(j+1,1)} = mgu(B_{(i_j,k)}\sigma_{(j,k)}\psi_{j+1}, H_{i_{j+1}}) \tag{4.6}$$

We call the step of deriving a new goal from the current goal and a clause in the program *procedure-entry.*

Suppose that there is a sub-refutation of

$$(\leftarrow L_{(i_{j+1},1)}, \cdots, L_{(i_{j+1},m[i_{j+1}])})\sigma_{(j+1,1)}$$

and $\sigma_{(j+1,m[i_{j+1}]+1)}$ is the substitution immediately after the sub-refutation. Then the suspended derivation is resumed and the new current goal becomes

$$\leftarrow (L_{(i_j,k+1)}, \cdots, L_{(i_j,m[i_j])})\sigma_{(j,k+1)} \tag{R3'}$$

where

$$\sigma_{(j,k+1)} = \sigma_{(j,k)} \circ mgu(B_{(i_j,k)}\sigma_{(j,k)}, H_{i_{j+1}}\sigma_{(j+1,m[i_{j+1}]+1)}\phi_{j+1}) \tag{4.7}$$

and ϕ_{j+1} is a renaming substitution such that

$$vars(H_{i_{j+1}}\sigma_{(j+1,m[i_{j+1}]+1)}\phi_{j+1}) \cap vars(C_{i_j}\sigma_{j,k}) = \emptyset \tag{4.8}$$

We call the step to derive a new goal from a suspended goal and a completed sub-derivation *procedure-exit.*

Lemma 4.1.1 VSLDNF is equivalent to SLDNF in the sense that, given the same goal and the same program, VSLDNF reaches a program point iff SLDNF reaches the same program point, and the instantiation of the variables in the clause of the program point by VSLDNF is equivalent (modulo renaming) to that by SLDNF.

PROOF: See p. 169. ∎

Example 4.1.2 This example illustrates VSLDNF. Let the program be that in example 2.2.1 and $\leftarrow \mathit{diff}(X,[2,1],[3,1])$ be the initial goal. Let $\sigma_{(0,1)} = \emptyset$. VSLDNF begins with the following current goal.

$$\leftarrow \mathit{diff}(X,[2,1],[3,1])\sigma_{(0,1)} \tag{G0}$$

Let $\psi_1 = \{X/X_1\}$ and $C_{i_1} = C_1$. Then

$$\begin{aligned}\sigma_{(1,1)} &= mgu(\mathit{diff}(X,[2,1],[3,1])\sigma_{(0,1)}\psi_1, \mathit{diff}(X,L,K)) \\ &= \{X_1/X, L/[2,1], K/[3,1]\}\end{aligned}$$

VSLDNF suspends goal G0, performs a procedure-entry, and derives the following goal.

$$\leftarrow (member(X,L), \neg member(X,K))\sigma_{(1,1)} \tag{G1}$$

Let $\psi_2 = \{X/X_2\}$ and $C_{i_2} = C_3$. Then

$$\begin{aligned}\sigma_{(2,1)} &= mgu(member(X,L)\sigma_{(1,1)}\psi_2, member(X,[X|L])) \\ &= \{X_2/2, X/2, L/[1]\}\end{aligned}$$

VSLDNF suspends goal G1, performs a procedure-entry, and derives the following empty goal.

$$\Box\sigma_{(2,1)} \tag{G2}$$

VSLDNF performs a procedure-exit step and derives the following from G1.

$$\leftarrow \neg member(X,K))\sigma_{(1,2)} \tag{G3}$$

where, letting $\phi_2 = \emptyset$,

$$\begin{aligned}\sigma_{(1,2)} &= \sigma_{(1,1)} \circ mgu(member(X,L)\sigma_{(1,1)}, member(X,[X|L])\sigma_{(2,1)}\phi_2) \\ &= \{X_1/2, X/2, L/[2,1], K/[3,1]\}\end{aligned}$$

The leftmost literal of G3 is negative, VSLDNF invokes itself recursively with $\leftarrow member(X, K))\sigma_{(1,2)}$ and fails to refute it. So, by the negation as failure rule, VSLDNF derives the following goal with $\sigma_{(1,3)} = \sigma_{(1,2)}$.

$$\Box\sigma_{(1,3)} \tag{G4}$$

This finishes a sub-refutation of G1. VSLDNF performs a procedure-exit step and derives the following from G0.

$$\Box\sigma_{(0,2)} \tag{G5}$$

where, letting $\phi_1 = \emptyset$,

$$\begin{aligned} \sigma_{(0,2)} &= \sigma_{(0,1)} \circ mgu(diff(X, [2,1], [3,1])\sigma_{(0,1)}, diff(X, L, K)\sigma_{(1,3)}\phi_1) \\ &= \{X/2\} \end{aligned}$$

This finishes a refutation of G0. So, $\sigma_{(0,2)} = \{X/2\}$ is a computed answer substitution of $\leftarrow diff(X, [2,1], [3,1])$. ∎

4.1.3 Differences between VSLDNF and SLDNF

VSLDNF differs from SLDNF in several ways. Firstly, a goal in VSLDNF is a part of a clause or an initial goal, in particular, it is a tail of a clause or an initial goal. This helps to approximate VSLDNF with a transition system later in this work. Secondly, when VSLDNF derives a new goal from the current goal and a clause, it renames the leftmost literal in the goal instead of the clause. This is to ensure that the domain of the substitution that will be applied to the body of the clause contains variables in the clause instead of their renamed counterparts. Thirdly, when a sub-refutation is finished, an extra renaming and an extra unification are needed for VSLDNF to calculate the substitution immediately after the sub-refutation whilst these extra operations are not needed in SLDNF. Note that VSLDNF is only used in formulating the collecting semantics.

4.1.4 Program Graph

Let $p, q \in \mathcal{N}_{\mathcal{P},\mathcal{G}}$, and q be the most recent program point that VSLDNF has reached. There are several possibilities that VSLDNF will reach p next. In order to facilitate the presentation, we assume that VSLDNF starts at a dummy program point $(0,0) \notin \mathcal{N}_{\mathcal{P},\mathcal{G}}$ from where it can reach the entry points of initial goals by doing nothing. If p is an entry point of a clause then

VSLDNF may reach p immediately only by performing a procedure-entry either directly when L_q is positive or indirectly when L_q is negative. In this case, q cannot be the exit point of a clause or an initial goal. If p is not an entry point of a clause or an initial goal then VSLDNF may reach p immediately either by performing a procedure-exit or by applying the negation as failure rule. VSLDNF may reach p immediately by performing a procedure-exit only if L_{p^-} is positive, q is the exit point of a clause, and that program clause has been used to resolve with L_{p^-}. VSLDNF may reach p immediately by applying the negation as failure rule only if $(L_q \equiv \neg B_q) \wedge q = p^-$. Therefore, there are four ways that VSLDNF will reach p immediately after it has reached q. We use a graph $< \mathcal{N}^+_{\mathcal{P},\mathcal{G}}, \mathcal{E}_{\mathcal{P},\mathcal{G}} >$, called a program graph, to represent the relation among program points p, q that VSLDNF will possibly visit p immediately after it has visited q. The set $\mathcal{N}^+_{\mathcal{P},\mathcal{G}}$ of nodes in the program graph is $\mathcal{N}_{\mathcal{P},\mathcal{G}} \cup \{(0,0)\}$ and each edge $p \leftarrow\!\bullet\, q$ in $\mathcal{E}_{\mathcal{P},\mathcal{G}}$ in the program graph denotes that VSLDNF will possibly visit p immediately after it has visited q. Formally, $\mathcal{E}_{\mathcal{P},\mathcal{G}}$ is inductively defined as follows.

$$
\begin{array}{lcl}
\mathcal{E}_{\mathcal{P},\mathcal{G}} & \stackrel{def}{=} & \bigcup_{0 \leq \jmath \leq 3} \mathcal{E}^{\jmath}_{\mathcal{P},\mathcal{G}} \\
\mathcal{E}^{0}_{\mathcal{P},\mathcal{G}} & \stackrel{def}{=} & \{entry(k) \leftarrow\!\bullet (0,0) \mid k \in \aleph_G\} \\
\mathcal{E}^{1}_{\mathcal{P},\mathcal{G}} & \stackrel{def}{=} & \left\{ entry(i) \leftarrow\!\bullet\, q \;\middle|\; \begin{array}{ll} & q[2] \leq m[q[1]] \\ \wedge & i \in \aleph_C \\ \wedge & \exists \rho. \left(\begin{array}{ll} & \rho \textit{ is a renaming substitution} \\ \wedge & vars(B_q\rho) \cap vars(H_i) = \emptyset \\ \wedge & mgu(B_q\rho, H_i) \neq \textit{fail} \end{array} \right) \end{array} \right\} \\
\mathcal{E}^{2}_{\mathcal{P},\mathcal{G}} & \stackrel{def}{=} & \{p \leftarrow\!\bullet\, exit(i) \mid entry(i) \leftarrow\!\bullet\, p^- \in \mathcal{E}^{1}_{\mathcal{P},\mathcal{G}} \wedge L_{p^-} \equiv B_{p^-}\} \\
\mathcal{E}^{3}_{\mathcal{P},\mathcal{G}} & \stackrel{def}{=} & \{p \leftarrow\!\bullet\, p^- \mid L_{p^-} \equiv \neg B_{p^-}\}
\end{array}
$$

$\mathcal{E}^{0}_{\mathcal{P},\mathcal{G}}$ is the collection of edges from dummy program point $(0,0)$ to the entry points of initial goals. Edges in $\mathcal{E}^{1}_{\mathcal{P},\mathcal{G}}$ correspond to procedure-entry, edges in $\mathcal{E}^{2}_{\mathcal{P},\mathcal{G}}$ to procedure-exit, and edges in $\mathcal{E}^{3}_{\mathcal{P},\mathcal{G}}$ to the negation as failure rule. $\mathcal{E}^{\imath}_{\mathcal{P},\mathcal{G}} \cap \mathcal{E}^{\jmath}_{\mathcal{P},\mathcal{G}} = \emptyset$ for any $0 \leq \imath, \jmath \leq 3$ such that $\imath \neq \jmath$.

Example 4.1.3 Let $\mathcal{P}$ and $\mathcal{G}$ be those in example 2.2.1.

$5 \in \aleph_G$. Hence, $(5,1) \leftarrow\!\bullet (0,0) \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$. Let $\rho = \{X/X_0, L/L_0\}$. $mgu(B_{(1,1)}\rho, H_3) = \{X_0/X, L_0/[X|L]\} \neq \textit{fail}$. So, $(3,1) \leftarrow\!\bullet (1,1) \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$. Since $L_{(1,1)}$ is positive,

$(1,2) \leftarrow\!\!\bullet (3,1) \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$. Let $\delta = \{X/X_0, K/K_0\}$. $mgu(B_{(1,2)}\delta, H_3) = \{X_0/X, K_0/[X|L]\} \neq fail$. So, $(3,1) \leftarrow\!\!\bullet (1,2) \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$. Since $L_{(1,2)}$ is negative, $(1,3) \leftarrow\!\!\bullet (1,2) \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$.

There are 23 edges in $\mathcal{E}_{\mathcal{P},\mathcal{G}}$. ∎

4.2 Collecting Semantics

In this section, we present the fixed-point collecting semantics. The collecting semantics is the least fixed-point of function $F^{\sharp}_{\mathcal{P},\mathcal{G}}$ that is defined below. The least fixed-point of $F^{\sharp}_{\mathcal{P},\mathcal{G}}$ associates a set of substitutions with each edge $p \leftarrow\!\!\bullet q \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$.

In sub-section 4.2.1, we use a transition system to approximate VSLDNF. A state in this transition system corresponds to a goal in VSLDNF. The set of states derivable from a set of initial states by the transition system is then characterised as the least fixed-point $lfpF_{\mathcal{P},\mathcal{G}}$ of a function $F_{\mathcal{P},\mathcal{G}}$ mapping a set of states into another set of states. Therefore, the set of goals derivable from a set of initial goals by VSLDNF is approximated by $lfpF_{\mathcal{P},\mathcal{G}}$. In sub-section 4.2.2, we derive the fixed-point collecting semantic function $F^{\sharp}_{\mathcal{P},\mathcal{G}}$ from $F_{\mathcal{P},\mathcal{G}}$ and prove that $lfpF^{\sharp}_{\mathcal{P},\mathcal{G}}$ is a safe approximation of $lfpF_{\mathcal{P},\mathcal{G}}$.

Let A, B be atoms, and $\theta, \omega \in Sub$. Define

$$unify(A,\theta,B,\omega) \stackrel{def}{=} \left\{ \begin{array}{l} let\ \rho\ be\ a\ renaming\ substitution\ such\ that \\ \quad vars(A\theta\rho) \cap vars(B\omega) = \emptyset \\ in \\ \left\{ \begin{array}{ll} \omega \circ mgu(A\theta\rho, B\omega) & if\ mgu(A\theta\rho, B\omega) \neq fail \\ fail & otherwise \end{array} \right. \end{array} \right. \tag{4.9}$$

Although there are an infinite number of renaming substitutions ρ such that $vars(A\theta\rho) \cap vars(B\omega) = \emptyset$ in equation 4.9, only one renaming substitution must be considered when computing $unify(A,\theta,B,\omega)$ because $unify(A,\theta,B,\omega)$ for one renaming substitution is equivalent (modulo renaming) to $unify(A,\theta,B,\omega)$ for another renaming substitution according to lemma A.1.2 (p. 164).

4.2.1 Approximating VSLDNF by a Transition System

We now devise a transition system to approximate VSLDNF. A state in the transition system corresponds to a goal in VSLDNF. The transition system approximates VSLDNF in the sense that, if a goal is derivable from an initial goal by VSLDNF then the state corresponding to the goal is derivable from the state corresponding to the initial goal by the transition system while the reverse is not necessarily true.

A state in the transition system is a stack of stack items. The empty stack is denoted as $\$$. A stack item is of the form $\parallel p \leftarrow\!\bullet q, \theta \parallel$ where $p \leftarrow\!\bullet q \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$ and $\theta \in Sub$. The meaning of $\parallel p \leftarrow\!\bullet q, \theta \parallel$ is that the control is transferred from q to p with θ being the substitution at p. The set $\mathbf{S}^\sharp$ of all the possible stack items is therefore

$$\mathbf{S}^\sharp = \{\parallel p \leftarrow\!\bullet q, \theta \parallel \mid p \leftarrow\!\bullet q \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \wedge \theta \in Sub\}$$

The set $\mathbf{S}$ of all the possible stacks is the set of all the possible sequences of the stack items from $\mathbf{S}^\sharp$. $\mathbf{S}$ can be inductively defined as follows.

- $\$ \in \mathbf{S}$; and
- $\parallel p \leftarrow\!\bullet q, \theta \parallel \cdot S \in \mathbf{S}$ if $\parallel p \leftarrow\!\bullet q, \theta \parallel \in \mathbf{S}^\sharp \wedge S \in \mathbf{S}$.

Let $x_1 \in \mathbf{S}^\sharp, \ldots, x_n \in \mathbf{S}^\sharp$ and $S \in \mathbf{S}$. We sometimes write $x_1 \cdot \ldots \cdot x_n \cdot S$ as

$$\begin{array}{c} \underline{x_1} \\ \vdots \\ \underline{x_n} \\ S \end{array}$$

The set $\mathbf{S}_0 \subseteq \mathbf{S}$ of initial states is determined by the set of initial goals in VSLDNF.

$$\mathbf{S}_0 \stackrel{def}{=} \{\parallel p \leftarrow\!\bullet q, \theta \parallel \cdot \$ \mid \parallel p \leftarrow\!\bullet q, \theta \parallel \in \mathbf{S}^\sharp \wedge p \leftarrow\!\bullet q \in \mathcal{E}^0_{\mathcal{P},\mathcal{G}} \wedge \theta \in \Theta_{p[1]}\}$$

The set of final states is

$$\mathbf{S}_\infty \stackrel{def}{=} \{\parallel exit(k) \leftarrow\!\bullet q, \theta \parallel \cdot \$ \mid k \in \aleph_G \wedge \theta \in Sub \wedge exit(k) \leftarrow\!\bullet q \in \mathcal{E}_{\mathcal{P},\mathcal{G}}\}$$

The set of the descendant states of the initial states in $\mathbf{S}_0$ is obtained by applying the transition rules in figure 4.1. Let S and S' be two states; then $S \stackrel{\mathcal{P},\mathcal{G}}{\leadsto} S'$ denotes that the transition system changes from the state S to the state S'. Rule (0) says that every final state is stable. Rule (1a) corresponds to direct procedure-entry and rule (1b) to indirect procedure-entry. Rule (2) corresponds to procedure-exit and rule (3) deals with negative literals.

Rule (3) causes the inaccuracy of the transition system. When the transition system reaches a state corresponding to a goal with its leftmost literal being negative, the transition system may apply either rule (1b) or rule (3). Applying rule (1b), it will go to a state corresponding to a goal

Rule (0)	$If \quad k \in \aleph_G$ $then \quad \parallel exit(k)\leftarrow\!\bullet q, \theta \parallel \cdot \$ \overset{\mathcal{P},\mathcal{G}}{\rightsquigarrow} \parallel exit(k)\leftarrow\!\bullet q, \theta \parallel \cdot \$$
Rule (1a)	$If \quad \left(\begin{array}{c} L_q \equiv B_q \wedge entry(i)\leftarrow\!\bullet q \in \mathcal{E}^1_{\mathcal{P},\mathcal{G}} \\ \wedge \quad \theta = unify(B_q, \sigma, H_i, \epsilon) \neq fail \end{array} \right)$ $then \quad \parallel q\leftarrow\!\bullet u, \sigma \parallel \cdot S \overset{\mathcal{P},\mathcal{G}}{\rightsquigarrow} \parallel entry(i)\leftarrow\!\bullet q, \theta \parallel \cdot \parallel q\leftarrow\!\bullet u, \sigma \parallel \cdot S$
Rule (1b)	$If \quad \left(\begin{array}{c} L_q \equiv \neg B_q \wedge entry(i)\leftarrow\!\bullet q \in \mathcal{E}^1_{\mathcal{P},\mathcal{G}} \\ \wedge \quad \theta = unify(B_q, \sigma, H_i, \epsilon) \neq fail \end{array} \right)$ $then \quad \parallel q\leftarrow\!\bullet u, \sigma \parallel \cdot S \overset{\mathcal{P},\mathcal{G}}{\rightsquigarrow} \parallel entry(i)\leftarrow\!\bullet q, \theta \parallel \cdot \$$
Rule (2)	$If \quad \left(\begin{array}{c} p\leftarrow\!\bullet q \in \mathcal{E}^2_{\mathcal{P},\mathcal{G}} \\ \wedge \quad \theta = unify(H_{q[1]}, \sigma, B_{p^-}, \eta) \neq fail \end{array} \right)$ $then \quad \parallel q\leftarrow\!\bullet u, \sigma \parallel \cdot \parallel p^-\leftarrow\!\bullet v, \eta \parallel \cdot S \overset{\mathcal{P},\mathcal{G}}{\rightsquigarrow} \parallel p\leftarrow\!\bullet q, \theta \parallel \cdot S$
Rule (3)	$If \quad p\leftarrow\!\bullet q \in \mathcal{E}^3_{\mathcal{P},\mathcal{G}}$ $then \quad \parallel q\leftarrow\!\bullet u, \theta \parallel \cdot S \overset{\mathcal{P},\mathcal{G}}{\rightsquigarrow} \parallel p\leftarrow\!\bullet q, \theta \parallel \cdot S$

Figure 4.1: Transition Rules

after performing a procedure-entry indirectly. The application of rule (1b) is to enable information to be propagated forward and thus ensure that the transition system safely approximates VSLDNF. Applying rule (3), it will go to a state corresponding to the goal as if the recursive invocation of VSLDNF with the negative literal had failed while the recursive invocation may succeed in some cases. This results in a straightforward and least precise approximation of VSLDNF. This issue will be discussed later.

The set of descendant states of the initial states in $\mathbf{S}_0$ is therefore the least fixed-point of a function $F_{\mathcal{P},\mathcal{G}}$ that is defined as follows.

$$F_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \bigcup_{0 \leq \jmath \leq 3} F^{\jmath}_{\mathcal{P},\mathcal{G}}(X) \tag{4.10}$$

$$F^0_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \{\parallel p\leftarrow\!\bullet q, \theta \parallel \cdot \$ \mid p\leftarrow\!\bullet q \in \mathcal{E}^0_{\mathcal{P},\mathcal{G}} \wedge \theta \in \Theta_{p[1]}\} \tag{4.11}$$

$$F^1_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ \begin{array}{c|l} \dfrac{\dfrac{\parallel p\leftarrow\!\bullet q, \theta \parallel}{\parallel q\leftarrow\!\bullet u, \sigma \parallel}}{S} & \begin{array}{l} p\leftarrow\!\bullet q \in \mathcal{E}^1_{\mathcal{P},\mathcal{G}} \wedge L_q \equiv B_q \\ \wedge \quad \parallel q\leftarrow\!\bullet u, \sigma \parallel \cdot S \in X \\ \wedge \quad \theta = unify(B_q, \sigma, H_{p[1]}, \epsilon) \neq fail \end{array} \end{array} \right\} \tag{4.12}$$

$$\cup \quad \left\{ \| p{\leftarrow\!\bullet}q, \theta \| \cdot \$ \;\middle|\; \begin{array}{ll} & p{\leftarrow\!\bullet}q \in \mathcal{E}^1_{\mathcal{P},\mathcal{G}} \wedge L_q \equiv \neg B_q \\ \wedge & \| q{\leftarrow\!\bullet}u, \sigma \| \cdot S \in X \\ \wedge & \theta = \mathit{unify}(B_q, \sigma, H_{p[1]}, \epsilon) \neq \mathit{fail} \end{array} \right\}$$

$$F^2_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ \| p{\leftarrow\!\bullet}q, \theta \| \cdot S \;\middle|\; \begin{array}{ll} & p{\leftarrow\!\bullet}q \in \mathcal{E}^2_{\mathcal{P},\mathcal{G}} \\ \wedge & \dfrac{\| q{\leftarrow\!\bullet}u, \sigma \|}{\dfrac{\| p^{-}{\leftarrow\!\bullet}v, \eta \|}{S}} \in X \\ \wedge & \theta = \mathit{unify}(H_{q[1]}, \sigma, B_{p^-}, \eta) \neq \mathit{fail} \end{array} \right\} \tag{4.13}$$

$$F^3_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \{\| p{\leftarrow\!\bullet}q, \theta \| \cdot S \mid p{\leftarrow\!\bullet}q \in \mathcal{E}^3_{\mathcal{P},\mathcal{G}} \wedge \| q{\leftarrow\!\bullet}u, \theta \| \cdot S \in X\} \tag{4.14}$$

The domain $\mathbf{D}$ of $F_{\mathcal{P},\mathcal{G}}$ is $\wp(\mathbf{S})$. $< \mathbf{D}, \subseteq, \emptyset, \mathbf{S}, \cap, \cup >$ is a complete lattice and $F_{\mathcal{P},\mathcal{G}}$ is a monotonic function on $< \mathbf{D}, \subseteq >$.

Rule (0) of the transition system is not embodied in $F_{\mathcal{P},\mathcal{G}}$. Since $F_{\mathcal{P},\mathcal{G}}$ is a monotonic function on $< \mathbf{D}, \subseteq >$, we have $F_{\mathcal{P},\mathcal{G}} \uparrow k \subseteq F_{\mathcal{P},\mathcal{G}} \uparrow (k+1)$ for any $k \geq 0$. Therefore, any final state will be in $\mathit{lfp} F_{\mathcal{P},\mathcal{G}}$ if it is derivable from an initial state by the transition system.

4.2.2 Collecting Semantics

$\mathit{lfp} F_{\mathcal{P},\mathcal{G}}$ is a set of states. A state is a stack that corresponds to a goal and records information about how the goal is derived from an initial goal. The collecting semantics $\mathit{lfp} F^{\sharp}_{\mathcal{P},\mathcal{G}}$ first abstracts away the sequential relation between the stack items in a stack and then classifies the stack items according to the edges $p{\leftarrow\!\bullet}q$. $\mathit{lfp} F^{\sharp}_{\mathcal{P},\mathcal{G}}$ associates each $p{\leftarrow\!\bullet}q$ in $\mathcal{E}_{\mathcal{P},\mathcal{G}}$ with the set of substitutions θ such that $\| p{\leftarrow\!\bullet}q, \theta \| \in \mathbf{S}^{\sharp}$. Each edge $p{\leftarrow\!\bullet}q \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$ will be associated with a member from $\wp(Sub)$. $< \wp(Sub), \subseteq, \emptyset, Sub, \cap, \cup >$ is a complete lattice. Therefore, the domain $\mathbf{D}^{\sharp}$ of the collecting semantics is the Cartesian product of the same component domain $\wp(Sub)$ for as many times as the number of edges $p{\leftarrow\!\bullet}q$ in $\mathcal{E}_{\mathcal{P},\mathcal{G}}$. Let $X^{\sharp} \in \mathbf{D}^{\sharp}$. We use $X^{\sharp}_{p\leftarrow\bullet q}$ to denote the component in $X^{\sharp}$ that corresponds to edge $p{\leftarrow\!\bullet}q$. Let $X^{\sharp}, Y^{\sharp} \in \mathbf{D}^{\sharp}$. Define

$$\begin{array}{lcl} X^{\sharp} \sqsubseteq^{\sharp} Y^{\sharp} & \stackrel{def}{=} & \forall p{\leftarrow\!\bullet}q \in \mathcal{E}_{\mathcal{P},\mathcal{G}}.(X^{\sharp}_{p\leftarrow\bullet q} \subseteq Y^{\sharp}_{p\leftarrow\bullet q}) \\ {[X^{\sharp} \sqcap^{\sharp} Y^{\sharp}]}_{p\leftarrow\bullet q} & \stackrel{def}{=} & X^{\sharp}_{p\leftarrow\bullet q} \cap Y^{\sharp}_{p\leftarrow\bullet q} \\ {[X^{\sharp} \sqcup^{\sharp} Y^{\sharp}]}_{p\leftarrow\bullet q} & \stackrel{def}{=} & X^{\sharp}_{p\leftarrow\bullet q} \cup Y^{\sharp}_{p\leftarrow\bullet q} \\ \top^{\sharp}_{p\leftarrow\bullet q} & \stackrel{def}{=} & Sub \\ \bot^{\sharp}_{p\leftarrow\bullet q} & \stackrel{def}{=} & \emptyset \end{array}$$

$< \mathbf{D}^{\sharp}, \sqsubseteq^{\sharp}, \bot^{\sharp}, \top^{\sharp}, \sqcap^{\sharp}, \sqcup^{\sharp} >$ is a complete lattice according to lemma 2.1.6 (p. 16).

The approximation of a set of stacks by a vector of sets of substitutions is modelled by the following concretisation function $\gamma^\sharp \in \mathbf{D}^\sharp \mapsto \mathbf{D}$.

$$\gamma^\sharp(X^\sharp) = \{\| p_1 \leftarrow\!\bullet q_1, \theta_1 \| \cdots \| p_n \leftarrow\!\bullet q_n, \theta_n \| \cdot \$ \mid 0 \le n \wedge \forall 1 \le i \le n . p_i \leftarrow\!\bullet q_i \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \wedge \theta_i \in X^\sharp_{p_i \leftarrow\bullet q_i}\} \tag{4.15}$$

Lemma 4.2.1 $\gamma^\sharp$ is monotonic.

PROOF: Immediate from equation 4.15. ∎

Let A, B be atoms, and Θ, Ω be sets of substitutions. Define

$$unify^\sharp(A, \Theta, B, \Omega) \stackrel{def}{=} \{unify(A, \theta, B, \omega) \neq fail \mid \theta \in \Theta \wedge \omega \in \Omega\} \tag{4.16}$$

The fixed-point collecting semantics is defined in the following.

$$[F^\sharp_{\mathcal{P},\mathcal{G}}(X^\sharp)]_{p\leftarrow\bullet q} \stackrel{def}{=}$$

$$\Theta_{p[1]} \qquad if\ p\leftarrow\!\bullet q \in \mathcal{E}^0_{\mathcal{P},\mathcal{G}} \tag{4.17}$$

$$\bigcup\{unify^\sharp(B_q, X^\sharp_{q\leftarrow\bullet u}, H_{p[1]}, \{\epsilon\}) \mid q\leftarrow\!\bullet u \in \mathcal{E}_{\mathcal{P},\mathcal{G}}\} \qquad if\ p\leftarrow\!\bullet q \in \mathcal{E}^1_{\mathcal{P},\mathcal{G}} \tag{4.18}$$

$$\bigcup\left\{ unify^\sharp(H_{q[1]}, X^\sharp_{q\leftarrow\bullet u}, B_{p^-}, X^\sharp_{p^-\leftarrow\bullet v}) \;\middle|\; \begin{array}{ll} & p^-\leftarrow\!\bullet v \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \\ \wedge & q\leftarrow\!\bullet u \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \end{array} \right\} \qquad if\ p\leftarrow\!\bullet q \in \mathcal{E}^2_{\mathcal{P},\mathcal{G}} \tag{4.19}$$

$$\bigcup\{X^\sharp_{q\leftarrow\bullet u} \mid q\leftarrow\!\bullet u \in \mathcal{E}_{\mathcal{P},\mathcal{G}}\} \qquad if\ p\leftarrow\!\bullet q \in \mathcal{E}^3_{\mathcal{P},\mathcal{G}} \tag{4.20}$$

$F^\sharp_{\mathcal{P},\mathcal{G}}$ is a monotonic function on $< \mathbf{D}^\sharp, \sqsubseteq^\sharp >$.

Example 4.2.2 Let $\mathcal{P}$ and $\mathcal{G}$ be those in example 2.2.1. The collecting semantics is the least fixed-point of a system of 23 simultaneous recurrence equations. Each equation corresponds to an edge in $\mathcal{E}_{\mathcal{P},\mathcal{G}}$. The following four equations are examples of equations 4.17-4.20 respectively. Let $A = member(X, L)$ and $B = member(X, [X|L])$.

$$[F^\sharp_{\mathcal{P},\mathcal{G}}(X^\sharp)]_{(5,1)\leftarrow\bullet(0,0)} = \Theta_5$$

$$[F^\sharp_{\mathcal{P},\mathcal{G}}(X^\sharp)]_{(3,1)\leftarrow\bullet(1,1)} =$$
$$unify^\sharp(A, X^\sharp_{(1,1)\leftarrow\bullet(5,1)}, B, \{\epsilon\})$$

$$[F^\sharp_{\mathcal{P},\mathcal{G}}(X^\sharp)]_{(1,2)\leftarrow\bullet(3,1)} =$$

$$
\begin{array}{rl}
& unify^\sharp(B, X^\sharp_{(3,1)\leftarrow\bullet(1,1)}, A, X^\sharp_{(1,1)\leftarrow\bullet(5,1)}) \\
\cup & unify^\sharp(B, X^\sharp_{(3,1)\leftarrow\bullet(1,2)}, A, X^\sharp_{(1,1)\leftarrow\bullet(5,1)}) \\
\cup & unify^\sharp(B, X^\sharp_{(3,1)\leftarrow\bullet(2,1)}, A, X^\sharp_{(1,1)\leftarrow\bullet(5,1)}) \\
\cup & unify^\sharp(B, X^\sharp_{(3,1)\leftarrow\bullet(2,2)}, A, X^\sharp_{(1,1)\leftarrow\bullet(5,1)}) \\
\cup & unify^\sharp(B, X^\sharp_{(3,1)\leftarrow\bullet(4,1)}, A, X^\sharp_{(1,1)\leftarrow\bullet(5,1)})
\end{array}
$$

$$
[F^\sharp_{\mathcal{P},\mathcal{G}}(X^\sharp)]_{(1,3)\leftarrow\bullet(1,2)} = X^\sharp_{(1,2)\leftarrow\bullet(3,1)} \cup X^\sharp_{(1,2)\leftarrow\bullet(4,2)}
$$

∎

Lemma 4.2.3 $lfp F_{\mathcal{P},\mathcal{G}} \subseteq \gamma^\sharp(lfp F^\sharp_{\mathcal{P},\mathcal{G}})$.

PROOF: See p. 171. ∎

4.3 The First Generic Abstract Semantics $F^\flat_{\mathcal{P},\mathcal{G}}$

The collecting semantics is a safe approximation of the operational semantics and can be used as a basis for program analysis because any safe approximation of this collecting semantics is a safe approximation of the operational semantics. $[lfp F^\sharp_{\mathcal{P},\mathcal{G}}]_{p\leftarrow\bullet q}$ contains all the substitutions whenever the control of execution is transferred from program point q to program point p. $[lfp F^\sharp_{\mathcal{P},\mathcal{G}}]_{p\leftarrow\bullet q}$ is usually a set of an infinite number of substitutions and is therefore not computable in finite time. In order to obtain useful information about the possible substitutions when the control of execution is transferred from program point q to program point p, further abstraction is needed. In this section, we derive $F^\flat_{\mathcal{P},\mathcal{G}}$ from the collecting semantics.

4.3.1 Abstract Domains

The collecting semantics $lfp F^\sharp_{\mathcal{P},\mathcal{G}}$ associates with each edge $p\leftarrow\bullet q$ a set of substitutions which is a superset of the set of the substitutions whenever the control of execution is transferred from q to p.

When the program is analysed by means of abstract interpretation, the set of substitutions associated with $p\leftarrow\bullet q$ is approximated by an abstract substitution associated with $p\leftarrow\bullet q$. For the edge $p\leftarrow\bullet q$, only values about the variables in $\mathcal{V}_{p[1]}$ are of interest and, for the edge $p'\leftarrow\bullet q'$, only values for the variables in $\mathcal{V}_{p'[1]}$ are of interest. The abstract substitutions for $p\leftarrow\bullet q$ and $p'\leftarrow\bullet q'$ are from different domains when $p'[1] \neq p[1]$. We will simply call a domain for abstract substitutions an abstract domain. We find it convenient to have abstract domains parameterised by finite sets of variables instead of having a single abstract domain for all abstract substitutions associated with different edges or by constructing abstract domains for different edges in different ways. Let $Sub^\flat[\![\mathcal{V}]\!]$ denote the domain for abstract substitutions for describing values of variables in $\mathcal{V}$. Then $[lfpF^\sharp_{\mathcal{P},\mathcal{G}}]_{p\leftarrow\bullet q}$ is represented by a member of $Sub^\flat[\![\mathcal{V}_{p[1]}]\!]$. We require that, for any finite $\mathcal{V} \subseteq \mathcal{VAR}$,

C1: $< Sub^\flat[\![\mathcal{V}]\!], \sqsubseteq[\![\mathcal{V}]\!], \perp[\![\mathcal{V}]\!], \top[\![\mathcal{V}]\!], \sqcap[\![\mathcal{V}]\!], \sqcup[\![\mathcal{V}]\!] >$ is a complete lattice where $\sqsubseteq[\![\mathcal{V}]\!]$ is a partial order on $Sub^\flat[\![\mathcal{V}]\!]$, $\perp[\![\mathcal{V}]\!]$ the infimum, $\top[\![\mathcal{V}]\!]$ the supremum, $\sqcap[\![\mathcal{V}]\!]$ the greatest lower bound operator and $\sqcup[\![\mathcal{V}]\!]$ the least upper bound operator; and

C2: there is a monotonic concretisation function $\gamma[\![\mathcal{V}]\!] \in Sub^\flat[\![\mathcal{V}]\!] \mapsto \wp(Sub)$.

The domain $\mathbf{D}^\flat$ of $F^\flat_{\mathcal{P},\mathcal{G}}$ is constructed in the same manner as the domain $\mathbf{D}^\sharp$ of $F^\sharp_{\mathcal{P},\mathcal{G}}$ is constructed. Each member $X^\flat$ in $\mathbf{D}^\flat$ is a vector that is indexed by edges $p\leftarrow\bullet q$ in $\mathcal{E}_{\mathcal{P},\mathcal{G}}$. $X^\flat_{p\leftarrow\bullet q}$ is an element from $Sub^\flat[\![\mathcal{V}_{p[1]}]\!]$. Let $X^\flat \in \mathbf{D}^\flat$ and $Y^\flat \in \mathbf{D}^\flat$. Define

$$
\begin{array}{lcl}
X^\flat \sqsubseteq^\flat Y^\flat & \stackrel{def}{=} & \forall p\leftarrow\bullet q \in \mathcal{E}_{\mathcal{P},\mathcal{G}}.(X^\flat_{p\leftarrow\bullet q} \ \sqsubseteq[\![\mathcal{V}_{p[1]}]\!] \ Y^\flat_{p\leftarrow\bullet q}) \\
[X^\flat \sqcap^\flat Y^\flat]_{p\leftarrow\bullet q} & \stackrel{def}{=} & X^\flat_{p\leftarrow\bullet q} \ \sqcap[\![\mathcal{V}_{p[1]}]\!] \ Y^\flat_{p\leftarrow\bullet q} \\
[X^\flat \sqcup^\flat Y^\flat]_{p\leftarrow\bullet q} & \stackrel{def}{=} & X^\flat_{p\leftarrow\bullet q} \ \sqcup[\![\mathcal{V}_{p[1]}]\!] \ Y^\flat_{p\leftarrow\bullet q} \\
\top^\flat_{p\leftarrow\bullet q} & \stackrel{def}{=} & \top[\![\mathcal{V}_{p[1]}]\!] \\
\perp^\flat_{p\leftarrow\bullet q} & \stackrel{def}{=} & \perp[\![\mathcal{V}_{p[1]}]\!]
\end{array}
$$

$< \mathbf{D}^\flat, \sqsubseteq^\flat, \perp^\flat, \top^\flat, \sqcap^\flat, \sqcup^\flat >$ is a complete lattice according to lemma 2.1.6 (p. 16).

The concretisation function $\gamma^\flat$ is defined in terms of $\gamma[\![\mathcal{V}_i]\!]$. For all $p\leftarrow\bullet q \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$, and all $X^\flat \in \mathbf{D}^\flat$,

$$[\gamma^\flat(X^\flat)]_{p\leftarrow\bullet q} = \gamma[\![\mathcal{V}_{p[1]}]\!]({X^\flat}_{p\leftarrow\bullet q}) \tag{4.21}$$

Lemma 4.3.1 $\gamma^\flat$ is monotonic.

PROOF: Immediate from equation 4.21 and condition C2. ∎

4.3.2 The First Generic Abstract Semantics $F^\flat_{\mathcal{P},\mathcal{G}}$

$F^\flat_{\mathcal{P},\mathcal{G}}$ is derived from $F^\sharp_{\mathcal{P},\mathcal{G}}$ as follows. A set $\Theta \in \wp(Sub)$ of substitutions is replaced by an abstract substitution $\theta^\flat$ in $Sub^\flat[\![\mathcal{V}]\!]$ where $\mathcal{V}$ is a set of variables whose values are of interest, the function $unify^\sharp$ operating on sets of concrete substitutions is replaced by $unify^\flat$ operating on abstract substitutions and $\cup$ in the definition of $[lfpF^\sharp_{\mathcal{P},\mathcal{G}}]_{p\leftarrow\bullet q}$ is replaced by $\sqcup[\![\mathcal{V}_{p[1]}]\!]$. Let $\theta^\flat_k \in Sub^\flat[\![\mathcal{V}_k]\!]$ be the least abstract substitution such that $\Theta_k \subseteq \gamma[\![\mathcal{V}_k]\!](\theta^\flat_k)$ for each $k \in \aleph_G$. Note that $\theta^\flat_k$ instead of Θ_k is given before the program is analysed. Therefore, $\mathcal{G}$ can be equivalently written as $\{< G_k, \theta^\flat_k > \mid k \in \aleph_G\}$. Let $\epsilon^\flat[\![\mathcal{V}_i]\!] \in Sub^\flat[\![\mathcal{V}_i]\!]$ be the least abstract substitution such that $\epsilon \in \gamma[\![\mathcal{V}_i]\!](\epsilon^\flat[\![\mathcal{V}_i]\!])$ for each $i \in \aleph_C$. $F^\flat_{\mathcal{P},\mathcal{G}}$ is defined as follows.

$$
\begin{aligned}
&[F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{p\leftarrow\!\bullet q} \stackrel{def}{=} \\
&\quad \theta^\flat_{p[1]} && if\ p\leftarrow\!\bullet q \in \mathcal{E}^0_{\mathcal{P},\mathcal{G}} && (4.22)\\
&\quad \sqcup[\![\mathcal{V}_{p[1]}]\!]\{unify^\flat(B_q, X^\flat_{q\leftarrow\bullet u}, H_{p[1]}, \epsilon^\flat[\![\mathcal{V}_{p[1]}]\!]) | q\leftarrow\!\bullet u \in \mathcal{E}_{\mathcal{P},\mathcal{G}}\} && && (4.23)\\
&\quad && if\ p\leftarrow\!\bullet q \in \mathcal{E}^1_{\mathcal{P},\mathcal{G}} \\
&\quad \sqcup[\![\mathcal{V}_{p[1]}]\!]\left\{ unify^\flat(H_{q[1]}, X^\flat_{q\leftarrow\bullet u}, B_{p^-}, X^\flat_{p^-\leftarrow\bullet v}) \left| \begin{array}{l} p^-\leftarrow\!\bullet v \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \\ \wedge\ q\leftarrow\!\bullet u \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \end{array} \right. \right\} && && (4.24)\\
&\quad && if\ p\leftarrow\!\bullet q \in \mathcal{E}^2_{\mathcal{P},\mathcal{G}} \\
&\quad \sqcup[\![\mathcal{V}_{p[1]}]\!]\{X^\flat_{q\leftarrow\bullet u} | q\leftarrow\!\bullet u \in \mathcal{E}_{\mathcal{P},\mathcal{G}}\} && if\ p\leftarrow\!\bullet q \in \mathcal{E}^3_{\mathcal{P},\mathcal{G}} && (4.25)
\end{aligned}
$$

Example 4.3.2 Let $\mathcal{P}$ and $\mathcal{G}$ be those in example 2.2.1. $F^\flat_{\mathcal{P},\mathcal{G}}$ is a system of 23 simultaneous recurrence equations. The following four equations correspond to the four equations in example 4.2.2 respectively. Let $A = member(X, L)$ and $B = member(X, [X|L])$.

$$
\begin{aligned}
&[F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{(5,1)\leftarrow\bullet(0,0)} = \theta^\flat_5 \\
&[F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{(3,1)\leftarrow\bullet(1,1)} = \\
&\qquad unify^\flat(A, X^\flat_{(1,1)\leftarrow\bullet(5,1)}, B, \epsilon^\flat[\![\mathcal{V}_3]\!]) \\
&[F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{(1,2)\leftarrow\bullet(3,1)} = \\
&\qquad unify^\flat(B, X^\flat_{(3,1)\leftarrow\bullet(1,1)}, A, X^\flat_{(1,1)\leftarrow\bullet(5,1)}) \\
&\quad \sqcup[\![\{X, L, K\}]\!] \quad unify^\flat(B, X^\flat_{(3,1)\leftarrow\bullet(1,2)}, A, X^\flat_{(1,1)\leftarrow\bullet(5,1)}) \\
&\quad \sqcup[\![\{X, L, K\}]\!] \quad unify^\flat(B, X^\flat_{(3,1)\leftarrow\bullet(2,1)}, A, X^\flat_{(1,1)\leftarrow\bullet(5,1)}) \\
&\quad \sqcup[\![\{X, L, K\}]\!] \quad unify^\flat(B, X^\flat_{(3,1)\leftarrow\bullet(2,2)}, A, X^\flat_{(1,1)\leftarrow\bullet(5,1)}) \\
&\quad \sqcup[\![\{X, L, K\}]\!] \quad unify^\flat(B, X^\flat_{(3,1)\leftarrow\bullet(4,1)}, A, X^\flat_{(1,1)\leftarrow\bullet(5,1)}) \\
&[F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{(1,3)\leftarrow\bullet(1,2)} = X^\flat_{(1,2)\leftarrow\bullet(3,1)} \ \sqcup[\![\{X, L, K\}]\!] \ X^\flat_{(1,2)\leftarrow\bullet(4,2)}
\end{aligned}
$$

∎

4.3.3 Safeness

Theorem 4.3.3 $lfp F^\sharp_{\mathcal{P},\mathcal{G}} \sqsubseteq^\sharp \gamma^\flat(lfp F^\flat_{\mathcal{P},\mathcal{G}})$ if

C3: $\epsilon \in \gamma[\![\mathcal{V}]\!](\epsilon^\flat[\![\mathcal{V}]\!])$, and

C4: $\mathit{unify}^\sharp(A, \gamma[\![\mathcal{U}]\!](\theta^\flat), B, \gamma[\![\mathcal{V}]\!](\sigma^\flat)) \subseteq \gamma[\![\mathcal{V}]\!] \cdot \mathit{unify}^\flat(A, \theta^\flat, B, \sigma^\flat)$ for any finite $\mathcal{U}, \mathcal{V} \subseteq \mathcal{VAR}$, any $\theta^\flat \in \mathit{Sub}^\flat[\![\mathcal{U}]\!]$, any $\sigma^\flat \in \mathit{Sub}^\flat[\![\mathcal{V}]\!]$, and any atoms A and B such that $\mathit{vars}(A) \subseteq \mathcal{U}$ and $\mathit{vars}(B) \subseteq \mathcal{V}$.

PROOF: See p. 172. ∎

4.3.4 Complexity of $F^\flat_{\mathcal{P},\mathcal{G}}$

The cost of performing an analysis by computing $\mathit{lfp}F^\flat_{\mathcal{P},\mathcal{G}}$ is affected by the characteristics of $\mathcal{P}$ and $\mathcal{G}$, the abstract domain for the analysis, the abstract substitutions which approximate Θ_i where $i \in \aleph_G$, and the algorithm for the least fixed-point computation. Using O'Keefe's algorithm for the least fixed-point computation [104], the worst case cost of computing $\mathit{lfp}F^\flat_{\mathcal{P},\mathcal{G}}$ is proportional to the product of the number of operations in $F^\flat_{\mathcal{P},\mathcal{G}}$ and the maximum height dmax of the Hasse diagram for $\mathit{Sub}^\flat[\![\mathcal{V}_i]\!]$ for $i \in \aleph$. Since in most cases $\sqcup[\![\mathcal{V}]\!]$ is much less costly than $\mathit{unify}^\flat$, we measure the worst case number of occurrences of $\mathit{unify}^\flat$ in $F^\flat_{\mathcal{P},\mathcal{G}}$.

For any set S, we define $\#S$ to be the cardinality of S. Let

$$\mathcal{N}^\jmath_{\mathcal{P},\mathcal{G}} \stackrel{def}{=} \{p \in \mathcal{N}_{\mathcal{P},\mathcal{G}} \mid \exists q. p \leftarrow\!\bullet q \in \mathcal{E}^\jmath_{\mathcal{P},\mathcal{G}}\}$$

for $0 \leq \jmath \leq 3$. $\mathcal{E}^\jmath_{\mathcal{P},\mathcal{G}}$ is the set of edges whose end points lie in $\mathcal{N}^\jmath_{\mathcal{P},\mathcal{G}}$. Let pmax be the maximum number of predecessors that a program point has. By equations 4.22 and 4.25, $\mathit{unify}^\flat$ does not occur in equations for edges in $\mathcal{E}^0_{\mathcal{P},\mathcal{G}} \cup \mathcal{E}^3_{\mathcal{P},\mathcal{G}}$. By equation 4.23, $\mathit{unify}^\flat$ occurs at most pmax times in an equation for an edge in $\mathcal{E}^1_{\mathcal{P},\mathcal{G}}$. So, $\mathit{unify}^\flat$ occurs at most $\#\mathcal{E}^1_{\mathcal{P},\mathcal{G}} \times \mathit{pmax}$ times in equations for edges in $\mathcal{E}^1_{\mathcal{P},\mathcal{G}}$. By equation 4.24, $\mathit{unify}^\flat$ occurs at most pmax^2 times in an equation for an edge in $\mathcal{E}^2_{\mathcal{P},\mathcal{G}}$. So, $\mathit{unify}^\flat$ occurs at most $\#\mathcal{E}^2_{\mathcal{P},\mathcal{G}} \times \mathit{pmax}^2$ times in equations for edges in $\mathcal{E}^2_{\mathcal{P},\mathcal{G}}$. Since $\#\mathcal{E}^2_{\mathcal{P},\mathcal{G}} \leq \#\mathcal{E}^1_{\mathcal{P},\mathcal{G}}$ and $\#\mathcal{E}^1_{\mathcal{P},\mathcal{G}} \leq \#\aleph_C \times \mathit{pmax}$, the worst case number of occurrences of $\mathit{unify}^\flat$ in $F^\flat_{\mathcal{P},\mathcal{G}}$ is $O(\#\aleph_C \times \mathit{pmax}^3)$. Therefore, the worst case cost of computing $\mathit{lfp}F^\flat_{\mathcal{P},\mathcal{G}}$ is

$$O(\mathit{dmax} \times \#\aleph_C \times \mathit{pmax}^3)$$

4.4 The Second Generic Abstract Semantics $F^\diamond_{\mathcal{P},\mathcal{G}}$

A further abstraction may be made of $F^\flat_{\mathcal{P},\mathcal{G}}$ so as to reduce the complexity of program analyses. $\mathit{lfp}F^\flat_{\mathcal{P},\mathcal{G}}$ is a vector indexed by edges in $\mathcal{E}_{\mathcal{P},\mathcal{G}}$. $\mathit{lfp}F^\flat_{\mathcal{P},\mathcal{G}}$ associates each program point with several abstract substitutions, each abstract substitution for one edge ending at the program point. However, there are some applications where such fine analyses are not beneficial with respect to their costs and one abstract substitution for each program point is a better choice. $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ fulfills such purposes and is derived from $F^{\flat}_{\mathcal{P},\mathcal{G}}$ by one further approximation.

The domain of $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ is $\mathbf{D}^{\diamond}$ and each $X^{\diamond} \in \mathbf{D}^{\diamond}$ is a vector indexed by program points. $\mathbf{D}^{\diamond}$ is constructed from $Sub^{\flat}[\![\mathcal{V}]\!]$ in a similar manner as is $\mathbf{D}^{\flat}$. Let $X^{\diamond}, Y^{\diamond} \in \mathbf{D}^{\diamond}$, $X^{\diamond}_p \in Sub^{\flat}[\![\mathcal{V}_{p[1]}]\!]$ and define

$$
\begin{aligned}
X^{\diamond} \sqsubseteq^{\diamond} Y^{\diamond} &\stackrel{def}{=} \forall p \in \mathcal{N}_{\mathcal{P},\mathcal{G}}.(X^{\diamond}_p \sqsubseteq[\![\mathcal{V}_{p[1]}]\!]\, Y^{\diamond}_p) && (4.26)\\
[X^{\diamond} \sqcap^{\diamond} Y^{\diamond}]_p &\stackrel{def}{=} X^{\diamond}_p \sqcap[\![\mathcal{V}_{p[1]}]\!]\, Y^{\diamond}_p && (4.27)\\
[X^{\diamond} \sqcup^{\diamond} Y^{\diamond}]_p &\stackrel{def}{=} X^{\diamond}_p \sqcup[\![\mathcal{V}_{p[1]}]\!]\, Y^{\diamond}_p && (4.28)\\
\top^{\diamond}_p &\stackrel{def}{=} \top[\![\mathcal{V}_{p[1]}]\!] && (4.29)\\
\bot^{\diamond}_p &\stackrel{def}{=} \bot[\![\mathcal{V}_{p[1]}]\!] && (4.30)
\end{aligned}
$$

The abstraction, through collapsing the abstract substitutions associated with all the edges ending at a common program point, is characterised by the following concretisation function.

$$[\gamma^{\diamond}(X^{\diamond})]_{p \leftarrow\!\bullet q} \stackrel{def}{=} X^{\diamond}_p \qquad (4.31)$$

It follows immediately from equation 4.31 that $\gamma^{\diamond}$ is a monotonic function from $< \mathbf{D}^{\diamond}, \sqsubseteq^{\diamond} >$ to $< \mathbf{D}^{\flat}, \sqsubseteq^{\flat} >$. We now construct a monotonic function $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ on $< \mathbf{D}^{\diamond}, \sqsubseteq^{\diamond} >$ such that $F^{\flat}_{\mathcal{P},\mathcal{G}} \cdot \gamma^{\diamond}(X^{\diamond}) \sqsubseteq^{\flat} \gamma^{\diamond} \cdot F^{\diamond}_{\mathcal{P},\mathcal{G}}(X^{\diamond})$ for every $X^{\diamond} \in \mathbf{D}^{\diamond}$.

$$
\begin{aligned}
&[F^{\diamond}_{\mathcal{P},\mathcal{G}}(X^{\diamond})]_p \stackrel{def}{=}\\
&\quad \theta^{\flat}_{p[1]} && if\ p \in \mathcal{N}^0_{\mathcal{P},\mathcal{G}} && (4.32)\\
&\quad \sqcup[\![\mathcal{V}_{p[1]}]\!]\{unify^{\flat}(B_q, X^{\diamond}_q, H_{p[1]}, \epsilon^{\flat}[\![\mathcal{V}_{p[1]}]\!]) \mid p \leftarrow\!\bullet q \in \mathcal{E}^1_{\mathcal{P},\mathcal{G}}\} && && (4.33)\\
&\quad && if\ p \in \mathcal{N}^1_{\mathcal{P},\mathcal{G}}\\
&\quad \sqcup[\![\mathcal{V}_{p[1]}]\!]\{unify^{\flat}(H_{q[1]}, X^{\diamond}_q, B_{p\text{-}}, X^{\diamond}_{p\text{-}}) \mid p \leftarrow\!\bullet q \in \mathcal{E}^2_{\mathcal{P},\mathcal{G}}\} && && (4.34)\\
&\quad && if\ p \in \mathcal{N}^2_{\mathcal{P},\mathcal{G}}\\
&\quad X^{\diamond}_{p\text{-}} && if\ p \in \mathcal{N}^3_{\mathcal{P},\mathcal{G}} && (4.35)
\end{aligned}
$$

Theorem 4.4.1 *$lfp F^{\flat}_{\mathcal{P},\mathcal{G}} \sqsubseteq^{\flat} \gamma^{\diamond}(lfp F^{\diamond}_{\mathcal{P},\mathcal{G}})$.*

PROOF: See p. 173. ∎

$lfp F^{\diamond}_{\mathcal{P},\mathcal{G}}$ associates each program point with an abstract substitution. The simultaneous recurrence equations for $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ are simpler than those for $F^{\flat}_{\mathcal{P},\mathcal{G}}$ and hence the computation of $lfp F^{\diamond}_{\mathcal{P},\mathcal{G}}$

is less costly than that of $lfpF^{\flat}_{\mathcal{P},\mathcal{G}}$. $lfpF^{\diamond}_{\mathcal{P},\mathcal{G}}$ is a generalisation of Nilsson's abstract semantics for definite logic programs [101, 102] onto normal logic programs. Specifically, if $\mathcal{P}$ does not have negative literals then $lfpF^{\diamond}_{\mathcal{P},\mathcal{G}}$ is equal to that in [101, 102].

Each program point in $\mathcal{N}_{\mathcal{P},\mathcal{G}}$ corresponds to an equation of $F^{\diamond}_{\mathcal{P},\mathcal{G}}$. By equations 4.32 and 4.35, $unify^{\flat}$ does not occur in equations for points in $\mathcal{N}^{0}_{\mathcal{P},\mathcal{G}} \cup \mathcal{N}^{3}_{\mathcal{P},\mathcal{G}}$. $unify^{\flat}$ occurs at most $pmax$ times in an equation for a point in $\mathcal{N}^{1}_{\mathcal{P},\mathcal{G}} \cup \mathcal{N}^{2}_{\mathcal{P},\mathcal{G}}$ according to equations 4.33 and 4.34. The worst case number of occurrences of $unify^{\flat}$ in $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ is $O(\#\mathcal{N}_{\mathcal{P},\mathcal{G}} \times pmax)$ since $\#\mathcal{N}^{1}_{\mathcal{P},\mathcal{G}} + \#\mathcal{N}^{2}_{\mathcal{P},\mathcal{G}} \leq \#\mathcal{N}_{\mathcal{P},\mathcal{G}}$. Therefore, the worst case cost of computing $lfpF^{\diamond}_{\mathcal{P},\mathcal{G}}$ is

$$O(dmax \times \#\mathcal{N}_{\mathcal{P},\mathcal{G}} \times pmax)$$

$F^{\diamond}_{\mathcal{P},\mathcal{G}}$ may also be used to obtain abstract substitutions for edges in $\mathcal{E}_{\mathcal{P},\mathcal{G}}$. We prefer $F^{\flat}_{\mathcal{P},\mathcal{G}}$ to $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ for such analyses since $F^{\flat}_{\mathcal{P},\mathcal{G}}$ is easier to specialise than $F^{\diamond}_{\mathcal{P},\mathcal{G}}$. In order to specialise $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ for such an analysis, in addition to the work required to specialise $F^{\flat}_{\mathcal{P},\mathcal{G}}$ for the same analysis, one needs to

- keep information about program points in abstract substitutions, where an abstract substitution for $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ is a set of pairs of a program point and an abstract substitution for $F^{\flat}_{\mathcal{P},\mathcal{G}}$, and
- replace $unify^{\flat}(A, \theta^{\flat}, B, \sigma^{\flat})$ with $unify^{\diamond}(A, \theta^{\diamond}, B, \sigma^{\diamond})$ which, for each member of $\theta^{\diamond} \times \sigma^{\diamond}$, discards point information and calls $unify^{\flat}$.

This amounts to requiring the person who specialises $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ for such an analysis to undo approximation $\gamma^{\diamond}$. A call to $unify^{\diamond}$ may cause $unify^{\flat}$ to be called as many times as $pmax^2$. So, the worst case complexity of $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ is no less than that of $F^{\flat}_{\mathcal{P},\mathcal{G}}$ for such an analysis.

We have so far developed generic abstract semantics $F^{\flat}_{\mathcal{P},\mathcal{G}}$ and $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ for forward analyses of normal logic programs. $lfpF^{\flat}_{\mathcal{P},\mathcal{G}}$ obtains an abstract substitution for each edge in $\mathcal{E}_{\mathcal{P},\mathcal{G}}$ while $lfpF^{\diamond}_{\mathcal{P},\mathcal{G}}$ obtains an abstract substitution for each program point in $\mathcal{N}_{\mathcal{P},\mathcal{G}}$. In order to specialise either of these generic abstract semantics to perform an analysis, it is sufficient to find a domain $Sub^{\flat}[\![\mathcal{V}]\!]$ of abstract substitutions and a concreterisation function $\gamma[\![\mathcal{V}]\!]$ such that conditions C1-C2 are satisfied, to define the abstract identity substitution $\epsilon^{\flat}[\![\mathcal{V}]\!]$ on $\mathcal{V}$ (see [9]) so that it satisfies C3, and to develop an algorithm for $unify^{\flat}$ that satisfies C4. An algorithm for $unify^{\flat}$ is often called an abstract unification algorithm for the analysis.

4.5 Example

In this section, we illustrate how $F^\flat_{\mathcal{P},\mathcal{G}}$ and $F^\diamond_{\mathcal{P},\mathcal{G}}$ can be specialised to perform an analysis through groundness analysis - a simplified version of mode analysis.

4.5.1 Abstract Domain for Groundness Analysis

Given a substitution, we can deduce which variables will be instantiated to a ground term by that substitution and which will not. In groundness analysis, we are interested in knowing which variables will be definitely instantiated to ground terms. An abstract substitution is a set of variables. Each variable is ground under any substitution described by the abstract substitution. So $Sub^\flat[\![\mathcal{V}]\!] = \wp(\mathcal{V})$. The partial order on $Sub^\flat[\![\mathcal{V}]\!]$ that is induced by $\subseteq$ on $\wp(Sub)$ is its dual $\supseteq$. $< \wp(\mathcal{V}), \supseteq, \mathcal{V}, \emptyset, \cup, \cap >$ is a complete lattice.

The approximation of a set of substitutions by a set of variables is modelled by the concretisation function $\gamma[\![\mathcal{V}]\!] \in \wp(\mathcal{V}) \mapsto \wp(Sub)$ defined as follows.

$$\gamma[\![\mathcal{V}]\!](\theta^\flat) \stackrel{def}{=} \{\theta \in Sub \mid \forall X \in \theta^\flat.(X\theta \ is \ ground)\} \tag{4.36}$$

Lemma 4.5.1 $\gamma[\![\mathcal{V}]\!]$ is monotonic for any $\mathcal{V} \subseteq \mathcal{VAR}$.

PROOF: Immediate from equation 4.36. ∎

4.5.2 Abstract Unification Algorithm for Groundness Analysis

We now present an abstract unification algorithm for groundness analysis. Given $A \in \mathcal{ATOMS}$ with $vars(A) \subseteq \mathcal{U}$, $\theta^\flat \in Sub^\flat[\![\mathcal{U}]\!]$, $B \in \mathcal{ATOMS}$ with $vars(B) \subseteq \mathcal{V}$ and $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$, the algorithm computes $unify^\flat(A, \theta^\flat, B, \sigma^\flat) \in Sub^\flat[\![\mathcal{V}]\!]$ in five steps. In step (1), a renaming substitution Ψ is applied to A and $\theta^\flat$ to obtain $A\Psi$ and $\theta^\flat\Psi$ so that $vars(A\Psi) \cap vars(B) = \emptyset$ and $vars(\theta^\flat\Psi) \cap vars(\sigma^\flat) = \emptyset$, and $\theta^\flat\Psi$ and $\sigma^\flat$ are combined to obtain $\zeta^\flat = \theta^\flat\Psi \cup \sigma^\flat$ so that a substitution satisfying $\zeta^\flat$ satisfies both $\theta^\flat\Psi$ and $\sigma^\flat$. Note that $\zeta^\flat \in Sub^\flat[\![\mathcal{U}\Psi \cup \mathcal{V}]\!]$. In step (2), $E_0 = eq(mgu(A\Psi, B))$ is computed. If $E_0 = fail$ then the algorithm returns $\perp[\![\mathcal{V}]\!] = \mathcal{V}$. Otherwise, the algorithm continues. In step (3), $\eta^\flat = downwards(E_0, \zeta^\flat)$ is computed so that $\eta^\flat$ is satisfied by any $\zeta \circ mgu(E_0\zeta)$ for any ζ satisfying $\zeta^\flat$. In step (4), the algorithm computes $\beta^\flat = upwards(E_0, \eta^\flat)$ from $\eta^\flat$ such that any substitution satisfies $\beta^\flat$ if it satisfies $\eta^\flat$ and unifies E_0. In step (5), the algorithm restricts $\beta^\flat$ to $\mathcal{V}$ and returns the result.

Algorithm 4.5.2 Let $\mathcal{U}, \mathcal{V} \subseteq \mathcal{VAR}$ be finite, $\theta^\flat \in Sub^\flat[\![\mathcal{U}]\!]$, $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$, $vars(A) \subseteq \mathcal{U}$ and $vars(B) \subseteq \mathcal{V}$.

$$unify^\flat(A, \theta^\flat, B, \sigma^\flat) \stackrel{def}{=} \begin{cases} let & \Psi \textit{ be a renaming substitution such that } \mathcal{U}\Psi \cap \mathcal{V} = \emptyset, \\ & E_0 = eq(mgu(A\Psi, B)), \\ if & E_0 \neq \mathit{fail} \\ then & \mathcal{V} \cap upwards(E_0, downwards(E_0, \theta^\flat\Psi \cup \sigma^\flat)) \\ else & \mathcal{V} \end{cases}$$

$$downwards(E, \zeta^\flat) \stackrel{def}{=} \zeta^\flat \cup \bigcup_{(X=t)\in E \wedge X \in \zeta^\flat} vars(t)$$

$$upwards(E, \eta^\flat) \stackrel{def}{=} \eta^\flat \cup \{X \mid (X = t) \in E \wedge vars(t) \subseteq \eta^\flat\}$$

C1-C2 (p. 85) are satisfied by the abstract domain and the concretisation function. C3 (p. 86) is satisfied by $\epsilon^\flat[\![\mathcal{V}]\!]$. The following theorem states that algorithm 4.5.2 satisfies C4 (p. 87).

Theorem 4.5.3

(C4) $unify^\sharp(A, \gamma[\![\mathcal{U}]\!](\theta^\flat), B, \gamma[\![\mathcal{V}]\!](\sigma^\flat)) \subseteq \gamma[\![\mathcal{V}]\!](unify^\flat(A, \theta^\flat, B, \sigma^\flat))$ for any finite $\mathcal{U}, \mathcal{V} \subseteq \mathcal{VAR}$, any $\theta^\flat \in Sub^\flat[\![\mathcal{U}]\!]$, any $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$, and any atoms A and B such that $vars(A) \subseteq \mathcal{U}$ and $vars(B) \subseteq \mathcal{V}$.

PROOF: See p. 173. ∎

Example 4.5.4 Let $A = g(U, f(V, f(W, W)), V)$, $B = g(f(X, Y), Z, X)$, $\theta^\flat = \{U\}$ and $\sigma^\flat = \{Z\}$. This example shows the computation of $unify^\flat(A, \theta^\flat, B, \sigma^\flat)$ by algorithm 4.5.2.

$\theta^\flat$ is an abstract substitution on domain $\mathcal{U} = \{U, V, W\}$ and $\sigma^\flat$ is an abstract substitution on domain $\mathcal{V} = \{X, Y, Z\}$.

In step (1), a renaming substitution $\Psi = \{U/U_0, V/V_0, W/W_0\}$ is applied to A and $\theta^\flat$.

$$\begin{aligned} A\Psi &= g(U_0, f(V_0, f(W_0, W_0)), V_0) \\ \theta^\flat\Psi &= \{U_0\} \end{aligned}$$

and $\zeta^\flat = (\theta^\flat\Psi \cup \sigma^\flat) = \{U_0, Z\}$ is computed.

In step (2), $E_0 = eq(mgu(A\Psi, B)) = \{U_0 = f(V_0, Y), Z = f(V_0, f(W_0, W_0)), X = V_0\}$ is computed.

In step (3), $\eta^\flat = downwards(E_0, \zeta^\flat) = \{U_0, Z, V_0, Y, W_0\}$ is computed.

In step (4), $\beta^\flat = upwards(E_0, \eta^\flat) = \{U_0, Z, V_0, Y, W_0, X\}$ is computed.

In step (5), the algorithm computes and returns $restrict(\beta^\flat, \mathcal{V}) = \{X, Y, Z\}$

So, $unify^\flat(A, \theta^\flat, B, \sigma^\flat) = \{X, Y, Z\}$. ∎

Example 4.5.5 This example shows the result of the groundness analysis of the $\mathcal{P}$ and $\mathcal{G}$ in example 2.2.1. $lfp F^{\flat}_{\mathcal{P},\mathcal{G}}$ has 23 components each of which corresponds to one edge in $\mathcal{E}_{\mathcal{P},\mathcal{G}}$. The following are four of them.

$$\begin{array}{lcl}
[lfp F^{\flat}_{\mathcal{P},\mathcal{G}}]_{(5,1)\leftarrow\bullet(0,0)} & = & \{Y, Z\} \\
[lfp F^{\flat}_{\mathcal{P},\mathcal{G}}]_{(3,1)\leftarrow\bullet(1,1)} & = & \{X, L\} \\
[lfp F^{\flat}_{\mathcal{P},\mathcal{G}}]_{(1,2)\leftarrow\bullet(3,1)} & = & \{X, L, K\} \\
[lfp F^{\flat}_{\mathcal{P},\mathcal{G}}]_{(1,3)\leftarrow\bullet(1,2)} & = & \{X, L, K\}
\end{array}$$

∎

4.6 Discussion

The problem of forward abstract interpretation of normal logic programs has not been addressed in the literature. We have proposed a straightforward solution to this problem. We now discuss this solution, and outline some other plausible solutions and the work required to justify them.

4.6.1 Negation as Failure

The treatment of negation as failure in $lfp F^{\flat}_{\mathcal{P},\mathcal{G}}$ and $lfp F^{\diamond}_{\mathcal{P},\mathcal{G}}$ is straightforward. The transition system $F_{\mathcal{P},\mathcal{G}}$ approximates VSLDNF (an equivalent of SLDNF) by assuming that a negative literal always succeeds while it may fail. This approximation introduces noise into $lfp F^{\flat}_{\mathcal{P},\mathcal{G}}$ and $lfp F^{\diamond}_{\mathcal{P},\mathcal{G}}$. However, any attempt to improve on this straightforward treatment is difficult within the provisions of abstract interpretation.

Let $(\neg A)\sigma$ be selected by SLDNF where $\neg A$ is a negative literal in the body of a clause or an initial goal and σ is a substitution. Let $\mathcal{V}$ be the set of variables in the clause or the initial goal. During abstract interpretation, σ is not known and possible values for σ are described by an abstract substitution $\sigma^{\flat} \in Sub^{\flat}[\![\mathcal{V}]\!]$, often called the abstract call substitution for $(\neg A)$. Since $\sigma^{\flat}$ usually describes an infinite set of substitutions, it may well be the case that $A\sigma$ succeeds for some $\sigma \in \gamma[\![\mathcal{V}]\!](\sigma^{\flat})$ and fails for other $\sigma \in \gamma[\![\mathcal{V}]\!](\sigma^{\flat})$. We take $\sigma^{\flat}$ as the abstract success substitution for $\neg A$ by simply assuming that $A\sigma$ fails for all $\sigma \in \gamma[\![\mathcal{V}]\!](\sigma^{\flat})$. An improvement on this straightforward treatment needs to make the abstract success substitution for $\neg A$ stronger, that is, replacing $\sigma^{\flat}$ with another abstract substitution $\eta^{\flat} \in Sub^{\flat}[\![\mathcal{V}]\!]$ such that $\eta^{\flat} \sqsubseteq[\![\mathcal{V}]\!]\ \sigma^{\flat}$. Let us assume that $\gamma[\![\mathcal{V}]\!](\sigma^{\flat}) \setminus \gamma[\![\mathcal{V}]\!](\eta^{\flat}) \neq \emptyset$, for otherwise $\eta^{\flat}$ is no stronger than $\sigma^{\flat}$. By the weaker safeness condition for the negation as failure rule and the safeness requirement for abstract interpretation, it is necessary to be able to infer

$$\forall \theta \in \gamma[\![\mathcal{V}]\!](\sigma^{\flat}) \setminus \gamma[\![\mathcal{V}]\!](\eta^{\flat}). \left(\begin{array}{l} A\theta \textit{ succeeds with } \epsilon \textit{ being a computed} \\ \textit{answer in } SLDNF \end{array} \right) \tag{4.37}$$

In 4.37, we need to *under-estimate* success and *over-estimate* failure in order to make the analysis safe. However, abstract interpreters over-estimate success and under-estimate failure. Note that the approximation made by the abstract interpreters is an over-estimation. An abstract interpreter over-estimates success by means of an abstract unification function which over-estimates the success of the concrete unification function. To infer 4.37, we must use a unification function which under-estimates the success of the concrete unification function. Such a unification function should succeed only if the concrete unification function succeeds. Of course, we could use a unification function which always fails. But this does not achieve any improvement since $A\sigma$ is always estimated to fail. Since an abstract domain for an analysis is much simpler than the concrete domain, some information about a set of substitutions is lost when the set of substitutions is approximated by an abstract substitution. It is difficult to design a unification function which under-estimates the success of the concrete unification function based on information represented by abstract substitutions because abstract substitutions are inaccurate descriptions of sets of substitutions and such a unification function, we believe, needs accurate descriptions of sets of substitutions.

4.6.2 Modification of Existing Generic Abstract Semantics

We have developed our generic abstract semantics $F^{\flat}_{\mathcal{P},\mathcal{G}}$ and $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ through a series of approximations of SLDNF - the operational semantics of normal logic programs. The generic abstract semantics for forward analyses of logic programs in the literature [9, 61, 87, 89] are derived by means of a series of approximations of SLD - the operational semantics of definite logic programs. We believe that these generic abstract semantics can be reworked by using SLDNF instead of SLD as the operational semantics and introducing an extra approximation which approximates SLDNF by a transition system. The derivation and the safeness proof of the resultant generic abstract semantics would look like a standard exercise and the resultant generic abstract semantics could be close to the original generic abstract semantics derived from SLD, for instance, $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ is close to that in [101, 102]. However, it is essential to do such exercises rather than just making changes to the original generic abstract semantics derived from SLD and assuming them to be safe.

4.6.3 Direct Use of Generic Abstract Semantics

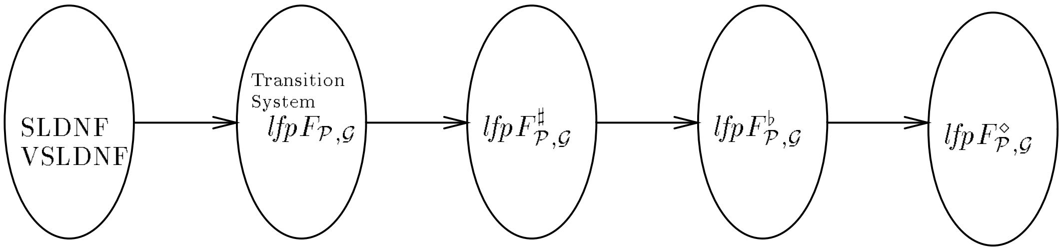
Another plausible approach to the forward abstract interpretation of normal logic programs is first to transform a normal logic program $\mathcal{P}$ and a set $\mathcal{G}$ of normal goal descriptions into a definite logic program $\mathcal{P}'$ and a set $\mathcal{G}'$ of definite goal descriptions and then directly apply an existing generic abstract semantics to $\mathcal{P}'$ and $\mathcal{G}'$. Consider the following transformation. $\mathcal{P}'$ and $\mathcal{G}'$ is obtained from $\mathcal{P}$ and $\mathcal{G}$ by changing each $L_p \equiv \neg B_p \equiv \neg a(\bar{X})$ into $not\_a(\bar{X})$ where $\bar{X}$ denotes the set of variables in L_p. In addition, $\mathcal{P}'$ contains, for each $L_p \equiv \neg B_p \equiv \neg a(\bar{X})$,

$$\begin{aligned} &not\_a(\bar{X}) \leftarrow a(\bar{X}), fail \\ &not\_a(\bar{X}) \end{aligned}$$

Before analysing $\mathcal{P}$ and $\mathcal{G}$ by applying an existing generic abstract semantics to $\mathcal{P}'$ and $\mathcal{G}'$, we need to prove that, for any $\leftarrow L_1, \cdots, L_m$, if $L_1', \cdots, L_m'$ is obtained from $L_1, \cdots, L_m$ as above and there is an SLDNF-derivation of $\mathcal{P} \cup \{\leftarrow L_1, \cdots, L_m\}$ reaching a program point p with current substitution σ then there is an SLD-derivation of $\mathcal{P}' \cup \{\leftarrow L_1', \cdots, L_m'\}$ reaching p with current substitution σ' such that σ' is equivalent (module renaming) to σ.

4.7 Summary

We have presented two generic fixed-point abstract semantics $F^\flat_{\mathcal{P},\mathcal{G}}$ and $F^\diamond_{\mathcal{P},\mathcal{G}}$ for forward analyses of normal logic programs under the SLDNF-resolution via the left-to-right computation rule. An analysis can be thought of as a series of approximations of the operational semantics as shown below. An arrow from A to B reads as "A is approximated by B".



To specialise either $F^\flat_{\mathcal{P},\mathcal{G}}$ or $F^\diamond_{\mathcal{P},\mathcal{G}}$ for an analysis, one needs to devise an abstract domain, a concretisation function, an abstract identity substitution and an abstract unification algorithm for the analysis. The abstract domain, the concretisation function and the abstract identity substitution must satisfy C1-C3 and the abstract unification algorithm must satisfy C4.

We deal with negation as failure by approximating SLDNF with a transition system. The way negation as failure is dealt with may be generalised to deal with the built-in predicate *cut*

(!/0). Although, we have not dealt with other built-in predicates, we believe that the generic abstract semantics can be augmented to deal with these built-in predicates in the way they are dealt with by Bruynooghe in [9].

Chapter 5

Mode and Type Analyses

In this chapter, we specialise $F^{\flat}_{\mathcal{P},\mathcal{G}}$ and $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ for inferring mode and type information. Mode and type information are both inferred together with pair sharing and aliasing information. A generic abstract domain and a generic abstract unification algorithm are developed. An abstract substitution on $\mathcal{V}$ is a triple $<\mathcal{I},\mathcal{S},\mathcal{A}>$. $\mathcal{I}$ is a function from $\mathcal{V}$ to a set $\check{D}$ of *descriptions*. Each description in $\check{D}$ describes a set of terms. There is an order $\check{\sqsubseteq}$ on $\check{D}$ such that $<\check{D}, \check{\sqsubseteq}>$ is a complete lattice and a monotonic function $\check{\gamma}$ from $\check{D}$ to $\wp(\mathcal{TERMS})$. For mode analysis, $<\check{D}, \check{\sqsubseteq}>$ is a complete lattice of *mode descriptions* and, for type analysis, $<\check{D}, \check{\sqsubseteq}>$ is a complete lattice of *type descriptions*. $\mathcal{S}$ is a symmetric relation on $\mathcal{V}$ and $\mathcal{A}$ is an equivalence relation on $\mathcal{V}$. Let $\theta \in \gamma[\![\mathcal{V}]\!](<\mathcal{I},\mathcal{S},\mathcal{A}>)$. $\mathcal{I}(X)$ describes possible values for $X\theta$, $\mathcal{S}$ describes pair sharing relationships between variables in $\mathcal{V}$ under θ, and $\mathcal{A}$ describes aliasing relationships between variables in $\mathcal{V}$ under θ.

In section 5.1, we construct the generic abstract domain and the corresponding concretisation function. In section 5.2, we investigate the relationship between the three parts of an abstract substitution, propose the rationale for the use of aliasing information in the detection of failure in concrete unification, and define some preliminary functions that are used in the generic abstract unification algorithm. Section 5.3 presents the generic abstract unification algorithm. Sections 5.4 and 5.5 specialise the generic abstract domain and the generic abstract unification algorithm for mode and type analyses respectively. Finally, section 5.6 concludes the chapter.

5.1 Generic Abstract Domain

Let $\mathcal{V} \subseteq \mathcal{VAR}$ be finite. An abstract substitution $\theta^{\flat}$ on $\mathcal{V}$ is a triple $<\mathcal{I},\mathcal{S},\mathcal{A}>$ where $\mathcal{I}$, called an *abstract instantiation state* on $\mathcal{V}$, maps each variable in $\mathcal{V}$ to a description in $\check{D}$, $\mathcal{S}$ describes possible pair sharing among variables in $\mathcal{V}$ and $\mathcal{A}$ describes definite aliasing among variables in $\mathcal{V}$.

An abstract instantiation state $\mathcal{I}$ on $\mathcal{V}$ is a function from $\mathcal{V}$ into $\check{D}$. Let $\mathbf{IS}[\![\mathcal{V}]\!] \stackrel{def}{=} \mathcal{V} \mapsto \check{D}$ be the set of abstract instantiation states on $\mathcal{V}$ and $\gamma_i[\![\mathcal{V}]\!] \in \mathbf{IS}[\![\mathcal{V}]\!] \mapsto \wp(Sub)$ be defined as follows.

$$\gamma_i[\![\mathcal{V}]\!](\mathcal{I}) \stackrel{def}{=} \{\theta \in Sub \mid \forall X \in \mathcal{V}.(X\theta \in \check{\gamma}(\mathcal{I}(X)))\}$$

A substitution θ is said to satisfy an abstract instantiation state $\mathcal{I}$ on $\mathcal{V}$ if $\theta \in \gamma_i[\![\mathcal{V}]\!](\mathcal{I})$.

Two distinct variables X, Y share under a substitution θ if they are instantiated by θ to two terms that have common variables, i.e., $vars(X\theta) \cap vars(Y\theta) \neq \emptyset$. A variable X is non-linear under a substitution θ if there are repeated occurrences of variables in $X\theta$. Let X be a variable and t a term. $repeat(X,t)$ is true iff X occurs more than once in t. The multiplicity function $\chi \in \mathcal{TERMS} \mapsto \{0,1,2\}$ is defined as follows [15].

$$\chi(t) \stackrel{def}{=} \begin{cases} 0 & if\ vars(t) = \emptyset \\ 2 & if\ \exists X \in vars(t).repeat(X,t) \\ 1 & otherwise \end{cases}$$

The possible pair sharing relationship among variables in $\mathcal{V}$ is a symmetric relation $\mathcal{S}$ on $\mathcal{V}$. Let $\mathbf{SR}[\![\mathcal{V}]\!] \stackrel{def}{=} \{\mathcal{S} \mid \mathcal{S}\ is\ a\ symmetric\ relation\ on\ \mathcal{V}\}$. $<\mathbf{SR}[\![\mathcal{V}]\!], \subseteq>$ is a complete lattice. Let $\gamma_s[\![\mathcal{V}]\!] \in \mathbf{SR}[\![\mathcal{V}]\!] \mapsto \wp(Sub)$ be defined as follows.

$$\gamma_s[\![\mathcal{V}]\!](\mathcal{S}) \stackrel{def}{=} \left\{ \theta \in Sub \;\middle|\; \begin{array}{c} \forall X, Y \in \mathcal{V}.(X \not\equiv Y \wedge (X,Y) \notin \mathcal{S} \rightarrow vars(X\theta) \cap vars(Y\theta) = \emptyset) \\ \wedge \\ \forall X \in \mathcal{V}.((X,X) \notin \mathcal{S} \rightarrow \chi(X\theta) \neq 2) \end{array} \right\}$$

$<\mathbf{SR}[\![\mathcal{V}]\!], \subseteq>$ is a complete lattice. A substitution θ is said to satisfy $\mathcal{S}$ if $\theta \in \gamma_s[\![\mathcal{V}]\!](\mathcal{S})$.

Two variables X and Y are aliased under a substitution θ if $X\theta \equiv Y\theta$. The aliasing relationship among variables in $\mathcal{V}$ is an equivalence relation $\mathcal{A}$ on $\mathcal{V}$. Let $\mathbf{ER}[\![\mathcal{V}]\!] \stackrel{def}{=} \{\mathcal{A} \mid \mathcal{A}\ is\ an\ equivalence\ relation\ on\ \mathcal{V}\}$. $<\mathbf{ER}[\![\mathcal{V}]\!], \supseteq>$ is a complete lattice with infimum $\mathcal{V}^2$, supremum $\{<X,X> \mid X \in \mathcal{V}\}$, greatest lower bound operator $\cap$ and least upper bound operator $\lambda\mathcal{A}_1, \mathcal{A}_2 \in \mathbf{ER}[\![\mathcal{V}]\!].(\mathcal{A}_1 \cup \mathcal{A}_2)^*$ where $(\mathcal{A}_1 \cup \mathcal{A}_2)^*$ is the transitive closure of $\mathcal{A}_1 \cup \mathcal{A}_2$. Let $\gamma_a[\![\mathcal{V}]\!] \in \mathbf{ER}[\![\mathcal{V}]\!] \mapsto \wp(Sub)$ be defined as follows.

$$\gamma_a[\![\mathcal{V}]\!](\mathcal{A}) \stackrel{def}{=} \{\theta \in Sub \mid \forall X, Y \in \mathcal{V}.((X,Y) \in \mathcal{A} \rightarrow X\theta \equiv Y\theta)\}$$

A substitution θ is said to satisfy $\mathcal{A}$ if $\theta \in \gamma_a[\![\mathcal{V}]\!](\mathcal{A})$.

Definition 5.1.1 Let $\mathcal{V} \subseteq \mathcal{VAR}$ and define

$$Sub^{\flat}[\![\mathcal{V}]\!] \stackrel{def}{=} \mathbf{IS}[\![\mathcal{V}]\!] \times \mathbf{SR}[\![\mathcal{V}]\!] \times \mathbf{ER}[\![\mathcal{V}]\!]$$

$$<\mathcal{I}_1, \mathcal{S}_1, \mathcal{A}_1> \ \sqsubseteq[\![\mathcal{V}]\!] \ <\mathcal{I}_2, \mathcal{S}_2, \mathcal{A}_2> \stackrel{def}{=} (\forall X \in \mathcal{V}.\mathcal{I}_1(X) \check{\sqsubseteq} \mathcal{I}_2(X)) \wedge (\mathcal{S}_1 \subseteq \mathcal{S}_2) \wedge (\mathcal{A}_1 \supseteq \mathcal{A}_2)$$

∎

Lemma 5.1.2 C1 ($<Sub^{\flat}[\![\mathcal{V}]\!], \sqsubseteq[\![\mathcal{V}]\!]>$ is a complete lattice) holds if

C1': $<\check{D}, \ \check{\sqsubseteq} \ >$ is a complete lattice.

PROOF: $<\mathbf{IS}[\![\mathcal{V}]\!], \lambda\mathcal{I}_1, \mathcal{I}_2 \in \mathbf{IS}[\![\mathcal{V}]\!].\forall X \in \mathcal{V}.(\mathcal{I}_1(X) \subseteq \mathcal{I}_2(X))>$ is a complete lattice by lemma 2.1.6 and C1'. By lemma 2.1.8, $<Sub^{\flat}[\![\mathcal{V}]\!], \sqsubseteq[\![\mathcal{V}]\!]>$ is a complete lattice. ∎

Lemma 5.1.2 reduces the design of the complete lattice $<Sub^{\flat}[\![\mathcal{V}]\!], \sqsubseteq[\![\mathcal{V}]\!]>$ of abstract substitutions for an analysis into the design of the complete lattice $<\check{D}, \ \check{\sqsubseteq} \ >$ of descriptions for that analysis.

Let $\theta^{\flat} =<\mathcal{I}, \mathcal{S}, \mathcal{A}>$. Define $\theta^{\flat}\downarrow Inst \stackrel{def}{=} \mathcal{I}$, $\theta^{\flat}\downarrow Share \stackrel{def}{=} \mathcal{S}$ and $\theta^{\flat}\downarrow Alias \stackrel{def}{=} \mathcal{A}$.

Definition 5.1.3 ($\gamma[\![\mathcal{V}]\!] \in Sub^{\flat}[\![\mathcal{V}]\!] \mapsto \wp(Sub)$)

$$\gamma[\![\mathcal{V}]\!](\theta^{\flat}) \stackrel{\text{def}}{=} \gamma_i[\![\mathcal{V}]\!](\theta^{\flat}\downarrow Inst) \cap \gamma_s[\![\mathcal{V}]\!](\theta^{\flat}\downarrow Share) \cap \gamma_a[\![\mathcal{V}]\!](\theta^{\flat}\downarrow Alias)$$

∎

Let $\theta^{\flat} =<\mathcal{I}, \mathcal{S}, \mathcal{A}>$. A substitution θ is said to satisfy $\theta^{\flat}$ if $\theta \in \gamma[\![\mathcal{V}]\!](\theta^{\flat})$. θ satisfies $\theta^{\flat}$ iff θ satisfies $\mathcal{I}$, $\mathcal{S}$ and $\mathcal{A}$ simultaneously according to definition 5.1.3.

Lemma 5.1.4 C2 ($\gamma[\![\mathcal{V}]\!]$ is monotonic) holds if

C2': $\check{\gamma} \in \check{D} \mapsto \wp(\mathcal{TERMS})$ is monotonic

PROOF: C2' implies that $\gamma_i[\![\mathcal{V}]\!]$ is monotonic. $\gamma_s[\![\mathcal{V}]\!]$ and $\gamma_a[\![\mathcal{V}]\!]$ are monotonic. Therefore, C2' implies that $\gamma[\![\mathcal{V}]\!]$ is monotonic. ∎

Lemma 5.1.4 reduces the design of the concretisation function $\gamma[\![\mathcal{V}]\!]$ from $<Sub^{\flat}[\![\mathcal{V}]\!], \sqsubseteq[\![\mathcal{V}]\!]>$ to $<\wp(Sub), \subseteq>$ for an analysis to the design of the concretisation function $\check{\gamma}$ from $<\check{D}, \ \check{\sqsubseteq} \ >$ to $<\wp(\mathcal{TERMS}), \subseteq>$. $\bot[\![\mathcal{V}]\!] =<\{X/\check{\bot} \mid X \in \mathcal{V}\}, \emptyset, \mathcal{V}^2>$ where $\check{\bot}$ is the infimum of $<\check{D}, \ \check{\sqsubseteq} \ >$.

Let $\check{\epsilon} \in \check{D}$ be the least element such that $\forall X \in \mathcal{VAR}.(X \in \check{\gamma}(\check{\epsilon}))$. The abstract identity substitution on $\mathcal{V}$ is constructed as follows.

$$\epsilon^{\flat}[\![\mathcal{V}]\!] \stackrel{def}{=} <\{X/\check{\epsilon} \mid X \in \mathcal{V}\}, \emptyset, \{(X,X) \mid X \in \mathcal{V}\}>$$

Lemma 5.1.5 C3 ($\epsilon \in \gamma[\![\mathcal{V}]\!](\epsilon^{\flat}[\![\mathcal{V}]\!])$) holds if

C3': $\forall X \in \mathcal{VAR}.(X \in \check{\gamma}(\check{\epsilon}))$

PROOF: Immediate. ∎

Lemma 5.1.5 reduces the design of the abstract identity substitution function for an analysis to the identification of $\check{\epsilon}$ in $\check{D}$.

5.2 Preliminary Lemmas and Ancillary Functions

This section investigates the relationship between the three parts of an abstract substitution, proposes the rationale for the use of aliasing information in the detection of failure in concrete unification, and defines some preliminary functions that are used in the generic abstract unification algorithm.

Lemma 5.2.1 states that the unification of $X\sigma$ and $t\sigma$ does not change the description of the value of any variable Y that does not share with variables in $\{X\} \cup vars(t)$ under σ. Lemma 5.2.2 identifies other variables the descriptions of the values of which are not affected by the unification of $X\sigma$ and $t\sigma$.

Lemma 5.2.1 Let X and Y be variables, t a term and σ a substitution such that Y does not share with any variable in $\{X\} \cup vars(t)$ under σ. If $\eta = mgu(X\sigma, t\sigma) \neq \mathit{fail}$ then $Y\sigma\eta = Y\sigma$.

PROOF: $dom(\eta) \subseteq vars(X\sigma) \cup \bigcup\{vars(U\sigma) \mid U \in vars(t)\}$ since η is relevant. Because Y and U do not share under σ for any $U \in \{X\} \cup vars(t)$, we have $vars(Y\sigma) \cap dom(\eta) = \emptyset$. So, $Y\sigma\eta = Y\sigma$. ∎

We require that $<\check{D}, \check{\sqsubseteq}>$ and $\check{\gamma}$ satisfy the following condition.

C4': For any $\check{d} \in \check{D}$, any term t and any $\delta \in Sub$ such that $X\delta$ is a variable for any $X \in \mathcal{VAR}$,

$$t \in \check{\gamma}(\check{d}) \rightarrow t\delta \in \check{\gamma}(\check{d})$$

Lemma 5.2.2 Let $C4'$ hold, $\check{d} \in \check{D}$, X, Y be variables, t a term, σ a substitution, and $\eta = mgu(X\sigma, t\sigma) \neq \mathit{fail}$. $Y\sigma \in \check{\gamma}(\check{d}) \rightarrow Y\sigma\eta \in \check{\gamma}(\check{d})$ if either (1) $X\sigma$ and $t\sigma$ are ground, or (2) $X\sigma$ and $t\sigma$ are variables, or (3) $X\sigma$ is a variable and $Y \neq X$ and Y does not share with X under σ, or (4) $t\sigma$ is a variable and Y does not share with any $U \in vars(t)$ under σ.

PROOF: (1) $Y\sigma\eta = Y\sigma$ since $\eta = \emptyset$. So, $Y\sigma \in \check{\gamma}(\check{d}) \rightarrow Y\sigma\eta \in \check{\gamma}(\check{d})$. (2) η maps any variable into a variable. So, $Y\sigma \in \check{\gamma}(\check{d}) \rightarrow Y\sigma\eta \in \check{\gamma}(\check{d})$ by C4'. (3) $X\sigma$ is a variable and $dom(\eta) = \{X\sigma\}$.

Because Y does not share with X under σ, $vars(Y\sigma) \cap dom(\eta) = vars(Y\sigma) \cap \{X\sigma\} = \emptyset$. Therefore, $Y\sigma \in \check{\gamma}(\check{d}) \rightarrow Y\sigma\eta \in \check{\gamma}(\check{d})$. (4) is symmetric with (3). ∎

Let $\mathbf{grd} : \check{D} \mapsto \{false, true\}$ and $\mathbf{var} : \check{D} \mapsto \{false, true\}$ satisfy C5' and C6' respectively.

C5': $\forall \check{d} \in \check{D}.\forall t \in \check{\gamma}(\check{d}).(\mathbf{grd}(\check{d}) \rightarrow t\ is\ a\ ground\ term)$.

C6': $\forall \check{d} \in \check{D}.\forall t \in \check{\gamma}(\check{d}).(\mathbf{var}(\check{d}) \rightarrow t\ is\ a\ variable)$.

Definition 5.2.3 Let $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$, $X \in \mathcal{V}$ and t' with $vars(t') \subseteq \mathcal{V}$ be a non-variable term.

$$\chi^\flat(X, \sigma^\flat) \stackrel{def}{=} \begin{cases} 0 & if\ \mathbf{grd}((\sigma^\flat \downarrow Inst)(X)) \\ 2 & if\ (X, X) \in \sigma^\flat \downarrow Share \\ 1 & otherwise \end{cases}$$

$$\chi^\flat(t', \sigma^\flat) \stackrel{def}{=} \begin{cases} 0 & if\ \forall X \in vars(t').\chi^\flat(X, \sigma^\flat) = 0 \\ 1 & if\ (\forall X \in vars(t').(repeat(X, t') \rightarrow \mathbf{grd}((\sigma^\flat \downarrow Inst)(X))) \\ & \quad and \\ & \quad \forall Y, Z \in vars(t').(\chi^\flat(Y, \sigma^\flat) = 1 \wedge \chi^\flat(Z, \sigma^\flat) = 1 \rightarrow (Y, Z) \notin \sigma^\flat \downarrow Share) \\ 2 & otherwise \end{cases}$$

∎

By definition 5.2.3, if C5' holds then $\chi^\flat(t, \sigma^\flat) \geq \chi(t\theta)$ for any substitution $\theta \in \gamma[\![\mathcal{V}]\!](\sigma^\flat)$.

Aliasing information is also used by the generic abstract unification algorithm to detect failure in concrete unification. One rationale for the detection of failure in concrete unification is that a term cannot be a proper sub-term of itself. Let t and t' be terms. t' is a proper sub-term of t iff $t' \not\equiv t$ and t' is a sub-term of t. Let $\mathcal{A}$ be an aliasing relation on $\mathcal{V}$, and E be a set of equations of the form $X = t$ where $X \in \mathcal{V}$ is a variable and t is a non-variable term with $vars(t) \subseteq \mathcal{V}$. Define

$$pst(\mathcal{A}, E) \stackrel{def}{=} \{<[V]_\mathcal{A}, [X]_\mathcal{A}> \mid (X = t) \in E \wedge V \in vars(t)\}$$

If $<[V]_\mathcal{A}, [X]_\mathcal{A}> \in pst(\mathcal{A}, E)$ then $V\sigma$ is a proper sub-term of $X\sigma$ for any substitution $\sigma \in \gamma_a[\![\mathcal{V}]\!](\mathcal{A})$. The proper sub-term relationship is transitive. Therefore, if there is a variable X such that $<[X]_\mathcal{A}, [X]_\mathcal{A}> \in (pst(\mathcal{A}, E))^*$, the transitive closure of $pst(\mathcal{A}, E)$, then $E\sigma$ cannot be unified for any $\sigma \in \gamma_a[\![\mathcal{V}]\!](\mathcal{A})$.

Another rationale for the detection of failure in concrete unification is that if $t_X\sigma \equiv t_Y\sigma$ then t_X and t_Y must unify.

$$failure(\mathcal{A},E) \stackrel{def}{=} \begin{cases} \exists X. <[X]_{\mathcal{A}},[X]_{\mathcal{A}}> \in (pst(\mathcal{A},E))^* \\ \vee \\ \exists (X=t_X),(Y=t_Y) \in E.(X \not\equiv Y \wedge (X,Y) \in \mathcal{A} \wedge mgu(t_X,t_Y) = \mathit{fail}) \end{cases}$$

Lemma 5.2.4 Let $\mathcal{V} \subseteq \mathcal{VAR}$, $\mathcal{A}$ be an equivalence relation and E be a set of equations $X = t$ where $X \in \mathcal{V}$ is a variable and t is a non-variable term such that $vars(t) \subseteq \mathcal{V}$. If $failure(\mathcal{A},E)$ then $mgu(E\sigma) = \mathit{fail}$ for any $\sigma \in \gamma_a[\![\mathcal{V}]\!](\mathcal{A})$.

PROOF: Immediate from the above argument. ∎

Lemma 5.2.4 allows aliasing information to be used to improve the precision of analyses by detecting failure in concrete unification. We have not found any evidence in the literature of this kind of use of aliasing information.

5.3 Generic Abstract Unification Algorithm

This section presents the generic abstract unification algorithm. In addition to those functions specified in section 5.2, the generic algorithm requires the analysis-dependent functions $\mathbf{ins} : \check{D} \mapsto \check{D}$ and $\mathbf{est} : \wp(\mathcal{EQ}) \times \mathbf{IS}[\![\mathcal{V}]\!] \mapsto \mathbf{IS}[\![\mathcal{V}]\!]$ which must satisfy C7' and C8' respectively.

C7': For any $\check{d} \in \check{D}$, any $t \in \check{\gamma}(\check{d})$ and any $\theta \in Sub$, $t\theta \in \check{\gamma}(\mathbf{ins}(\check{d}))$.

C8': If $X \in \mathcal{V}$, $vars(t) \subseteq \mathcal{V}$, $\mathcal{I} \in \mathbf{IS}[\![\mathcal{V}]\!]$, $\sigma \in \gamma_i[\![\mathcal{V}]\!](\mathcal{I})$ and $\eta = mgu(X\sigma, t\sigma) \neq \mathit{fail}$ then $\mathcal{I}_0 = \mathbf{est}(X=t,\mathcal{I}) \rightarrow \forall Y \in vars(t) \cup \{X\}.Y\sigma\eta \in \check{\gamma}(\mathcal{I}_0(Y))$.

Let $\mathcal{U},\mathcal{V} \subseteq \mathcal{VAR}$, $\theta^\flat \in Sub^\flat[\![\mathcal{U}]\!]$, $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$, and A, B be atoms such that $vars(A) \subseteq \mathcal{U}$ and $vars(B) \subseteq \mathcal{V}$. The algorithm for $unify^\flat$ computes $unify^\flat(A,\theta^\flat,B,\sigma^\flat)$ in three stages. Let Ψ be a renaming substitution such that $\mathcal{U}\Psi \cap \mathcal{V} = \emptyset$. In the first stage, it computes $E_0 = eq(mgu(\{A\Psi = B\}))$. If $E_0 = \mathit{fail}$ then it returns $\perp[\![\mathcal{V}]\!]$ since $\perp[\![\mathcal{V}]\!]$ approximates the empty set $\emptyset$ of substitutions more precisely than any other abstract substitution in $Sub^\flat[\![\mathcal{V}]\!]$. In the second stage, letting $\mathcal{W} = \mathcal{U}\Psi \cup \mathcal{V}$ and $\theta^\flat\Psi \uplus \sigma^\flat \in Sub^\flat[\![\mathcal{W}]\!]$ be the component-wise unions of $\theta^\flat\Psi$ and $\sigma^\flat$, it computes an abstract substitution $\eta^\flat \in Sub^\flat[\![\mathcal{W}]\!]$ such that, $\zeta \circ mgu(E_0\zeta) \in \gamma[\![\mathcal{W}]\!](\eta^\flat)$ for any $\zeta \in \gamma[\![\mathcal{W}]\!](\theta^\flat \uplus \sigma^\flat)$. This is accomplished by a function *solve* which solves a set of equations in solved form in an abstract substitution. In the third stage, it restricts $\eta^\flat$ to $\mathcal{V}$. This is accomplished by a function *restrict*. Note that $vars(\sigma^\flat \downarrow Alias) = \mathcal{V}$ for any $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$.

Algorithm 5.3.1 Let $\mathcal{U},\mathcal{V}\subseteq \mathcal{VAR}$, $\theta^\flat \in Sub^\flat[\![\mathcal{U}]\!]$, $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$, and A, B be atoms such that $vars(A)\subseteq \mathcal{U}$ and $vars(B)\subseteq\mathcal{V}$. Define $share(X,t,\mathcal{S}) \stackrel{def}{=} \exists Y\in vars(t).(X,Y)\in\mathcal{S}$ where X is a variable, t is a term and $\mathcal{S}$ is a pair sharing relation.

$$unify^\flat(A,\theta^\flat,B,\sigma^\flat) \stackrel{def}{=} \begin{cases} let & \Psi \text{ be a renaming substitution such that} \\ & \mathcal{U}\Psi\cap\mathcal{V}=\emptyset, \\ & E_0 = eq(mgu(A\Psi,B)), \\ & \mathcal{V} = vars(\sigma^\flat\downarrow Alias) \\ in & \\ \multicolumn{2}{l}{\begin{cases} restrict(solve(E_0,\theta^\flat\Psi\uplus\sigma^\flat),\mathcal{V}) & if\ E_0\neq \mathit{fail} \\ \bot[\![\mathcal{V}]\!] & otherwise\end{cases}} \end{cases} \tag{5.1}$$

$$<\mathcal{I},\mathcal{S},\mathcal{A}>\uplus<\mathcal{I}',\mathcal{S}',\mathcal{A}'> \stackrel{def}{=} <\mathcal{I}\cup\mathcal{I}',\mathcal{S}\cup\mathcal{S}',\mathcal{A}\cup\mathcal{A}'> \tag{5.2}$$

$$solve(E,\theta^\flat) \stackrel{def}{=} \begin{cases} let & E_1 = \{(X=Y)\in E \mid Y\ \in \mathcal{VAR}\}, \\ & E_2 = E\setminus E_1, \\ & \zeta^\flat = solve0(E_1,\theta^\flat) \\ in & \\ \multicolumn{2}{l}{\begin{cases} \bot[\![vars(\theta^\flat\downarrow Alias)]\!] & if\ failure(\zeta^\flat\downarrow Alias, E_2) \\ solve0(E_2,\zeta^\flat) & otherwise\end{cases}} \end{cases} \tag{5.3}$$

$$solve0(E,\theta^\flat) \stackrel{def}{=} \begin{cases} \theta^\flat & if\ E=\emptyset \\ solve0(E', soln(e,\theta^\flat)) & if\ E=\{e\}\cup E' \end{cases} \tag{5.4}$$

$$restrict(<\mathcal{I},\mathcal{S},\mathcal{A}>,\mathcal{V}) \stackrel{def}{=} <\{X/\mathcal{I}(X)\mid X\in\mathcal{V}\},\mathcal{S}\cap\mathcal{V}^2,\mathcal{A}\cap\mathcal{V}^2> \tag{5.5}$$

$$soln(e,\sigma^\flat) \stackrel{def}{=} reduce(compose(\sigma^\flat, unieq(e,\sigma^\flat), vars(e))) \tag{5.6}$$

$$unieq(X=t,<\mathcal{I},\mathcal{S},\mathcal{A}>) \stackrel{def}{=} <<\mathcal{I}',\mathcal{S}',\mathcal{A}'>,\mathcal{Z}'> \tag{5.7}$$

where

$$\mathcal{I}' = \begin{cases} let\ \mathcal{I}_0 = \mathbf{est}(X=t,\mathcal{I}) \\ in \\ \lambda U. \begin{cases} \mathcal{I}_0(Y) & if\ \exists Y\in vars(t)\cup\{X\}.((U,Y)\in\mathcal{A}) \\ \mathcal{I}(U) & otherwise \end{cases} \end{cases}$$

$$\mathcal{S}' = \begin{cases} let\ \mathcal{V}_t = vars(t) \setminus \{V \mid \mathbf{grd}(\mathcal{I}(V))\}, \\ \quad \kappa^1 = \chi^\flat(X, <\mathcal{I}, \mathcal{S}, \mathcal{A}>), \kappa^2 = \chi^\flat(t, <\mathcal{I}, \mathcal{S}, \mathcal{A}>), \kappa = (\kappa^1, \kappa^2) \\ in \\ \begin{cases} \emptyset & if\ \kappa^1 = 0 \vee \kappa^2 = 0 \\ \{(X,Y),(Y,X) | Y \in \mathcal{V}_t\} & if\ \kappa = (1,1) \\ \{(X,Y),(Y,X) | Y \in \mathcal{V}_t\} \cup \{(X,X)\} & if\ \kappa = (1,2) \wedge \neg share(X,t,\mathcal{S}) \\ \{(X,Y),(Y,X) | Y \in \mathcal{V}_t\} \cup \mathcal{V}_t^2 & if\ \kappa = (2,1) \wedge \neg share(X,t,\mathcal{S}) \\ \{(X,Y),(Y,X) | Y \in \mathcal{V}_t\} \cup \{(X,X)\} \cup \mathcal{V}_t^2 & otherwise \end{cases} \end{cases}$$

$$\mathcal{A}' = \begin{cases} (\{(X,t),(t,X)\} \cup \mathcal{A})^* & if\ t \in \mathcal{VAR} \\ \mathcal{A} & otherwise \end{cases}$$

$$\mathcal{Z}' = \begin{cases} \emptyset & if \quad \mathbf{grd}(\mathcal{I}(X)) \wedge \forall Y \in vars(t).\mathbf{grd}(\mathcal{I}(Y)) \\ & \quad\ \vee\ \mathbf{var}(\mathcal{I}(X)) \wedge t \in \mathcal{VAR} \wedge \mathbf{var}(\mathcal{I}(t)) \\ \{X\} & if \quad \mathbf{var}(\mathcal{I}(X)) \\ vars(t) & if \quad t \in \mathcal{VAR} \wedge \mathbf{var}(\mathcal{I}(t)) \\ \{X\} \cup vars(t) & otherwise \end{cases}$$

$$compose(<\mathcal{I}, \mathcal{S}, \mathcal{A}>, <<\mathcal{I}', \mathcal{S}', \mathcal{A}'>, \mathcal{Z}'>, \mathcal{Z}_0) \stackrel{def}{=} <\mathcal{I}'', \mathcal{S}'', \mathcal{A}''> \tag{5.8}$$

where

$$\mathcal{I}'' = \lambda U. \begin{cases} \mathcal{I}'(U) & if\ \exists Y \in \mathcal{Z}_0.(U,Y) \in \mathcal{A} \\ \mathbf{ins}(\mathcal{I}'(U)) & if\ \exists Z \in \mathcal{Z}'.(U,Z) \in \mathcal{S} \\ \mathcal{I}'(U) & otherwise \end{cases} \tag{5.9}$$

$$\begin{aligned} \mathcal{S}'' = \ & \mathcal{S} \cup \mathcal{S}' \cup \\ & \{(U,Z) | \exists V.((U,V) \in \mathcal{S}' \wedge (V,Z) \in \mathcal{S})\} \cup \\ & \{(W,V) | \exists U.((W,U) \in \mathcal{S} \wedge (U,V) \in \mathcal{S}')\} \cup \\ & \{(W,Z) | \exists U,V.((W,U) \in \mathcal{S} \wedge (U,V) \in \mathcal{S}' \wedge (V,Z) \in \mathcal{S})\} \end{aligned} \tag{5.10}$$

$$\mathcal{A}'' = \mathcal{A}' \tag{5.11}$$

$$reduce(<\mathcal{I}, \mathcal{S}, \mathcal{A}>) \stackrel{def}{=} <\mathcal{I}, \left\{ (X,Y) \in \mathcal{S} \left| \begin{array}{l} \quad \neg\mathbf{grd}(\mathcal{I}(X)) \wedge \neg\mathbf{grd}(\mathcal{I}(Y))\} \\ \wedge \quad (X \equiv Y \rightarrow \neg\mathbf{var}(\mathcal{I}(X))) \end{array} \right. \right\}, \mathcal{A}> \tag{5.12}$$

solve solves a set of equations in an abstract substitution by calling *solve*0 to solve those equations whose right sides are variables, calling *failure* to detect failure in concrete unification, and then calling $solve0$ to solve those equations whose right sides are not variables. $solve0$ solves a set of equations in an abstract substitution by repeatedly calling $soln$ to solve one equation from the set of equations in the abstract substitution and updating the abstract substitution.

Let $\sigma^\flat \in Sub^\flat[\![\mathcal{W}]\!]$ and $\{X\} \cup vars(t) \subseteq \mathcal{W}$. The rationale of $soln(X = t, \sigma^\flat)$ is to ensure that, for any $\sigma \in \gamma[\![\mathcal{W}]\!](\sigma^\flat)$, if $\eta = mgu(X\sigma, t\sigma) \neq \mathit{fail}$ then $\sigma \circ \eta \in \gamma[\![\mathcal{W}]\!](soln(X = t, \sigma^\flat))$. The unification of $X\sigma$ and $t\sigma$ will possibly

(i) change the descriptions of the values of variables in $\{X\} \cup vars(t)$,

(ii) cause more pair sharing among variables in $\{X\} \cup vars(t)$, and

(iii) cause more aliasing among variables in $\{X\} \cup vars(t)$.

The composition of $\eta = mgu(X\sigma, t\sigma)$ to σ will possibly

(iv) change the descriptions of the values of variables in $\mathcal{W} \setminus (\{X\} \cup vars(t))$,

(v) cause more pair sharing among variables in $\mathcal{W} \setminus (\{X\} \cup vars(t))$ and/or more pair sharing between variables in $\{X\} \cup vars(t)$ and variables in $\mathcal{W} \setminus (\{X\} \cup vars(t))$, and

(vi) cause more aliasing among variables in $\mathcal{W} \setminus (\{X\} \cup vars(t))$ and/or more aliasing between variables in $\{X\} \cup vars(t)$ and variables in $\mathcal{W} \setminus (\{X\} \cup vars(t))$.

The effects of (i), (ii), (iii) and (vi) are estimated by a function $unieq$. $unieq$ also estimates part of the effect of (iv) - estimating the descriptions of the values of those variables in $\mathcal{W} \setminus (\{X\} \cup vars(t))$ which are aliased with variables in $\{X\} \cup vars(t)$ by σ. The effect of (v) is estimated by a function *compose*. *compose* also estimates part of the effect of (iv) - estimating the descriptions of the values of those variables in $\mathcal{W} \setminus (\{X\} \cup vars(t))$ which are not aliased with variables in $\{X\} \cup vars(t)$ by σ.

By lemmas 5.2.1 and 5.2.2, the variables outside $\{X\} \cup vars(t)$, the descriptions of the values of which are affected by the unification of $X\sigma$ and $t\sigma$, are those that share with variables in a subset of $\{X\} \cup vars(t)$. $unieq$ returns, in addition to an abstract substitution, that subset of $\{X\} \cup vars(t)$ which will be used by *compose* to update descriptions of values of variables outside $\{X\} \cup vars(t)$. Since the descriptions of the values of variables in $\{X\} \cup vars(t)$ and those aliased with variables in $\{X\} \cup vars(t)$ have been updated by $unieq$, *compose* takes $\{X\} \cup vars(t)$ as an input to avoiding updating their descriptions. Finally, $soln$ calls *reduce* to remove redundant sharing pairs. *reduce* uses the fact that a ground term does not share with other terms and the fact that a non-linear term can be neither a ground term nor a variable.

The following theorem states that algorithm 5.3.1 satisfies C4 (p. 87) when C4'-C8' hold. Some of the lemmas used in its proof are given afterwards.

Theorem 5.3.2 If C4'-C8' hold then

(C4) $unify^\sharp(A, \gamma[\![\mathcal{U}]\!](\theta^\flat), B, \gamma[\![\mathcal{V}]\!](\sigma^\flat)) \subseteq \gamma[\![\mathcal{V}]\!](unify^\flat(A, \theta^\flat, B, \sigma^\flat))$ for any finite $\mathcal{U}, \mathcal{V} \subseteq \mathcal{VAR}$, any $A \in \mathcal{ATOMS}$ with $vars(A) \subseteq \mathcal{U}$, any $B \in \mathcal{ATOMS}$ with $vars(B) \subseteq \mathcal{V}$, any $\theta^\flat \in Sub^\flat[\![\mathcal{U}]\!]$, and any $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$.

PROOF: See p. 174. ∎

Lemma 5.3.3 Let C4'-C8' hold, $\sigma^\flat \in Sub^\flat[\![\mathcal{W}]\!]$ and E be a set of equations in solved form such that $vars(E) \subseteq \mathcal{W}$. If $\sigma \in \gamma[\![\mathcal{W}]\!](\sigma^\flat)$ and $mgu(E\sigma) \neq fail$ then $\sigma \circ mgu(E\sigma) \in \gamma[\![\mathcal{W}]\!](solve(E, \sigma^\flat))$.

PROOF: See p. 174. ∎

Lemma 5.3.4 If C4'-C8' hold, $soln(X = t, <\mathcal{I}_i, \mathcal{S}_i, \mathcal{A}_i>) = <\mathcal{I}_o, \mathcal{S}_o, \mathcal{A}_o>$, $\sigma \in \gamma[\![\mathcal{W}]\!](<\mathcal{I}_i, \mathcal{S}_i, \mathcal{A}_i>)$, and $\eta = \sigma \circ mgu(\{X\sigma = t\sigma\}) \neq fail$ then $\eta \in \gamma[\![\mathcal{W}]\!](<\mathcal{I}_o, \mathcal{S}_o, \mathcal{A}_o>)$.

PROOF: See p. 174. ∎

To specialise algorithm 5.3.1 for an analysis, we need to ensure that C4' is satisfied by $<\check{D}, \check{\sqsubseteq}>$ and $\check{\gamma}$, and provide analysis-dependent functions **grd**, **var**, **ins** and **est**, making sure that they satisfy C5'-C8' respectively.

5.4 Mode Analysis

This section defines $<\check{D}, \check{\sqsubseteq}>$, $\check{\gamma}$, $\check{\epsilon}$, **grd**, **var**, **ins** and **est** for inferring mode information together with pair sharing and aliasing information, and proves that C1'-C8' hold.

A term is in mode ⓕ if it is a variable. A term is in mode ⓖ if it is a ground term. A term is in mode ⓞ if it is neither a variable nor a ground term. Let $\triangle = \{$ⓕ, ⓖ, ⓞ$\}$. Let $mo(t)$ be the mode of t. When a set of substitutions is approximated, a variable X is described by a subset of $\triangle$, called a mode description. For instance, if X is described by $\{$ⓕ, ⓖ$\}$ then X is instantiated to either a variable or a ground term.

Example 5.4.1 Let $\mathcal{P}$ be the following program and the possible goals are $\leftarrow a(X)\sigma$ such that $X\sigma$ is a variable.

```
a(Y) :- b(Y),c(Y,Z),d(Z).
b(W).
b(1).
c(U,U).
d(2).
```

∇	$\emptyset$	$\{\textcircled{f}\}$	$\{\textcircled{g}\}$	$\{\textcircled{o}\}$	$\{\textcircled{f},\textcircled{g}\}$	$\{\textcircled{f},\textcircled{o}\}$	$\{\textcircled{g},\textcircled{o}\}$	$\triangle$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{\textcircled{f}\}$	$\emptyset$	$\{\textcircled{f}\}$	$\{\textcircled{g}\}$	$\{\textcircled{o}\}$	$\{\textcircled{f},\textcircled{g}\}$	$\{\textcircled{f},\textcircled{o}\}$	$\{\textcircled{g},\textcircled{o}\}$	$\triangle$
$\{\textcircled{g}\}$	$\emptyset$	$\{\textcircled{g}\}$	$\{\textcircled{g}\}$	$\{\textcircled{g}\}$	$\{\textcircled{g}\}$	$\{\textcircled{g}\}$	$\{\textcircled{g}\}$	$\{\textcircled{g}\}$
$\{\textcircled{o}\}$	$\emptyset$	$\{\textcircled{o}\}$	$\{\textcircled{g}\}$	$\{\textcircled{g},\textcircled{o}\}$	$\{\textcircled{g},\textcircled{o}\}$	$\{\textcircled{g},\textcircled{o}\}$	$\{\textcircled{g},\textcircled{o}\}$	$\{\textcircled{g},\textcircled{o}\}$
$\{\textcircled{f},\textcircled{g}\}$	$\emptyset$	$\{\textcircled{f},\textcircled{g}\}$	$\{\textcircled{g}\}$	$\{\textcircled{g},\textcircled{o}\}$	$\{\textcircled{f},\textcircled{g}\}$	$\triangle$	$\{\textcircled{g},\textcircled{o}\}$	$\triangle$
$\{\textcircled{f},\textcircled{o}\}$	$\emptyset$	$\{\textcircled{f},\textcircled{o}\}$	$\{\textcircled{g}\}$	$\{\textcircled{g},\textcircled{o}\}$	$\triangle$	$\triangle$	$\{\textcircled{g},\textcircled{o}\}$	$\triangle$
$\{\textcircled{g},\textcircled{o}\}$	$\emptyset$	$\{\textcircled{g},\textcircled{o}\}$	$\{\textcircled{g}\}$	$\{\textcircled{g},\textcircled{o}\}$	$\{\textcircled{g},\textcircled{o}\}$	$\{\textcircled{g},\textcircled{o}\}$	$\{\textcircled{g},\textcircled{o}\}$	$\{\textcircled{g},\textcircled{o}\}$
$\triangle$	$\emptyset$	$\triangle$	$\{\textcircled{g}\}$	$\{\textcircled{g},\textcircled{o}\}$	$\triangle$	$\triangle$	$\{\textcircled{g},\textcircled{o}\}$	$\triangle$

Figure 5.1: Operator ∇.

We can infer that (1) Y is instantiated to either a variable or a ground term when an instance of $c(Y,Z)$ is selected, (2) Y and Z are instantiated to the same term which is either a variable or a ground term when an instance of $d(Z)$ is selected, and (3) Y and Z are instantiated to the same ground term after an instance of $d(Z)$ is successfully satisfied. ∎

Definition 5.4.2 Let $M \in \triangle$. Define

$$
\begin{array}{lclcl}
<\check{D},\ \check{\sqsubseteq}> & \stackrel{def}{=} & <\check{D}^m, \check{\sqsubseteq}^m> & \stackrel{def}{=} & <\wp(\triangle), \subseteq> \\
\check{\gamma}(M) & \stackrel{def}{=} & \check{\gamma}^m(M) & \stackrel{def}{=} & \{t \mid mo(t) \in M\} \\
\check{\epsilon} & \stackrel{def}{=} & \check{\epsilon}^m & \stackrel{def}{=} & \{\textcircled{f}\} \\
\mathbf{grd}(M) & \stackrel{def}{=} & \mathbf{grd}^m(M) & \stackrel{def}{=} & M = \{\textcircled{g}\} \\
\mathbf{var}(M) & \stackrel{def}{=} & \mathbf{var}^m(M) & \stackrel{def}{=} & M = \{\textcircled{f}\} \\
\mathbf{ins}(M) & \stackrel{def}{=} & \mathbf{ins}^m(M) & \stackrel{def}{=} & \left\{ \begin{array}{ll} \triangle & if\ \textcircled{f} \in M \\ \{\textcircled{o},\textcircled{g}\} & if\ \textcircled{o} \in M \\ \{\textcircled{g}\} & otherwise \end{array} \right.
\end{array}
$$

∎

Definition 5.4.3 Let $\nabla \in \wp(\triangle) \times \wp(\triangle) \mapsto \wp(\triangle)$ be defined as in figure 5.1. ∎

Given mode descriptions of two terms, ∇ over-estimates the mode description of the most general common instance of these two terms.

Lemma 5.4.4 Let t_1 and t_2 be terms. If $\eta = mgu(t_1, t_2) \neq \mathit{fail}$ then

$$mo(t_1\eta) = mo(t_2\eta) \in \{mo(t_1)\}\nabla\{mo(t_2)\}$$

PROOF: Each of $mo(t_1)$ and $mo(t_2)$ can assume one of three modes in $\triangle$. Therefore, there are nine cases to consider in all. Let $mo(t_1) = \circledcirc$ and $mo(t_2) = \circledcirc$. Then neither $mo(t_1\zeta)$ nor $mo(t_2\zeta)$ can be $\textcircled{f}$ for any substitution ζ. So $mo(t_1\eta) = mo(t_2\eta) \in \{\textcircled{g}, \circledcirc\} = \{mo(t_1)\}\nabla\{mo(t_2)\}$ according to figure 5.1. Other cases can be proved in a similar manner. ∎

Definition 5.4.5 Let $\mathcal{I}$ be an abstract instantiation state on $\mathcal{V}$ and t a term with $vars(t) \subseteq \mathcal{V}$.

$$mo^\flat(t, \mathcal{I}) \stackrel{def}{=} \begin{cases} \mathcal{I}(t) & if\ t \in \mathcal{V} \\ \emptyset & if\ \exists X \in vars(t).\mathcal{I}(X) = \emptyset \\ \{\textcircled{g}\} & if\ \forall X \in vars(t).\mathcal{I}(X) = \{\textcircled{g}\} \\ \{\textcircled{g}, \circledcirc\} & if\ \forall X \in vars(t).(\textcircled{g} \in \mathcal{I}(X)) \\ \{\circledcirc\} & otherwise \end{cases}$$

∎

For instance, $mo^\flat(f(X, h(Y)), \{X/\{\textcircled{g}\}, Y/\{\circledcirc\}, Z/\{\circledcirc\}\}) = \{\circledcirc\}$.

By definition 5.4.5, if $\mathcal{I}$ is an abstract instantiation state on $\mathcal{V}$ and t is a term with $vars(t) \subseteq \mathcal{V}$ then $mo(t\sigma) \in mo^\flat(t, \mathcal{I})$ for any $\sigma \in \gamma_i[\![\mathcal{V}]\!](\mathcal{I})$.

Lemma 5.4.6 Let X and U be variables, t be a term, $U \in vars(t)$, σ a substitution and $\eta = mgu(X\sigma, t\sigma) \neq \mathit{fail}$. Then (1) if $t = U$ is a variable then $mo(U\sigma\eta) \in \{mo(X\sigma)\}\nabla\{mo(t\sigma)\}$; and (2) if $t \neq U \wedge (mo(X\sigma) = \textcircled{g} \vee mo(t\sigma) = \textcircled{g})$ then $mo(U\sigma\eta) = \textcircled{g}$; and (3) if $t \neq U \wedge mo(X\sigma) \neq \textcircled{g} \wedge mo(t\sigma) \neq \textcircled{g}$ then $mo(U\sigma\eta) \in \triangle\nabla\{mo(U\sigma)\}$.

PROOF: See p. 175. ∎

Definition 5.4.7 Let $\mathcal{I}$ be an abstract instantiation state on $\mathcal{V}$, $X \in \mathcal{V}$ be a variable, and t be a term with $vars(t) \subseteq \mathcal{V}$. Let $\mathcal{V}_e = \{X\} \cup vars(t)$.

$$\begin{aligned} \mathbf{est}(X = t, \mathcal{I}) &\stackrel{def}{=} \mathbf{est}^m(X = t, \mathcal{I}) \\ &\stackrel{def}{=} \lambda U \in \mathcal{V}_e. \begin{cases} \mathcal{I}(X)\nabla mo^\flat(t, \mathcal{I}) & if\ t \equiv U \vee U \equiv X \\ \emptyset & if\ \mathcal{I}(X) = \emptyset \vee mo^\flat(t, \mathcal{I}) = \emptyset \\ \{\textcircled{g}\} & if\ \mathcal{I}(X) = \{\textcircled{g}\} \vee mo^\flat(t, \mathcal{I}) = \{\textcircled{g}\} \\ \triangle\nabla\mathcal{I}(U) & otherwise \end{cases} \end{aligned}$$

∎

Lemma 5.4.8 C1'-C8' holds.

PROOF: C1'-C7' are immediate from definition 5.4.2 and C8' is immediate from definition 5.4.5, lemma 5.4.6 and definition 5.4.7. ∎

Example 5.4.9 This example shows the computation of $unify^\flat(A, \theta^\flat, B, \sigma^\flat)$ by algorithm 5.3.1 for mode analysis. Let

$$\begin{array}{rcl}
\mathcal{U} & = & \{X, Y, Z\} \\
\mathcal{V} & = & \{K, L\} \\
A & = & t(X, Y, Z) \\
B & = & t(K, f(K, L), L) \\
\theta^\flat & = & <\{X/\{ⓖ\}, Y/\{ⓞ\}, Z/\{ⓕ\}\}, \{(Y, Y)\}, \{(X, X), (Y, Y), (Z, Z)\}> \\
\sigma^\flat & = & <\{K/\{ⓕ\}, L/\{ⓕ\}\}, \emptyset, \{(K, K), (L, L)\}>
\end{array}$$

In stage (1), $\Psi = \{X/X_0, Y/Y_0, Z/Z_0\}$ is applied to A and $\theta^\flat$.

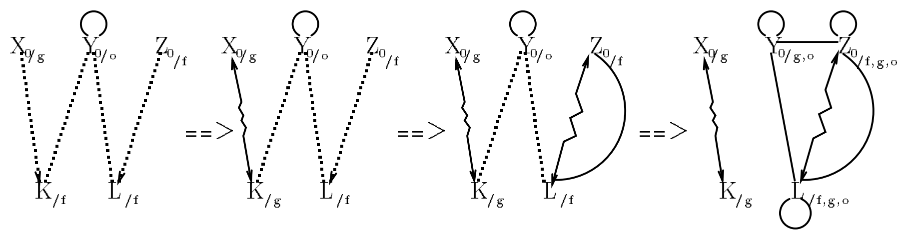
$$\begin{array}{rcl}
A\Psi & = & t(X_0, Y_0, Z_0) \\
\theta^\flat\Psi & = & <\{X_0/\{ⓖ\}, Y_0/\{ⓞ\}, Z_0/\{ⓕ\}\}, \\
& & \{(Y_0, Y_0)\}, \\
& & \{(X_0, X_0), (Y_0, Y_0), (Z_0, Z_0)\}>
\end{array}$$

$$E_0 = eq(mgu(A\Psi, B)) = \{X_0 = K, Y_0 = f(K, L), Z_0 = L\}$$

In stage (2),

$$\begin{array}{rl}
\zeta^\flat = \theta^\flat\Psi \uplus \sigma^\flat = & <\{X_0/\{ⓖ\}, Y_0/\{ⓞ\}, Z_0/\{ⓕ\}, K/\{ⓕ\}, L/\{ⓕ\}\}, \\
& \{(Y_0, Y_0)\}, \\
& \{(X_0, X_0), (Y_0, Y_0), (Z_0, Z_0), (K, K), (L, L)\}>
\end{array}$$

and, $\eta^\flat = solve(E_0, \zeta^\flat)$ is computed. This is accomplished by calling $soln$ via $solve0$ to solve one equation in E_0 after another with $\zeta^\flat$ as the constraint for the first equation and the intermediate results as the constraints for the other equations. This process is illustrated below by a sequence of graphs.



Each graph represents a pair of a set of equations and an abstract substitution. Variables with their mode descriptions are represented as nodes. Each pair in the pair sharing relation is represented as a solid edge without an arrow. A pair of distinct variables in the aliasing relation is represented as a solid edge with an arrow. The reflexive pairs of the aliasing relation are implicit. Each equation whose right side is a variable is represented as a dotted edge with an arrow. Each equation whose right side is a non-variable term is represented as a group of dotted edges without an arrow with each edge linking the variable on the left side to one variable in the term. The initial graph represents E_0 and $\zeta^\flat$. Each subsequent graph represents the set of the remaining equations to be solved and the abstract substitution resulting from solving an equation in the previous graph in the sequence. The final graph represents $\emptyset$ and $\eta^\flat$.

$$\begin{aligned}\eta^\flat = \ & <\{X_0/\{\textcircled{g}\}, Y_0/\{\textcircled{g}, \textcircled{o}\}, Z_0/\triangle, K/\{\textcircled{g}\}, L/\triangle\},\\ & \{(Y_0,Y_0),(Z_0,Z_0),(L,L),(Y_0,L),(L,Y_0),(Z_0,L),(L,Z_0),(Y_0,Z_0),(Z_0,Y_0)\},\\ & \left\{\begin{array}{l}(X_0,K),(K,X_0),(Z_0,L),(L,Z_0),\\ (X_0,X_0),(Y_0,Y_0),(Z_0,Z_0),(K,K),(L,L)\end{array}\right\}>\end{aligned}$$

In stage (3),

$$restrict(\eta^\flat, \mathcal{V}) = <\{K/\{\textcircled{g}\}, L/\triangle\}, \{(L,L)\}, \{(K,K),(L,L)\}>$$

So, $unify^\flat(A, \theta^\flat, B, \sigma^\flat) = <\{K/\{\textcircled{g}\}, L/\triangle\}, \{(L,L)\}, \{(K,K),(L,L)\}>$. ∎

5.5 Type Analysis

This section defines $<\check{D}, \check{\sqsubseteq}>$, $\check{\gamma}$, $\check{\epsilon}$, **grd**, **var**, **ins** and **est** for inferring type information together with pair sharing and aliasing information, and proves that C1'-C8' hold.

Types

A *type* is a finite expression denoting a possibly infinite set of values. Values in logic programs are terms. A type denotes a set of terms. Syntactically, a type is a term constructible from a set $\mathcal{CONS}$ of *type constructors* and a denumerable set $\mathcal{PARS}$ of *type parameters*. If a type constructor has an arity of zero then it is called a *base type*. There are two special base types *any* and *empty*, denoting $\mathcal{TERMS}$ and $\emptyset$ respectively. Recall that $\mathcal{MTYPES}$ is the set of mono-types constructible from $\mathcal{CONS}$. Mono-types in $\mathcal{MTYPES}$ can be compared according to the partial order defined in the following.

Definition 5.5.1 Let $M, M_1, \cdots, M_m, M'_1, \cdots, M'_m$ be mono-types and c/m be a type constructor.

$$\begin{array}{rcll} empty & \check{\sqsubseteq}^t & M & \\ M & \check{\sqsubseteq}^t & any & \\ c(M_1, \cdots, M_m) & \check{\sqsubseteq}^t & c(M'_1, \cdots, M'_m) & \stackrel{def}{=} \quad \forall 1 \leq j \leq m.(M_j \check{\sqsubseteq}^t M'_j) \end{array}$$

∎

Example 5.5.2 Let $\mathcal{CONS} = \{nat/0, any/0, empty/0, list/1\}$. We have $empty \check{\sqsubseteq}^t nat$, $list(nat) \check{\sqsubseteq}^t any$, $list(empty) \check{\sqsubseteq}^t list(nat)$ and $nat \check{\not\sqsubseteq}^t list(nat)$. ∎

Lemma 5.5.3 $< \mathcal{MTYPES}, \check{\sqsubseteq}^t, empty, any, \check{\sqcap}^t, \check{\sqcup}^t >$ is a complete lattice where, letting $T = c(T_1, \cdots, T_m)$ and $T' = c'(T'_1, \cdots, T'_k)$ be mono-types, the least upper bound operator $\check{\sqcup}^t$ is

$$\begin{array}{rclcl} empty \check{\sqcup}^t T & = & T \check{\sqcup}^t empty & = & T \\ any \check{\sqcup}^t T & = & T \check{\sqcup}^t any & = & any \\ T \check{\sqcup}^t T' & = & T' \check{\sqcup}^t T & = & \begin{cases} c(T_1 \check{\sqcup}^t T'_1, \cdots, T_m \check{\sqcup}^t T'_m) & if\ c/m = c'/k \\ any & otherwise \end{cases} \end{array}$$

and the greatest lower bound operator $\check{\sqcap}^t$ is

$$\begin{array}{rclcl} empty \check{\sqcap}^t T & = & T \check{\sqcap}^t empty & = & empty \\ any \check{\sqcap}^t T & = & T \check{\sqcap}^t any & = & T \\ T \check{\sqcap}^t T' & = & T' \check{\sqcap}^t T & = & \begin{cases} c(T_1 \check{\sqcap}^t T'_1, \cdots, T_m \check{\sqcap}^t T'_m) & if\ c/m = c'/k \\ empty & otherwise \end{cases} \end{array}$$

Proof: Immediate from the definition 5.5.1. ∎

A type is a general type if it is either a base type except *empty* or of the form $c(\beta_1, \cdots, \beta_m)$ where c/m is a type constructor and $\beta_1, \cdots, \beta_m \in \mathcal{PARS}$ are different type parameters. For instance, $list(\beta)$ is a general type while $list(nat)$ and $list(list(\beta))$ are not. The types of terms and the sets of terms denoted by them are both decided by type definitions for function symbols in Σ.

Definition 5.5.4 A type clause is a clause for predicate $:/2$ which is of the form

$$f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$$

where $m, n \geq 0$, f/n is a function symbol, c/m is a type constructor, $\beta_1, \cdots, \beta_m$ are different type parameters, $X_1, \cdots, X_n$ are different program variables, and, for $1 \leq \imath \leq n$, $T_\imath$ is either a general type or a type parameter. The above type clause relates the type of $f(t_1, \cdots, t_n)$ to the types of $t_1, \cdots, t_n$.

In addition to the above restriction on the form of a type clause, we also require that

(i) each function symbol f/n is typed by exactly one type clause; and

(ii) $vars(T_\jmath) \subseteq \{\beta_1, \cdots, \beta_m\}$ for $1 \leq \jmath \leq n$.

(i) disallows *overloading* [12] and (ii) is often referred to as *type preserving* [131]. ∎

If a function symbol f/n is not typed explicitly by a type clause, it is assumed that it has the following default type definition.

$$f(X_1, \cdots, X_n) : f(\beta_1, \cdots, \beta_n) \leftarrow X_1 : \beta_1, \cdots, X_n : \beta_n$$

Note that f/n is a type constructor as well as a function symbol. The default type clause means that the type of $f(t_1, \cdots, t_n)$ is $f(T_1, \cdots, T_n)$ if the types of $t_1, \cdots, t_n$ are $T_1, \cdots, T_n$ respectively.

Example 5.5.5 This example illustrates some type definitions.

$$\begin{array}{rcll} 0 : nat & \leftarrow & & \%\ type\ clause\ D1 \\ s(X) : nat & \leftarrow & X : nat & \%\ type\ clause\ D2 \\ [\,] : list(\beta') & \leftarrow & & \%\ type\ clause\ D3 \\ [H|T] : list(\beta'') & \leftarrow & H : \beta'', T : list(\beta'') & \%\ type\ clause\ D4 \end{array}$$

D1 says that 0 is a member of type nat. D2 means that if X has type nat then $s(X)$ has type nat. D1 and D2 also gives the set of terms denoted by type nat. D3 states that [ ] is a member of type $list(\beta')$ for any type β' while D4 reads that, for any type β'', if H is a member of type β'' and T a member of type $list(\beta'')$ then $[H|T]$ is a member of type $list(\beta'')$. ∎

A type-substitution κ is a set of formulae of the form β/T where β is a type parameter and T is a type. If $\kappa = \{\beta_1/T_1, \cdots, \beta_n/T_n\}$ then the domain $dom(\kappa)$ of κ is $\{\beta_1, \cdots, \beta_n\}$. κ is a mono-type-substitution if $T_1, \cdots, T_n$ are mono-types. Mono-type-substitutions on the same domain are partially ordered by the point-wise extension of the partial order $\check{\sqsubseteq}^t$ on mono-types. The application of κ to a type is to replace simultaneously each occurrence of each β_i for $1 \leq i \leq n$ in that type with T_i. For instance, $list(\beta_1)\{\beta_1/list(nat), \beta_2/nat\} = list(list(nat))$.

Normalised Types

The size of $\mathcal{MTYPES}$ is infinite when $\mathcal{CONS}$ contains type constructors that are not base types. In order to have a finite abstract domain for type analysis, it is necessary to use mono-types from a finite subset of $\mathcal{MTYPES}$, called normalised types, to describe terms. This is achieved by a normalisation function norm from $\mathcal{MTYPES}$ to the finite sub-set of $\mathcal{MTYPES}$. The normalisation functions we consider are the depth abstraction functions [115] and the stump abstraction functions [142] that are described in the following.

Let $M \in \mathcal{MTYPES}$. M is a depth 0 sub-mono-type of M. Let $M = c(M_1, \cdots, M_m)$ and $k > 0$. M' is a depth k sub-mono-type of M if M' is a depth $(k-1)$ sub-mono-type of M_j for some $1 \leq j \leq m$. A depth k abstraction function $\mathsf{depth}[\![k]\!]$, when applied to a mono-type M, substitutes *any* for each depth k sub-mono-type of M that is not a base type. For instance,

$$\mathsf{depth}[\![2]\!](c(g(h(nat), g(nat, int)), h(c(h(nat), c(nat, int))))) = c(g(any, any), h(any))$$

$$\mathsf{depth}[\![k]\!](M) = \begin{cases} M & \mathit{if\ M\ is\ a\ base\ type} \\ any & \mathit{if}\ k = 0 \\ c(\mathsf{depth}[\![k-1]\!](M_1), \cdots, \mathsf{depth}[\![k-1]\!](M_m)) & \mathit{if}\ M = c(M_1, \cdots, M_m) \end{cases} \tag{5.13}$$

Let $M \in \mathcal{MTYPES}$. M corresponds to a tree, called the tree for M. If M is a base type then the tree for M is a single node labelled with M. If $M = c(M_1, \cdots, M_m)$ then the root node of the tree for M, labelled with c/m, has m child nodes which are the root nodes of the trees for $M_1, \cdots, M_m$ respectively. A path is denoted as the sequence of the labels of the nodes along the path, and the empty path is denoted by ε. Let w be a path and c/m a type constructor. The number of the occurrences of c/m in w is denoted as $noc(c/m, w)$.

Let $M', M \in \mathcal{MTYPES}$ and the tree for M' be a sub-tree of the tree for M, c/m be the label of the root node of the tree for M', and w be the path from, and including, the root node of the tree for M to, but excluding, the root node of the tree for M'. The repetition-depth of M' in M is defined as $noc(c/m, w)$. For instance, in $c(g(h(nat), g(nat, int)), h(c(h(nat), c(nat, int))))$, the repetition-depth of $c(nat, int)$ is 2, and that of $g(nat, int)$ is 1. A stump k abstraction function $\mathsf{stump}[\![k]\!]$, when applied to a mono-type M, substitutes *any* for each sub-mono-type M' of M whose repetition-depth in M is k. For instance,

$$\mathsf{stump}[\![1]\!](c(g(h(nat), g(nat, int)), h(c(h(nat), c(nat, int))))) = c(g(h(nat), any), h(any))$$

$$\mathsf{stump}[\![k]\!](M) = \mathsf{st}[\![k]\!][\![\varepsilon]\!](M) \tag{5.14}$$

$$\mathsf{st}[\![k]\!][\![w]\!](M) =$$

$$\begin{cases} M & if\ M\ is\ a\ base\ type \\ c(\mathsf{st}[\![k]\!][\![w(c/m)]\!](M_1), \cdots, \mathsf{st}[\![k]\!][\![w(c/m)]\!](M_m)) & if\quad M = c(M_1, \cdots, M_m) \\ & \quad \wedge\ noc(c/m, w) < k \\ any & otherwise \end{cases}$$

$\mathsf{depth}[\![k]\!]$ and $\mathsf{stump}[\![k]\!]$ defines two classes of normalisation functions. Choosing different values for k in $\mathsf{depth}[\![k]\!]$ or $\mathsf{stump}[\![k]\!]$ will result in different depth or stump abstraction functions. norm will be used to refer to $\mathsf{depth}[\![k]\!]$ or $\mathsf{stump}[\![k]\!]$ for some $k \geq 0$.

Definition 5.5.6 $<\check{D},\ \check{\sqsubseteq}\ > \stackrel{\text{def}}{=} <\check{D}^t, \check{\sqsubseteq}^t > \stackrel{\text{def}}{=} < \{\mathsf{norm}(M) \mid M \in \mathcal{MTYPES}\}, \check{\sqsubseteq}^t >$ ∎

Lemma 5.5.7 For any $\mathsf{norm} \in \{\mathsf{depth}[\![k]\!], \mathsf{stump}[\![k]\!] \mid k \geq 0\}$,

(a) norm is monotonic.

(b) $\forall M \in \mathcal{MTYPES}.(M\ \check{\sqsubseteq}^t\ \mathsf{norm}(M))$.

(c) C1' holds.

Proof: (a) and (b) are immediate from equations 5.13 and 5.14. (c) is immediate from equations 5.13 and 5.14 and lemma 5.5.3. ∎

Substituting different functions from $\{\mathsf{depth}[\![k]\!] \mid k \geq 0\} \cup \{\mathsf{stump}[\![k]\!] \mid k \geq 0\}$ for norm will result in different abstract domains and therefore different type analyses. Choosing $\mathsf{depth}[\![k]\!]$ for norm and using default type definitions for all the function symbols, the type analysis becomes depth k analysis [115]. Choosing $\mathsf{stump}[\![k]\!]$ for norm and using default type definitions for all the function symbols, the type analysis becomes stump k analysis [142].

Definition 5.5.8 The meaning of a mono-type is defined as follows.

$$\check{\gamma}(M) \stackrel{def}{=} \check{\gamma}^t(M) \stackrel{def}{=} \tag{5.15}$$

$$\begin{cases} \emptyset & if\ M = empty \\ \mathcal{TERMS} & if\ M = any \\ \left\{ f(t_1, \cdots, t_n) \left| \begin{array}{c} f(X_1,\cdots,X_n){:}c(\beta_1,\cdots,\beta_m) \leftarrow X_1{:}T_1,\cdots,X_n{:}T_n \\ \kappa = \{\beta_j/M_j \mid 1 \leq j \leq m\} \\ \forall 1 \leq i \leq n.(t_i \in \check{\gamma}^t(T_i\kappa)) \end{array} \right. \right\} & if\ M = c(M_1, \cdots, M_m) \end{cases}$$

In the above equation, $\kappa = \{\beta_j/M_j | 1 \leq j \leq m\}$ is a type-substitution. Since M_j for $1 \leq j \leq m$ is a mono-type and $\{\beta_1, \cdots, \beta_m\}$ contains all the type parameters in T_i for $1 \leq i \leq n$, $T_i\kappa$ is a mono-type for $1 \leq i \leq n$. ∎

Example 5.5.9 Let $\Sigma = \{0/0, s/1, [\,]/0, ./2\}$, $\mathcal{CONS} = \{nat/0, any/0, empty/0, list/1\}$, $\mathcal{VAR} = \{X, Y, \cdots\}$, and the type definitions for function symbols in Σ be those in example 5.5.5.

$$\begin{aligned} \check{\gamma}^t(list(empty)) &= \{[\,]\} \\ \check{\gamma}^t(list(nat)) &= \{[\,], [0], \cdots, [0, 0], \cdots\} \\ \check{\gamma}^t(list(any)) &= \{[\,], [0], \cdots, [X], \cdots, [Y], \cdots\} \end{aligned}$$

∎

Lemma 5.5.10 C2' holds.

PROOF: See p. 175. ∎

Lemma 5.5.11 Let M be a mono-type. If *any* does not occur in M then t is a ground term for any $t \in \check{\gamma}^t(M)$.

PROOF: See p. 176. ∎

Lemma 5.5.12 $t \in \check{\gamma}^t(M) \to t\theta \in \check{\gamma}^t(M)$ for any $M \in \mathcal{MTYPES}$, any $t \in \mathcal{TERMS}$ and any $\theta \in Sub$.

PROOF: See p. 176. ∎

Definition 5.5.13

$$\begin{array}{lclcl} \check{\epsilon} & \stackrel{\text{def}}{=} & \check{\epsilon}^t & \stackrel{\text{def}}{=} & any \\ \mathbf{grd}(M) & \stackrel{\text{def}}{=} & \mathbf{grd}^t(M) & \stackrel{\text{def}}{=} & any\ does\ not\ occur\ in\ M \\ \mathbf{var}(M) & \stackrel{\text{def}}{=} & \mathbf{var}^t(M) & \stackrel{\text{def}}{=} & false \\ \mathbf{ins}(M) & \stackrel{\text{def}}{=} & \mathbf{ins}^t(M) & \stackrel{\text{def}}{=} & M \end{array}$$

∎

Lemma 5.5.14 C3'-C7' hold.

PROOF: C3' is immediate from equation 5.15. C4' is implied by lemma 5.5.12. C5' is implied by lemma 5.5.11. C6' is vacuous. C7' is implied by lemma 5.5.12. ∎

Definition 5.5.15 Let M be a mono-type and G a general type or a type parameter. Define

$$lmts(M,G) \stackrel{def}{=} \begin{cases} \emptyset & if\ G = any \\ \{\beta/empty \mid \beta \in vars(G)\} & if\ M = empty \\ \{G/M\} & if\ G\ is\ a\ type\ parameter \\ \bigcup_{1\le j\le m}\{\beta_j/M_j\} & if\quad M = c(M_1,\cdots,M_m) \\ & \quad \wedge\ G = c(\beta_1,\cdots,\beta_m) \\ fail & otherwise \end{cases}$$

∎

If $lmts(M,G) \neq fail$ then $lmts(M,G)$ is the least mono-type-substitution under which the instance of G is greater than M with respect to $\check{\sqsubseteq}^t$.

Example 5.5.16 $\mathcal{CONS} = \{nat/0, any/0, empty/0, list/1, pair/2\}$ and $\beta, \beta_1, \beta_2 \in \mathcal{PARS}$.

$$\begin{aligned} lmts(pair(nat, list(any)), pair(\beta_1,\beta_2)) &= \{\beta_1/nat, \beta_2/list(any)\} \\ lmts(list(any), pair(\beta_1,\beta_2)) &= fail \end{aligned}$$

∎

Lemma 5.5.17 If M, M' are mono-types, G is a general type or a type parameter, and $\kappa, \kappa', \kappa''$ are mono-type-substitutions such that

$$M \check{\sqsubseteq}^t M', \quad \kappa = lmts(M,G), \quad \kappa' = lmts(M',G)$$

then

(a) $\kappa \neq fail \rightarrow M \check{\sqsubseteq}^t G\kappa$.

(b) $\kappa = fail \rightarrow \kappa' = fail$.

(c) $\kappa \neq fail \rightarrow (\kappa' = fail \vee \forall\beta \in vars(G).(\beta\kappa \check{\sqsubseteq}^t \beta\kappa'))$.

(d) $M \check{\sqsubseteq}^t G\kappa'' \rightarrow (\kappa \neq fail \wedge \forall\beta \in vars(G).(\beta\kappa \check{\sqsubseteq}^t \beta\kappa''))$.

PROOF: Immediate from definitions 5.5.15 and 5.5.1 ∎

A variable-typing μ is a set of formulae of the form X/T where X is a variable in $\mathcal{VAR}$ and T is a type. Let $\mu = \{X_1/T_1, \cdots, X_n/T_n\}$. The domain $dom(\mu)$ of μ is $\{X_1, \cdots, X_n\}$. μ is a mono-variable-typing if T_i is a mono-type for each $1 \le i \le n$. Mono-variable-typings on the same domain are partially ordered by the point-wise extension of the partial order $\check{\sqsubseteq}^t$ on mono-types.

Definition 5.5.18 Let t be a term and μ a mono-variable-typing such that $vars(t) \subseteq dom(\mu)$. Define

$$\breve{\alpha}^t(t) \stackrel{def}{=} \breve{A}^t(t, \{X/any \mid X \in vars(t)\}) \tag{5.16}$$

$$\breve{A}^t(t,\mu) \stackrel{def}{=} \begin{cases} \mu(t) & if\ t\ is\ a\ variable \\ c(\beta_1, \cdots, \beta_m)\kappa & if \quad t = f(t_1, \cdots, t_n) \\ & \quad \wedge \ f(X_1,\cdots,X_n){:}c(\beta_1,\cdots,\beta_m) \leftarrow X_1{:}T_1,\cdots,X_n{:}T_n \\ & \quad \wedge \ \kappa_i = lmts(\breve{A}^t(t_i,\mu), T_i) \\ & \quad \wedge \ \forall 1 \leq i \leq n. \kappa_i \neq \textsf{fail} \\ & \quad \wedge \ \kappa = \bigcup_{1 \leq j \leq m} \{\beta_j / (\check{\bigsqcup}^t_{\substack{1 \leq i \leq n \\ \beta_j \in dom(\kappa_i)}} \{\beta_j \kappa_i\})\} \\ any & otherwise \end{cases} \tag{5.17}$$

∎

For any $\theta \in Sub$, $t\theta$ is of type $\breve{A}^t(t,\mu)$ if, for each variable $X \in vars(t)$, $X\theta$ is of type $\mu(X)$. For any $\theta \in Sub$, $t\theta$ is of type $\breve{\alpha}^t(t)$.

Example 5.5.19 Let the type definitions be those in example 5.5.5. Consider the computation of $\breve{\alpha}^t([0,X]) = \breve{A}^t([0,X], \{X/any\})$ which involves the following steps.

(1) $\breve{A}^t(0, \{X/any\}) = nat$;

(2) $lmts(nat, \beta'') = \{\beta''/nat\}$;

(3) compute $\breve{A}^t([X], \{X/any\})$ as follows;

  (i) $\breve{A}^t(X, \{X/any\}) = any$ since X is a variable;

  (ii) $lmts(any, \beta'') = \{\beta''/any\}$;

  (iii) $\breve{A}^t([\,], \{X/any\}) = list(\beta')\{\beta'/\check{\bigsqcup}^t \emptyset\} = list(\beta')\{\beta'/empty\} = list(empty)$;

  (iv) $lmts(list(empty), list(\beta'')) = \{\beta''/empty\}$;

  (v) $\breve{A}^t([X], \{X/any\}) = list(\beta'')\{\beta''/(any \check{\sqcup}^t empty)\} = list(any)$.

(4) $lmts(list(any), list(\beta'')) = \{\beta''/any\}$;

(5) $\breve{A}^t([0,X], \{X/any\}) = list(\beta'')\{\beta''/(nat \check{\sqcup}^t any)\} = list(any)$.

∎

Lemma 5.5.20 Let t be a term and μ, ν be mono-variable-typings. If $vars(t) \subseteq dom(\mu)$, $vars(t) \subseteq dom(\nu)$ and $\forall X \in vars(t).(\mu(X) \check{\sqsubseteq}^t \nu(X))$ then $\check{A}^t(t,\mu) \check{\sqsubseteq}^t \check{A}^t(t,\nu)$.

PROOF: See p. 176. ∎

Lemma 5.5.21 For any term t, any substitution θ and any mono-variable-typing μ,

(a) $vars(t) \subseteq dom(\mu) \wedge \forall X \in vars(t).\check{\alpha}^t(X\theta) \check{\sqsubseteq}^t \mu(X) \rightarrow \check{\alpha}^t(t\theta) \check{\sqsubseteq}^t \check{A}^t(t,\mu)$.

(b) $\check{\alpha}^t(t\theta) \check{\sqsubseteq}^t \check{\alpha}^t(t)$.

PROOF: By equations 5.16 and 5.17,

$$\begin{array}{rcl} \check{\alpha}^t(t) & = & \check{A}^t(t, \{X/any \mid X \in vars(t)\}) \\ \check{\alpha}^t(t\theta) & = & \check{A}^t(t, \{X/\check{\alpha}^t(X\theta) \mid X \in vars(t)\}) \end{array}$$

Therefore, (a) and (b) hold by lemma 5.5.20. ∎

Lemma 5.5.22 For any term t and any mono-type M, $t \in \check{\gamma}^t(M) \leftrightarrow \check{\alpha}^t(t) \check{\sqsubseteq}^t M$.

PROOF: See p. 177. ∎

Definition 5.5.23 Let t be a term and M be a mono-type. Define

$$gmvt(M,t) \stackrel{def}{=} \tag{5.18}$$
$$\begin{cases} \{X/M \mid X \in vars(t)\} & if\ M = any\ or\ t\ is\ a\ variable \\ \bigcup_{X \in vars(t)} \{X/(\check{\sqcap}^t_{\substack{1 \le i \le n \\ X \in dom(\mu_i)}} \{\mu_i(X)\})\} & if \quad t = f(t_1, \cdots, t_n) \\ & \quad \wedge\ f(X_1,\cdots,X_n){:}c(\beta_1,\cdots,\beta_m) \leftarrow X_1{:}T_1,\cdots,X_n{:}T_n \\ & \quad \wedge\ M = c(M_1, \cdots, M_m) \\ & \quad \wedge\ \kappa = \{\beta_j/M_j | 1 \le j \le m\} \\ & \quad \wedge\ \forall 1 \le i \le n.(\mu_i = gmvt(T_i\kappa, t_i)) \\ \{X/empty \mid X \in vars(t)\} & otherwise \end{cases}$$

∎

For any $\theta \in Sub$ and any variable $X \in vars(t)$, $X\theta$ must be of type $gmvt(M,t)(X)$ in order for $t\theta$ to be of type M.

Example 5.5.24 Let the type definitions be those in example 5.5.5. The computation of $gmvt(list(nat), [X,Y])$ involves the following steps and makes use of the type definitions D3 and D4.

(1) $gmvt(nat, X) = \{X/nat\}$;

(2) compute $gmvt(list(nat), [Y])$;

(i) $gmvt(nat, Y) = \{Y/nat\}$;

(ii) $gmvt(list(nat), [\,]) = \emptyset$;

(iii) $gmvt(list(nat), [Y]) = \{Y/nat\}$.

(3) $gmvt(list(nat), [X, Y]) = \{X/nat, Y/nat\}$.

∎

Example 5.5.25 Let the type definitions be those in example 5.5.5. The computation of $gmvt(list(nat), [X, Y|X])$ involves the following steps and makes use of the type definitions D3 and D4.

(1) $gmvt(nat, X) = \{X/nat\}$;

(2) compute $gmvt(list(nat), [Y|X])$;

(i) $gmvt(nat, Y) = \{Y/nat\}$;

(ii) $gmvt(list(nat), X) = \{X/list(nat)\}$;

(iii) $gmvt(list(nat), [Y|X]) = \{X/list(nat), Y/nat\}$.

(3) $gmvt(list(nat), [X, Y|X]) = \{X/empty, Y/nat\}$.

∎

Lemma 5.5.26 Let t be a term, θ a substitution, M a mono-type and $\nu = gmvt(M, t)$.

$$\check{\alpha}^t(t\theta) \check{\sqsubseteq}^t M \rightarrow \forall X \in vars(t).(\check{\alpha}^t(X\theta) \check{\sqsubseteq}^t \nu(X))$$

Proof: See p. 178. ∎

Definition 5.5.27 Let μ be a mono-variable-typing on $\mathcal{V}$, $X \in \mathcal{V}$ and $vars(t) \subseteq \mathcal{V}$

$$
\begin{array}{rcl}
\mathbf{est}(X = t, \mu) & \stackrel{def}{=} & \mathbf{est}^t(X = t, \mu) \\
& \stackrel{def}{=} & \left\{
\begin{array}{l}
let\ \mathcal{V}_t = vars(t), \\
\quad \mu_1 = gmvt(\mu(X), t), \\
\quad \mu_2 = \{Y/(\mu(Y) \check{\sqcap}^t \mu_1(Y)) \mid Y \in \mathcal{V}_t\} \\
in \\
\left\{
\begin{array}{ll}
\mu(Y) \check{\sqcap}^t \mathsf{norm}(\check{A}^t(t, \mu_2)) & if\ Y \equiv X \\
\mu_2(Y) & if\ Y \in \mathcal{V}_t
\end{array}
\right.
\end{array}
\right.
\end{array}
\tag{5.19}
$$

■

Lemma 5.5.28 C8' holds.

PROOF: See p. 179. ■

Example 5.5.29 This example shows the computation of $unify^\flat(A, \theta^\flat, B, \sigma^\flat)$ by algorithm 5.3.1 for type analysis. Let the type definitions be those in example 5.5.5 and

$$
\begin{array}{rcl}
\mathcal{U} & = & \{W, Y, Z\} \\
\mathcal{V} & = & \{X, W, Y, Z\} \\
A & = & ap(W, Y, Z) \\
B & = & ap([X|W], Y, [X|Z]) \\
\theta^\flat & = & <\{W/list(nat), Y/list(nat), Z/any\}, \{(Z, Z)\}, \{(W, W), (Y, Y), (Z, Z)\}> \\
\sigma^\flat & = & <\{W/any, X/any, Y/any, Z/any\}, \emptyset, \{(W, W), (X, X), (Y, Y), (Z, Z)\}>
\end{array}
$$

In stage (1), $\Psi = \{W/W_0, Y/Y_0, Z/Z_0\}$ is applied to A and $\theta^\flat$.

$$
\begin{array}{rcl}
A\Psi & = & ap(W_0, Y_0, Z_0) \\
\theta^\flat\Psi & = & <\{W_0/list(nat), Y_0/list(nat), Z_0/any\}, \{(Z_0, Z_0)\}, \{(W_0, W_0), (Y_0, Y_0), (Z_0, Z_0)\}> \\
E_0 & = & eq(mgu(A\Psi, B)) = \{W0 = [X|W], Y0 = Y, Z0 = [X|Z]\}
\end{array}
$$

In stage (2),

$$
\begin{array}{rl}
\zeta^\flat = \theta^\flat\Psi \uplus \sigma^\flat = & <\{W_0/list(nat), Y_0/list(nat), Z_0/any, W/any, X/any, Y/any, Z/any\}, \\
& \{(Z_0, Z_0)\}, \\
& \{(W_0, W_0), (Y_0, Y_0), (Z_0, Z_0), (W, W), (X, X), (Y, Y), (Z, Z)\}>
\end{array}
$$

and, $\eta^\flat = solve(E_0, \zeta^\flat)$ is computed. This is accomplished by calling $soln$ via $solve0$ to solve one equation in E_0 after another with $\zeta^\flat$ as the constraint for the first equation and the intermediate results as the constraints for other equations.

$$
\begin{aligned}
\eta^\flat = \ & <\{W0/list(nat), Y0/list(nat), Z0/any, W/list(nat), X/any, Y/list(nat), Z/any\}, \\
& \{(Z_0, Z_0), (Z_0, X), (Z_0, Z), (X, Z_0), (X, X), (X, Z), (Z, Z_0), (Z, X), (Z, Z)\} \\
& \{(W_0, W_0), (Y_0, Y_0), (Z_0, Z_0), (W, W), (X, X), (Y, Y), (Z, Z), (Y_0, Y), (Y, Y_0)\}>
\end{aligned}
$$

In stage (3),

$$
\begin{aligned}
unify^\flat(A, \theta^\flat, B, \sigma^\flat) = restrict(\eta^\flat, \mathcal{V}) = \ & <\{W/list(nat), X/any, Y/list(nat), Z/any\}, \\
& \{(X, X), (X, Z), (Z, X), (Z, Z)\} \\
& \{(W, W), (X, X), (Y, Y), (Z, Z)\}>
\end{aligned}
$$

■

5.6 Summary

We have developed a generic abstract domain and a generic abstract unification algorithm for a class of analyses which infer pair sharing and aliasing information together with descriptions of the values of variables. To specialise the generic abstract domain and the generic abstract unification algorithm for an analysis, it is sufficient to design a complete lattice $<\check{D}, \ \check{\sqsubseteq}\ >$ and a concretisation function $\check{\gamma}$, identify $\check{\epsilon} \in \check{D}$, develop analysis-dependent functions **grd**, **var**, **ins** and **est**, and prove that they satisfy C1'-C8'.

We have used Jones and Søndergaard's abstract domain for sharing and non-linearity [60]. An alternative is to use Sundararajan and Conery's abstract domain [126] for sharing and non-linearity which is Jacobs and Langen's abstract domain for set sharing [55] augmented with a non-linearity component. Sundararajan and Conery's abstract domain captures sharing information more precisely than Jones and Søndergaard's. Sundararajan and Conery's abstract domain for sharing and non-linearity leads to abstract unification algorithms of exponential complexity [126] while Jones and Søndergaard's leads to abstract unification algorithms of polynomial complexity [15].

The core function *solve* of the generic abstract unification algorithm solves a set of equations in an abstract unification by calling *soln* via *solve*0 to solve one equation after another. This process can be thought of as an abstraction of the concrete unification process.

Mode information is derived together with pair sharing and aliasing information. Sharing information is necessary to ensure the correctness of the abstract unification algorithm while aliasing information is used to increase the accuracy of mode analysis. All the mode descriptions but **iu** in the literature can be expressed in mode descriptions in $\wp(\triangle)$. **e** is expressed by $\emptyset$, **f** by $\{\textcircled{f}\}$, **g** by $\{\textcircled{g}\}$, **nf** by $\{\textcircled{g}, \textcircled{o}\}$ and **a** by $\triangle$.

The work reported by Codish, Dams and Yardeni in [15] developed an abstract unification algorithm for inferring groundness and pair sharing information by abstracting the concrete unification process. However, there are significant differences between our work and theirs. Firstly, they used one of $\{\mathbf{g}, \mathbf{a}\}$ as the mode description of a variable. So, only groundness information is captured. In this case, sharing information is not necessary for the purpose of safeness but is used to increase the accuracy of mode analysis. Since **a** is a safe mode description of any term, it is not necessary to update the mode description of any other variable when the mode description of one variable changes even if the two variables might share. Secondly, in [15], given a set of equations and an abstract substitution, an equation which is ground (more instantiated) in the abstract substitution is solved before an equation that is not ground (less instantiated) in the abstract substitution. We do not solve equations in a particular instantiation order because it is not possible to find an instantiation order on the mode descriptions that we have used. Indeed, ∇ is not a least bound operator since $\{\textcircled{o}\}\nabla\{\textcircled{o}\} \neq \{\textcircled{o}\}$. To exploit an instantiation order, it is necessary to use a subset of $\wp(\triangle)$ as the set of mode descriptions such as that in [24].

Type information is also derived together with pair sharing and aliasing information. The idea to incorporate type definitions in type analysis originates from the work by Horiuchi and Kanamori [52, 62]. The abstract domain we have presented is richer than that used by Horiuchi and Kanamori who used a mono-variable-typing to approximate a set of substitutions. They did not address the problem of an infinite abstract domain. We use a normalisation function choosing from two classes to enforce a finite abstract domain. This problem can also be resolved by designing a pair of widening/narrowing operators [22]. Horiuchi and Kanamori required that each object function symbol is typed by a type definition. We relax this requirement by allowing default type definitions. The default type definitions and choice of normalisation functions can be used to perform depth and stump analyses.

Chapter 6

Detection

Having developed the generic abstract semantics $F^{\flat}_{\mathcal{P},\mathcal{G}}$ and $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ in Chapter 4 and specialised $F^{\flat}_{\mathcal{P},\mathcal{G}}$ and $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ for mode and type analyses in Chapter 5, we now apply the mode analysis and type analysis to bug detection. Since $lfp F^{\diamond}_{\mathcal{P},\mathcal{G}}$ is an approximation of $lfp F^{\flat}_{\mathcal{P},\mathcal{G}}$, $lfp F^{\flat}_{\mathcal{P},\mathcal{G}}$ is finer-grained than $lfp F^{\diamond}_{\mathcal{P},\mathcal{G}}$. We therefore use $F^{\flat}_{\mathcal{P},\mathcal{G}}$ to perform the mode and the type analyses for the purpose of bug detection.

A bug symptom is either operational or declarative (see figure 1.1 in p. 5). In respect of operational symptoms, the program $\mathcal{P}$ is first statically checked for the satisfaction of its associated assertions. This is accomplished by computing $lfp F^{\flat}_{\mathcal{P},\mathcal{G}}$ with $\mathcal{G}$ describing all the possible initial goals for $\mathcal{P}$ and comparing, for each program point associated with an assertion, the assertion with the abstract substitutions associated, by $lfp F^{\flat}_{\mathcal{P},\mathcal{G}}$, with the edges in $\mathcal{E}_{\mathcal{P},\mathcal{G}}$ that end at the program point. If $\mathcal{P}$ passes the static checking then $\mathcal{P}$ does not have inadmissibility symptoms. Otherwise, a finite set of goals is generated from $\mathcal{G}$ and $\mathcal{P}$ is tested on the set of goals. In respect of declarative symptoms, the success set of each procedure in $\mathcal{P}$ is first approximated mainly by computing $lfp F^{\flat}_{\mathcal{P},\mathcal{G}}$ with $\mathcal{G}$ describing all the possible initial goals for the procedure, then the actual abstract success set of the procedure is compared with the intended abstract success set for the procedure. If they are different, a finite set of goals is generated from their difference and $\mathcal{P}$ is tested on the set of goals. The detection of both operational and declarative symptoms has three stages: comparing actual program properties with intended program properties, generating a finite set of goals and testing $\mathcal{P}$ on the set of goals. In the prototype debugger illustrated by figure 1.2, the tasks at these three stages are accomplished respectively by the *abstract bug detector*, the *test data generator* and the *tester*, which collectively constitute the *bug detector* in the *prototype debugger*. The *prototype debugger* represents a program point in $\mathcal{N}_{\mathcal{P},\mathcal{G}}$ as a formula of the form $(proc\_name/proc\_arity, clause\_no), position$ where $proc\_name$ and $proc\_arity$ are the name and the arity of the procedure in which the program point lies, $clause\_no$ is the rank, in the procedure, of the clause to which the program point belongs, and $position$ is the rank of the program point in the body of the clause.

The detection of operational symptoms is presented in section 6.1. Section 6.2 is devoted to the detection of declarative symptoms. The generation of a finite set of goals is the subject of section 6.3 and the testing of normal programs is that of section 6.4. The final section 6.5 is a summary of the chapter.

6.1 Operational Symptoms

An assertion is a formula of the form $(\mathcal{K}, \mathcal{I})$ where $\mathcal{K}$ is the *kind* of the assertion such as "type" and "mode" and $\mathcal{I}$ is an abstract instantiation state of kind $\mathcal{K}$. A *goal specification* is a tuple comprising a goal and an assertion. It specifies a set of goals. The kind of the goal specification is the kind of the assertion in it. For instance, $<\leftarrow rev(Li, Lo), (type, \{Li/list(nat)\})>$ specifies the set of goals of the form $\leftarrow rev(Li, Lo)$ with Li being a list of natural numbers.

For a normal program $\mathcal{P}$, a goal specification $< G_0, (\mathcal{K}_0, \mathcal{I}_0) >$ and a set S of associations of assertions with program points in $\mathcal{P}$, the detection of operational symptoms is accomplished as follows. Let $\mathcal{V}_0$ be the set of variables in G_0 and $\mathcal{V}_0'$ be the domain of $\mathcal{I}_0$. $lfp F^{\flat}_{\mathcal{P},\mathcal{G}}$ is first computed where $\mathcal{G} = \{< G_0, \theta_0^{\flat} >\}$ with

$$\theta_0^{\flat} =< \mathcal{I}_0 \cup \{X/\check{\top} | X \in \mathcal{V}_0 \setminus \mathcal{V}_0'\}, (\mathcal{V}_0 \setminus \{Y|\mathbf{grd}(\mathcal{I}_0(Y))\})^2 \setminus \{Z|\mathbf{var}(\mathcal{I}_0(Z))\}^2, \{(X, X)|X \in \mathcal{V}_0\} >$$

where $\check{\top}$ is the supremum of $< \check{D}, \check{\sqsubseteq} >$. The set of substitutions satisfying $\theta_0^{\flat}$ is the same as the set of substitutions satisfying $\mathcal{I}_0$. Then, for each $((\mathcal{K}_0, \mathcal{I})@p) \in S$ and each $p \leftarrow\!\bullet q \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$, $[lfp F^{\flat}_{\mathcal{P},\mathcal{G}}]_{p \leftarrow\bullet q}\downarrow Inst$ is compared with $\mathcal{I}$ so as to check statically whether or not $(\mathcal{K}_0, \mathcal{I})$ is satisfied each time the control is transferred from program point q to program point p. $(\mathcal{K}_0, \mathcal{I})$ is satisfied if $([lfp F^{\flat}_{\mathcal{P},\mathcal{G}}]_{p \leftarrow\bullet q}\downarrow Inst)(X) \check{\sqsubseteq} \mathcal{I}(X)$ for each variable X in the domain of $\mathcal{I}$. The program $\mathcal{P}$ is tested on a finite set of goals generated from the goal specification $< G_0, (\mathcal{K}_0, \mathcal{I}_0) >$ should the static checking fail.

Example 6.1.1 Let the goal specification be $<\leftarrow rev(Li, Lo), (type, \{Li/list(nat)\})>$ with the type definitions in example 5.5.5 and $\mathcal{P}$ the program in example 1.0.3. The programmer's expectation that Xs be a list each time the execution of the program reaches v can be expressed as $(type, \{Xs/list(any)\})@v$.

$lfpF^{\flat}_{\mathcal{P},\mathcal{G}}$ is first computed where $\mathcal{G} = \{<\leftarrow rev(Li, Lo), \theta^{\flat}_0 >\}$ with $\theta^{\flat}_0 =< \{Li/list(nat), Lo/any\}, \{(Lo, Lo)\}, \{(Li, Li), (Lo, Lo)\} >$.

$[lfpF^{\flat}_{\mathcal{P},\mathcal{G}}]_{v\leftarrow\bullet u}\downarrow Inst = \{Xs/nat\}$ and $[lfpF^{\flat}_{\mathcal{P},\mathcal{G}}]_{v\leftarrow\bullet w}\downarrow Inst = \bot[\![\{Xs\}]\!]$ are then compared with $\{Xs/list(nat)\}$. Since $\bot[\![\{Xs\}]\!](Xs)\check{\sqsubseteq}^t list(any)$, the assertion $(type, \{Xs/list(any)\})$ associated with v will be satisfied each time control is transferred from w to v. Since $nat \,\check{\not\sqsubseteq}^t list(any)$, it is not certain that the assertion $(type, \{Xs/list(any)\})$ associated with v will be satisfied each time control is transferred from u to v. The following message is issued by the *bug detector*, with u and v being written as $(rev/2, 2), 2$ and $(ap/3, 1), 1$ respectively.

```
**** Warning ****
when control is transferred from (rev/2,2),2 to (ap/3,1),1
        (type,[Xs/list(any)])
may be invalidated because the derived invariant is
        (type,[Xs/nat])
```

$\mathcal{P}$ is then tested on a set of goals generated from $<\leftarrow rev(Li, Lo), (type, \{Li/list(nat)\}) >$ and $rev([s(0)], Lo)$ is found to be an inadmissibility symptom of $\mathcal{P}$ since the execution of $\mathcal{P}$ on $rev([s(0)], Lo)$ reaches the program point v with a current substitution violating $(type, \{Xs/list(any)\})$. The interactions between the debugger and the oracle are as follows.

```
Is rev([],[]) valid? y
This is a valid solution for rev([],X).
        X = []
Is there any other solution for rev([],X)? n
operational(type,rev([s(0)],_15219)) has been detected.
```

The meaning of $operational(type, rev([s(0)], \_15219))$ is that the execution of $\mathcal{P}$ on $rev([s(0)], \_15219)$ causes a violation of a type assertion. ∎

Example 6.1.2 Let $\mathcal{P}$ be the following buggy maze search program. In the correct version of the program appearing in [14] (pp. 112-145), the first clause for the procedure $d/2$ is $d(a, b)$ instead of $d(a, B)$.

```
go(X,X,T).                          d(a,B). % point u
go(X,Y,T) :-                        d(b,e).
        d(X,Z), % point v           d(b,c).
        \+ member(Z,T),             d(c,d).
        go(Z,Y,[Z|T]).              d(e,f).
go(X,Y,T) :-                        d(g,e).
        d(Z,X), % point w
        \+ member(Z,T),
        go(Z,Y,[Z|T]).
```

One safeness condition for the negation as failure rule requires a negative literal to be ground when it is selected [74]. For this program, it can be expressed by $(mode, \{Z/\{ⓖ\}, T/\{ⓖ\}\})@v$

and $(mode, \{Z/\{ⓖ\}, T/\{ⓖ\}\})@w$. Let $<\leftarrow go(a, X, [\,]), (mode, \{X/\{ⓕ\}\})>$ be the goal specification.

$lfpF^{\flat}_{\mathcal{P},\mathcal{G}}$ is first computed with $\mathcal{G} = \{<\leftarrow go(a, X, [\,]), <\{X/\{ⓕ\}\}, \emptyset, \{(X, X)\}>>\}$.

It is then found that $(mode, \{Z/\{ⓖ\}, T/\{ⓖ\}\})$ is not guaranteed to be satisfied when control is transferred from u to either v or w since

$$[lfpF^{\flat}_{\mathcal{P},\mathcal{G}}]_{v\leftarrow\bullet u}\downarrow Inst = \{X/\{ⓖ\}, Y/\{ⓕ\}, Z/\{ⓕ, ⓖ\}, T/\{ⓖ, ⓞ\}\}$$

and

$$[lfpF^{\flat}_{\mathcal{P},\mathcal{G}}]_{w\leftarrow\bullet u}\downarrow Inst = \{X/\{ⓕ, ⓖ\}, Y/\{ⓕ\}, Z/\{ⓖ\}, T/\{ⓖ, ⓞ\}\}$$

So, the following messages are issued by the *bug detector*, with the program points u, v, w being written as $(d/2, 1), 1, (go/3, 2), 2$ and $(go/3, 3), 2$ respectively.

```
**** Warning ****
when control is transferred from (d/2,1),1 to (go/3,2),2
        (mode,[Z/[g],T/[g]])
may be invalidated because the derived invariant is
        (mode,[Z/[f,g],T/[g,o]])

**** Warning ****
when control is transferred from (d/2,1),1 to (go/3,3),2
        (mode,[Z/[g],T/[g]])
may be invalidated because the derived invariant is
        (mode,[Z/[g],T/[g,o]])
```

$\mathcal{P}$ is then tested on a set of goals generated from $<\leftarrow go(a, X, [\,]), (mode, \{X/\{ⓕ\}\})>$. $go(a, X, [\,])$ is found to be an inadmissibility symptom of $\mathcal{P}$ since the execution of $\mathcal{P}$ on $go(a, X, [\,])$ reaches v with a current substitution violating $(mode, \{Z/\{ⓖ\}, T/\{ⓖ\}\})$ which is reported as follows.

```
operational(mode,go(a,_15219,[])) has been detected.
```

$operational(mode, go(a, \_15219, [\,])$ means that the execution of $\mathcal{P}$ on $go(a, \_15219, [\,])$ violates a mode assertion. ∎

6.2 Declarative Symptoms

An abstract atom is of the form $r(\check{d}_1, \cdots, \check{d}_n)$ where r/n is a procedure and $\check{d}_1, \cdots, \check{d}_n \in \check{D}$ are descriptions. The concretisation function $\check{\gamma}$ mapping a description into a set of terms is extended to map an abstract atom into a set of atoms as follows.

$$\check{\gamma}(r(\check{d}_1, \cdots, \check{d}_n)) \stackrel{def}{=} \{r(t_1, \cdots, t_n) \mid 1 \leq i \leq n \wedge t_i \in \check{\gamma}(\check{d}_i)\}$$

The order on descriptions can be similarly extended to order abstract atoms.

$$r(\check{d}_1, \cdots, \check{d}_n) \check{\sqsubseteq} r(\check{d'}_1, \cdots, \check{d'}_n) \stackrel{def}{=} \forall 1 \leq i \leq n.(\check{d} \check{\sqsubseteq} \check{d'})$$

For a normal program $\mathcal{P}$ and a kind $\mathcal{K}$ of program property such as "type" or "mode", the detection of declarative symptoms is done procedure by procedure. Let r/n be a procedure defined in $\mathcal{P}$. The detection of declarative symptoms for r/n involves three stages. In the first stage, the actual abstract success set $\mathcal{AAS}_{r/n,\mathcal{K}}$ of r/n is obtained. This is done by computing $lfpF^{\flat}_{\mathcal{P},\mathcal{G}}$ with $\mathcal{G} = \{<\leftarrow r(X_1, \cdots, X_n), \top[\![\{X_1, \cdots, X_n\}]\!] >\}$, collecting an abstract atom $r(\check{d}_1, \cdots, \check{d}_n)$ with $d_i = ([lfpF^{\flat}_{\mathcal{P},\mathcal{G}}]_{p\leftarrow\bullet q} \downarrow Inst)(X_i)$ for each edge $p\leftarrow\bullet q \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$ with p being the exit point of $\leftarrow r(X_1, \cdots, X_n)$, and discarding those collected abstract atoms that are "less than" some other collected abstract atoms with respect to $\check{\sqsubseteq}$. In the second stage, $\mathcal{AAS}_{r/n,\mathcal{K}}$ is compared with the intended abstract success set $\mathcal{IAS}_{r/n,\mathcal{K}}$ for r/n, this being provided by the oracle. Let

$$\mathcal{WR}_{r/n,\mathcal{K}} \stackrel{def}{=} \{A \in \mathcal{AAS}_{r/n,\mathcal{K}} \mid \neg\exists B \in \mathcal{IAS}_{r/n,\mathcal{K}}.(A \check{\sqsubseteq} B)\}$$

$$\mathcal{MI}_{r/n,\mathcal{K}} \stackrel{def}{=} \{B \in \mathcal{IAS}_{r/n,\mathcal{K}} \mid \neg\exists A \in \mathcal{AAS}_{r/n,\mathcal{K}}.(B \check{\sqsubseteq} A)\}$$

In the third stage, $\mathcal{P}$ is tested on a finite set of goals generated from each abstract atom in $\mathcal{WR}_{r/n,\mathcal{K}}$ so as to find an inconsistency symptom, if any, and on a finite set of goals generated from each abstract atom in $\mathcal{MI}_{r/n,\mathcal{K}}$ so as to find an insufficiency symptom, if any.

Example 6.2.1 Consider the program in example 1.0.1 and the type definitions in example 5.5.5. For the procedure $ap/3$,

$$\mathcal{AAS}_{ap/3,type} = \{ap(list(empty), any, list(empty)), ap(list(any), list(empty), any)\}$$

The only intended abstract atom for $ap/3$ is $ap(list(any), list(any), list(any))$, that is,

$$\mathcal{IAS}_{ap/3,type} = \{ap(list(any), list(any), list(any))\}$$

Therefore,

$$\mathcal{WR}_{ap/3,type} = \{ap(list(empty), any, list(empty)), ap(list(any), list(empty), any)\}$$

$$\mathcal{MI}_{ap/3,type} = \{ap(list(any), list(any), list(any))\}$$

For the procedure $le/2$, $\mathcal{WR}_{le/2,type} = \emptyset$ and $\mathcal{MI}_{le/2,type} = \emptyset$.

For the procedure $gt/2$, $\mathcal{WR}_{gt/2,type} = \emptyset$ and $\mathcal{MI}_{gt/2,type} = \emptyset$.

For the procedure $pt/4$, $\mathcal{WR}_{pt/4,type} = \{pt(list(any), any, list(nat), list(any))\}$ and $\mathcal{MI}_{pt/4,type} = \emptyset$.

For the procedure $qs/2$, $\mathcal{WR}_{qs/2,type} = \{qs(list(any), any)\}$ and $\mathcal{MI}_{qs/2,type} = \emptyset$.

The following five symptoms are detected. The meaning of $declarative(wrong, A)$ is that A is an inconsistency symptom and that of $declarative(miss, A)$ is that A is an insufficiency symptom.

```
declarative(wrong,ap([_196043],[],[_196043|_197355]))
declarative(wrong,ap([],_197518,[]))
declarative(miss,ap([],[0],[0]))
declarative(wrong,part([_198517],_198604,[],[_198517]))
declarative(wrong,qs([_199165],[_199165|_199748]))
```

The following are interactions between the debugger and the oracle during testing.

```
Is ap([],[],[]) valid? y
Is ap([X],[],[X|Y]) valid? n
Is ap([],X,[]) valid? n
Is ap([],[],[X]) satisfiable? n
Is ap([],[],[X,Y]) satisfiable? n
Is ap([],[],[X,Y,Z]) satisfiable? n
Is ap([],[],[X,Y,Z,A1]) satisfiable? n
Is ap([],[X],[Y]) satisfiable? y
    X = ? 0.
    Y = ? 0.
Is part([],X,[],[]) valid? y
Is part([X],Y,[],[X]) valid? n
Is qs([],[]) valid? y
Is qs([X],[X|Y]) valid? n
```

∎

6.3 Test Data Generation

Test data generation is a process of selecting a set of input data from an input domain. The input domain is usually infinite. To make testing practical, the set of selected input data must be finite and of reasonable size. This section is devoted to generating a finite set of goals, called a test data set, from a goal specification or an abstract atom. Both cases reduce to generating a finite set of terms from a mode or type description. Let the goal specification be $< G_0, (\mathcal{K}_0, \mathcal{I}_0) >$. Let the domain of $\mathcal{I}_0$ be $\{X_1, \cdots, X_n\}$ and the finite set of terms generated from $\mathcal{I}_0(X_i)$ be $\mathcal{T}_i$. Then the set of goals generated from the goal specification is

$$\{G_0\theta_0 \mid dom(\theta_0) = \{X_1, \cdots, X_n\} \wedge \forall 1 \le i \le n.(X_i\theta_0 \in \mathcal{T}_i)\}$$

Let $r(d_1, \cdots, d_n)$ be an abstract atom and $\mathcal{T}_i$ be the finite set of terms generated from d_i for $1 \le i \le n$. Then the test data set generated from $r(d_1, \cdots, d_n)$ is

$$\{\leftarrow r(t_1, \cdots, t_n) \mid \forall 1 \le i \le n.(t_i \in \mathcal{T}_i)\}$$

A mode or a type description can be thought of as defining an *input domain* for a variable. The finite set of terms is generated by selecting terms in the input domain according to certain *testing hypotheses* [6, 84].

Regularity hypothesis Let us assume that it is possible, in some way, to associate a level of complexity with each member of some input domain. A regularity hypothesis states that the program behaves regularly with respect to this measure of the complexity. Specifically, it states that if the program works correctly for every input in the domain whose complexity is less than or equal to a constant k then it works correctly for any input in this domain. The complexity function which maps inputs in this domain into natural numbers and the constant k are parameters defined externally. Such an input domain is said to be *regular*.

Uniformity hypothesis If no complexity measure is suitable, it is necessary to partition the input domain into a collection of subdomains, called uniform subdomains, in such a way that the program under test behaves uniformly on these subdomains. For a program and a subdomain, if it is true that in working correctly for one input in the subdomain implies working correctly for any input in the subdomain, then the program is said to behave uniformly on the subdomain. Such a subdomain is said to be *uniform*.

Complexity function We make the following assumptions about the complexity function $|\cdot|$ in order to ensure the termination of test data generation.

- The complexity of any term in a uniform subdomain is 0. A term in that subdomain can be replaced by any other term in the same subdomain when testing takes place. Therefore, the terms in the subdomain are of the same complexity which we require to be 0.
- The complexity of any variable is 0.
- The complexity of a term in a regular subdomain is greater than the complexity of any of its subterms, should the term have any subterms. For otherwise, the complexity of the term might decrease or remain the same even if a subterm in a regular subdomain would be replaced by another subterm in the same regular subdomain with a greater complexity. This would make the test data set infinite. For instance, if $|\cdot|$ is defined as in the following then $TS_k \stackrel{def}{=} \{t \mid |t| \leq k\}$ is infinite for any k.

$$\begin{aligned} |[H|L]| &= -|H| + |L| \\ |[\,]| &= 0 \end{aligned}$$

Even if these assumptions are satisfied, TS_k can be infinite for two other reasons. Firstly, if a term t has a subterm t' which lies in an infinite uniform subdomain then an infinite number of terms of the same complexity can be produced from t by replacing t' with terms in the same uniform subdomain. Secondly, if a term t contains a variable X then an infinite number of terms of the same complexity can be produced from t by replacing X with other variables. Therefore, a finite subset of TS_k must be generated instead. The algorithms we develop address the first problem by selecting just one term from a uniform subdomain. The second problem is solved by generating only those terms which do not have multiple occurrences of variables. The result is such that any term t in TS_k can be obtained from a generated term t' by replacing variables in t' with variables and subterms of t' in uniform subdomains with terms in the same uniform subdomains.

6.3.1 Generating a Finite Set of Terms from a Type Description

Algorithm 6.3.1 generates a finite set of terms from a mono-type. Both regularity and uniformity hypotheses are supported. Algorithm 6.3.1 uses the following data structures. One data structure TS is the set of generated terms. Another data structure PG is a stack of templates. A template is a triple $< ST, VT, UT >$ where ST is the skeleton of the template. VT is a mono-variable-typing whose domain is a subset of $vars(ST)$. For each $X/M' \in VT$, X is to be replaced by a term in $\check{\gamma}^t(M')$. UT is also a mono-variable-typing whose domain is a subset of $vars(ST)$. For each $X/M' \in UT$, X is to be replaced by a term randomly selected from $\check{\gamma}^t(M')$. The domains of VT and UT are disjoint.

Algorithm 6.3.1 Generating a finite set of terms from a mono-type.

- inputs: A mono-type M, a set Σ of function symbols each of which is typed by a type clause, a set $\mathcal{H}$ of uniformity hypotheses, a complexity function $|\cdot|$ and a natural number k. If $uniform(M') \in \mathcal{H}$ then M' is a uniform subdomain. Otherwise, M' is a regular subdomain.

- output: a set of terms $TS \subseteq \check{\gamma}^t(M)$ such that any term $t \in \check{\gamma}^t(M)$ with $|t| \leq k$ can be obtained from a term $t' \in TS$ by changing variables in t' and replacing subterms in uniform subdomains with terms in the same uniform subdomains.

- Auxiliary functions and procedures:

  - procedure $newvar(V_1, \cdots, V_n)$ sets $V_1, \cdots, V_n$ to newly created variables;

  - function $replace(T, V, T')$ returns the result obtained by replacing each occurrence of V in T with T';

  - procedure $push(Stack, Element)$ pushes the element $Element$ onto the stack $Stack$;

- procedure $pop(Stack, Element)$ sets $Element$ to the top element of the stack $Stack$ and pops the top element off the stack $Stack$;
- function $rand\_sel(M', \Sigma)$ reterms a term that is randomly selected from $\check{\gamma}^t(M')$ with $\check{\gamma}^t$ being determined by the type clauses for function symbols in Σ. $rand\_sel(M', \Sigma)$ first enumerates a fixed number, say N, of terms in $\check{\gamma}^t(M')$, then generates a random number i satisfying $0 \leq i \leq N$, and finally returns the i^{th} enumerated term.

- Algorithm:

```
1   TS := ∅; PG := null; newvar(Y); push(PG, < Y, {Y/M}, ∅ >);
2   while PG ≠ null do
3       pop(PG, < ST, VT, UT >);
4       if    ∃X/M' ∈ VT
5       then if    uniform(M') ∈ 𝓗
6               then push(PG, < ST, VT \ {X/M'}, UT ∪ {X/M'} >)
7               else  if    M' = c(M'_1, ···, M'_m) ≠ any
8                       then forall f(X_1, ···, X_n) : c(β_1, ···, β_m) ← X_1 : T_1, ···, X_n : T_n
9                               do  newvars(X'_1, ···, X'_n);
10                                  ST' := replace(ST, X, f(X'_1, ···, X'_n));
11                                  κ := {β_j/M'_j | 1 ≤ j ≤ m};
12                                  VT' := VT \ {X/M'} ∪ {X'_i/T_i κ | 1 ≤ i ≤ n}
13                                  if |ST'| ≤ k then push(PG, < ST', VT', UT >) fi;
14                              od;
15                      else  push(PG, < ST, VT \ {X/M'}, UT >);
16                              forall f(X_1, ···, X_n) : c(β_1, ···, β_m) ← X_1 : T_1, ···, X_n : T_n
17                              do  newvars(X'_1, ···, X'_n);
18                                  ST' := replace(ST, X, f(X'_1, ···, X'_n));
19                                  VT' := VT \ {X/M'} ∪ {X'_i/any | 1 ≤ i ≤ n}
20                                  if |ST'| ≤ k then push(PG, < ST', VT', UT >) fi;
21                              od;
22                      fi;
23              fi;
24      else   forall X/M' ∈ UT do ST := replace(ST, X, rand_sel(M', Σ)) od;
25              TS := TS ∪ {ST};
26      fi;
27  od;
28  return TS;
```

The following is an invariant of the **while-do** loop statement (lines 2-27) in algorithm 6.3.1.

$$\left\{\begin{array}{l}\text{(i) Each term } t \in ST \text{ satisfies } |t| \leq k\text{, and (ii) for each } < ST, VT, UT > \\ \text{in } PG\text{, } |ST| \leq k\text{, and replacing } X \text{ in } ST \text{ with a term in } \check{\gamma}^t(M') \text{ for each} \\ X/M' \in VT \cup UT \text{ will result in a term in } \check{\gamma}^t(M).\end{array}\right\} \quad \ldots. \ (*)$$

$(*)$ holds when the loop is first entered since $TS = \emptyset$, $|Y| = 0$, and replacing Y with a term in $\check{\gamma}^t(M)$ will result in a term in $\check{\gamma}^t(M)$. $(*)$ is preserved by the body (lines 3-26) of the loop statement which pops a template $< ST, VT, UT >$ off PG (line 3) and pushes onto PG a (possibly zero) number of new templates obtained from $< ST, VT, UT >$. If $VT = \emptyset$ then each occurrence of X in ST is replaced by a term randomly selected from $\check{\gamma}^t(M')$ for each $X/M' \in UT$ (line 24), and the result is added into TS (line 25). $(*)$ is preserved since the line 24 does not change the complexity of ST and no new template is pushed onto PG. Otherwise, a formula X/M' is selected from VT (line 4) and a number of new templates obtained from $< ST, VT, UT >$ is pushed onto PG (lines 5-23) according to three different cases. In the case that M' is a uniform input subdomain, a single template $< ST, VT \setminus \{X/M'\}, UT \cup \{X/M'\} >$ is pushed onto PG (line 6). $(*)$ is preserved since TS, ST and $VT \cup UT$ are unchanged. In the case that $M' = c(M'_1, \cdots, M'_m) \neq any$, a new template $< ST', VT', UT >$ is obtained from $< ST, VT, UT >$ for each $f/n \in \Sigma$ such that $f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$ where ST' is obtained from ST by replacing X in ST with $f(X'_1, \cdots, X'_n)$ and $VT' = VT \setminus \{X/M'\} \cup \{X'_\imath/T_\imath\kappa \mid 1 \leq \imath \leq n\}$ with $\kappa = \{\beta_\jmath/M'_\jmath \mid 1 \leq \jmath \leq m\}$. $< ST', VT', UT >$ is pushed onto PG iff $|ST'| \leq k$ (line 13). $(*)$ is preserved since TS does not change, $f(t'_1, \cdots, t'_n) \in \check{\gamma}^t(M')$ if $(t'_i \in \check{\gamma}^t(T_i\kappa))$ for all $1 \leq i \leq n$ and $< ST', VT', UT >$ is pushed onto PG only if $|ST'| \leq k$. In the case that $M' = any$, a new template $< ST, VT \setminus \{X/M'\}, UT >$ is pushed onto PG (line 15). In addition, a new template $< ST', VT', UT >$ is obtained from $< ST, VT, UT >$ for each $f/n \in \Sigma$ where ST' is obtained from ST by replacing X in ST with $f(X'_1, \cdots, X'_n)$ and $VT' = VT \setminus \{X/M'\} \cup \{X'_\imath/any \mid 1 \leq \imath \leq n\}$. $< ST', VT', UT >$ is pushed onto PG iff $|ST'| \leq k$ (line 20). $(*)$ is preserved since TS does not change, $f(t'_1, \cdots, t'_n) \in \check{\gamma}^t(any)$ if $t'_i \in \check{\gamma}^t(any)$ and $< ST', VT', UT >$ is pushed onto PG only if $|ST'| \leq k$.

Example 6.3.2 Consider the type definitions in example 5.5.5. Let $|L|$ be the length of L for any list L and $k = 10$. The following is the set of terms generated for the mono-type $list(nat)$ with $\mathcal{H} = \{uniform(nat)\}$. The terms of the type nat are randomly selected from the first ten terms in $\check{\gamma}^t(nat)$. $s^n(0)$ is defined by $s^0(0) \stackrel{def}{=} 0$ and $s^n(0) \stackrel{def}{=} s(s^{n-1}(0))$ for $n > 0$.

$[\,]$

$[0]$

$[s^7(0), s^4(0)]$

$[s^3(0), s^5(0), s^9(0)]$

$[s^4(0), s^6(0), s^9(0), s^5(0)]$

$[s(0), s^6(0), s^2(0), s(0), s^5(0)]$

$[s^5(0), 0, s^4(0), s^4(0), s^2(0), s^5(0)]$

$[s^3(0), s^9(0), s^5(0), 0, s^3(0), s^4(0), s^4(0)]$

$[s^2(0), s^3(0), s^8(0), s^8(0), s^8(0), s^8(0), 0, s^2(0), s(0)]$

$[s^3(0), s^6(0), s^5(0), s^9(0), 0, 0, 0, 0, s^8(0)]$

$[0, s^7(0), s^4(0), s^7(0), s^2(0), s^(0), s^7(0), s^8(0), s^8(0), s^6(0)]$

$\blacksquare$

6.3.2 Generating a Finite Set of Terms from a Mode Description

Algorithm 6.3.3 generates a finite set of terms from a mode description, this being a set of modes. Only regularity hypotheses are supported since it is unlikely that a program should behave uniformly on all terms having the same mode description. The data structures in algorithm 6.3.3 are similar to those in algorithm 6.3.1. One data structure is the set TS of generated terms. Another data structure PG is a stack of templates. A template is a tuple $< ST, VS >$ where ST is the skeleton of the template. VS is a set of variables in ST. For a given description MD, algorithm 6.3.3 simply enumerates a number of terms t satisfying $|t| \leq k$ and returns those enumerated terms whose modes are in MD.

Algorithm 6.3.3 Generating a finite set of terms from a mode description.

- inputs: a mode description MD, a set Σ of function symbols, a complexity function $|\cdot|$ and a natural number k.
- output: a set of terms $TS \subseteq \{t \mid |t| \leq k \wedge mo(t) \in MD\}$ such that any term t with $|t| \leq k \wedge mo(t) \in MD$ can be obtained from a term $t' \in TS$ by changing variables in t'.
- Auxiliary function $replace(\cdot,\cdot,\cdot)$ and procedures $newvar(\cdots)$, $push(\cdot,\cdot)$ and $pop(\cdot,\cdot)$ have the same specifications as in algorithm 6.3.1.

- Algorithm:

$TS := \emptyset$; $PG := null$; $newvar(Y)$; $push(PG, < Y, \{Y\} >)$;
while $PG \neq null$ **do**
    $pop(PG, < ST, VS >)$;
    if $\exists X \in VS$
    then $push(PG, < ST, VS \setminus \{X\} >)$;
        forall $f/n \in \Sigma$
        do $newvars(X'_1, \cdots, X'_n)$;
            $ST' := replace(ST, X, f(X'_1, \cdots, X'_n))$;
            $VS' := VS \setminus \{X\} \cup \{X'_\imath \mid 1 \leq \imath \leq n\}$
            if $|ST'| \leq k$ **then** $push(PG, < ST', VS' >)$ **fi**;
        od;
    else **if** $mo(ST) \in MD$ **then** $TS := TS \cup \{ST\}$ **fi**;
    fi;
od;
return TS;

Example 6.3.4 Let $\Sigma = \{0/0, s/1, [\,]/0, ./2\}$, $|t|$ be the number of the nodes in the tree representing t for any term t and $k = 3$. The set of terms generated from the mode description $\{ⓖ\}$ is $\{0, s(0), s(s(0)), s(s([\,])), s([\,]), [\,], [0|0], [0], [[\,]|0], [[\,]]\}$. ∎

6.4 Testing

Let $\mathcal{P}$ be a normal program, S be a set of associations of assertions with program points in $\mathcal{P}$ and A an atom. The testing of $\mathcal{P}$ on $\leftarrow A$ is done in three steps.

In the first step, $\mathcal{P}$ and S are combined to form a new program $\mathcal{P}^*$. Assume that $assertion/2$ is not a predicate in $\mathcal{P}$. For each $((\mathcal{K}, \mathcal{I})@p) \in S$, a new literal $assertion(\mathcal{K}, \mathcal{I})$ is inserted into $\mathcal{P}$ at the program point p. Each literal previously at or to the right of the program point p is pushed one place rightwards. This combination of $\mathcal{P}$ and S needs to be done once when $\mathcal{P}$ is tested on a set of goals.

In the second step, the detection of operational symptoms is carried out by the following meta-interpreter ($exec/3$) which simulates SLDNF, checks for satisfaction of assertions and detects suspected non-termination symptoms and violations of the weaker safeness condition for the negation as failure rule.

```
exec(A,0,err(overflow)).
exec(true,s(N),true).
exec(\+A,s(N),R)          :-
    copy(A,B),
    (
        exec(B,N,Ra)
    ->
        (
            Ra = true
        ->
            (
                \+ variant(A,B)
            ->
                R = err(unsafe)
            )
        ;
            R = Ra
        )
    ;
        R=true
    ).
exec(assertion(K,A),s(N),R) :-
    (
        verify(K,A)
    ->
        R = true
    ;
        R = err(K)
    ).
exec((A,B),s(N),R)          :-
    exec(A,s(N),Ra),
    (
        Ra = true
    ->
        exec(B,s(N),R)
    ;
        R = Ra
    ).
exec(A,s(N),R)              :-
    clause(A,B),
    exec(A,N,R).
```

The program $\mathcal{P}^*$ under test is an implicit input to *exec*/3. The first argument to *exec*/3 is the test data on which $\mathcal{P}^*$ is to be tested. The second argument to *exec*/3 is a limit on the number of nested procedure calls. The third argument, which is output, indicates if an operational symptom has been detected and records the kind of the operational symptom. If $\mathcal{P}^*$ succeeds on A within the limit N then $exec(A, N, R)$ succeeds with R being instantiated to the constant true. $exec(A, N, R)$ also succeeds if an operational symptom is detected during the execution of $\mathcal{P}^*$ on A. R is instantiated to a term of the form $err(K)$ where K is the kind of the operational symptom.

The first clause detects suspected non-termination symptoms when the limit is exceeded. The second clause states that $\mathcal{P}^*$ succeeds on true. The third clause copes with negative literals. Let $\neg A$ be the literal to be solved. A copy B of A is made by *copy*/2 in order to avoid instantiating variables in A. Any operational symptom detected while $\mathcal{P}^*$ is executed on B is carried through. If $\mathcal{P}^*$ finitely fails on B then $\mathcal{P}^*$ succeeds on $\neg A$. Two cases arise when $\mathcal{P}^*$ succeeds on B. If A is a variant of B then $\mathcal{P}^*$ fails on $\neg A$. Otherwise, a violation of the weaker safeness condition for the negation as failure rule has been detected. The specification for *variant*/2 is that $variant(A, B)$ succeeds iff A is a variant of B. The fourth clause deals with assertions and detects inadmissibility symptoms. The specification for *verify*/2 is that $verify(\mathcal{K}, \mathcal{I}\sigma)$ does not instantiate any variable and succeeds iff $\mathcal{I}$ is satisfied by σ. The fifth clause deals with conjunctions of literals and the sixth clause simulates a derivation step.

By lemma 5.5.22, a type assertion $(type, \mathcal{I})$ is satisfied by a substitution σ iff $\check{\alpha}^t(X\sigma) \check{\sqsubseteq}^t \mathcal{I}(X)$

for each variable X in the domain of $\mathcal{I}$. Therefore, $verify(type, \{t_1/M_1, \cdots, t_k/M_k\})$ succeeds iff $\forall 1 \leq j \leq k.(\check{\alpha}^t(t_j) \check{\sqsubseteq}^t M_j)$. A mode assertion $(mode, \mathcal{I})$ is satisfied by a substitution σ iff $mo(X\sigma) \in \mathcal{I}(X)$ for each variable X in the domain of $\mathcal{I}$. So, $verify(mode, \{t_1/M_1, \cdots, t_k/M_k\})$ succeeds iff $\forall 1 \leq j \leq k.(mo(t_j) \in M_j)$.

The operational symptom detected by $exec/3$ is returned by the *tester*. If $exec/3$ does not detect any operational symptom then the result of the execution is compared with the intended interpretation $\mathcal{M}$ in the third step so as to detect a declarative symptom.

6.5 Summary

This chapter has presented our approach to the detection of operational and declarative symptoms. The combination of static checking and program testing is used. In respect of operational symptoms, the program with its goal specification is statically checked for possible violations of its associated assertions. If the program with its goal specification passes the static checking then the execution of the program on any goal specified by the goal specification will not cause any violation of its associated assertions. Otherwise, the program is tested on a test data set, generated from the goal specification, in order to detect operational symptoms. In respect of declarative symptoms, the actual abstract success set of each procedure in the program is compared with its intended abstract success set and the program is tested on a test data set, generated from each abstract atom in the difference between the actual abstract success set and the intended abstract success set, in order to detect declarative symptoms.

Chapter 7

Diagnosis

A bug symptom alone does not provide the programmer with much information, be it operational or declarative. Before making any correction to the program, he needs to diagnose the program in order to find the bug in the program which has caused the bug symptom. Any software supporting bug diagnosis is called a diagnoser. In this chapter, we present a declarative diagnoser for normal programs. It comprises an inconsistency diagnosis procedure, an insufficiency diagnosis procedure, and a diagnosis procedure for operational symptoms.

The inconsistency diagnosis procedure and the insufficiency diagnosis procedure that we have developed have a smaller search space than those reported in the literature. Whilst a program $\mathcal{P}$ is being debugged, the oracle incrementally provides the debugging system with the intended interpretation $\mathcal{M}$ for $\mathcal{P}$. Therefore, at some stage of debugging, the debugging system has already acquired part $\mathcal{M}'$ of $\mathcal{M}$. The inconsistency diagnosis procedure and the insufficiency diagnosis procedure use $\mathcal{M}'$ to reduce the size of the search space.

Given an inconsistency symptom A_w of $\mathcal{P}$ w.r.t. $\mathcal{M}$, the inconsistency diagnosis procedure first constructs a tree called an $\mathcal{M}$-congruent partial proof tree ($\mathcal{M}$-*cpp*) for $\mathcal{P}$ and A_w and then diagnoses $\mathcal{P}$ by searching this tree. An $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w is similar to a proof tree for $\mathcal{P}$ and A_w [29] except that a leaf of an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w is an atom that is valid in $\mathcal{M}$ while a leaf of a proof tree for $\mathcal{P}$ and A_w is the atom true. Because $\mathcal{M}'$ is a part of $\mathcal{M}$, an $\mathcal{M}'$-*cpp* for $\mathcal{P}$ and A_w is also an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w. Therefore, an $\mathcal{M}'$-*cpp* for $\mathcal{P}$ and A_w can be used where an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w is required. Whilst an $\mathcal{M}'$-*cpp* for $\mathcal{P}$ and A_w is being constructed it is not necessary to query the oracle because $\mathcal{M}'$ has already been made known to the debugging system. An $\mathcal{M}'$-*cpp* for $\mathcal{P}$ and A_w is smaller than or equal in size to a proof tree for $\mathcal{P}$ and A_w because an $\mathcal{M}'$-*cpp* for $\mathcal{P}$ and A_w may contain only one node for an atom that is valid in

$\mathcal{M}'$ while a proof tree for $\mathcal{P}$ and A_w has a subtree for the same atom. When $\mathcal{M}'$ is empty, an $\mathcal{M}'$-*cpp* for $\mathcal{P}$ and A_w is a proof tree for $\mathcal{P}$ and A_w. As $\mathcal{M}'$ increases during debugging, the inconsistency diagnosis procedure is able to construct a smaller and smaller $\mathcal{M}'$-*cpp* for $\mathcal{P}$ and A_w.

Given an insufficiency symptom A_m of $\mathcal{P}$ w.r.t. $\mathcal{M}$, the insufficiency diagnosis procedure first constructs a tree called an $\mathcal{M}$-complete partial SLDNF-tree ($\mathcal{M}$-*cps*) for $\mathcal{P} \cup \{\leftarrow A_m\}$ and then diagnoses by searching this tree. An $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ is similar to an SLDNF-tree for $\mathcal{P} \cup \{\leftarrow A_m\}$ [74] except that a node $\leftarrow W$ in an SLDNF-tree for $\mathcal{P} \cup \{\leftarrow A_m\}$ has a child node for each goal $\leftarrow W'$ that is derived from $\leftarrow W$ and $\mathcal{P}$, while a node $\leftarrow W$ in an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ only needs to have a child node for a goal $\leftarrow W'$ that is derived from $\leftarrow W$ and $\mathcal{P}$ such that W' is satisfiable in $\mathcal{M}$. In other words, if W' is unsatisfiable in $\mathcal{M}$ then node $\leftarrow W'$ and the subtree rooted at $\leftarrow W'$ can be removed from an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$. Because $\mathcal{M}'$ is a part of $\mathcal{M}$, an $\mathcal{M}'$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ is also an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$. Hence, an $\mathcal{M}'$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ can be used where an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ is needed. The construction of an $\mathcal{M}'$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ does not need to query the oracle because $\mathcal{M}'$ has already been made known to the debugging system. An $\mathcal{M}'$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ is smaller in size than an SLDNF-tree for $\mathcal{P} \cup \{\leftarrow A_m\}$ because a node $\leftarrow W$ in an SLDNF-tree for $\mathcal{P} \cup \{\leftarrow A_m\}$ usually has more child nodes than in an $\mathcal{M}'$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$. When $\mathcal{M}'$ is empty, an $\mathcal{M}'$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ is an SLDNF-tree for $\mathcal{P} \cup \{\leftarrow A_m\}$. As $\mathcal{M}'$ increases during debugging, the insufficiency diagnosis procedure is able to construct a smaller and smaller $\mathcal{M}'$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$.

Given an operational symptom A_o of $\mathcal{P}$, the diagnosis procedure that we have developed for operational symptoms first constructs a tree, called the active execution tree for $\mathcal{P}$ and A_o that models the execution of $\mathcal{P}$ on A_o up to the detection of an unexpected behaviour of the execution and then diagnoses by searching the tree.

Our inconsistency diagnosis procedure and insufficiency diagnosis procedure are generic in that different tree search strategies can be used with the them and the user can specialise them by specifying the search strategy to be used.

Section 7.1 formally introduces the concepts of $\mathcal{M}$-*cpp* and $\mathcal{M}$-*cps* and establishes that they are sufficient for the purpose of diagnosis, that is, an inconsistency symptom A_w of $\mathcal{P}$ w.r.t. $\mathcal{M}$ can be diagnosed by searching an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w, and an insufficiency symptom A_m of $\mathcal{P}$ w.r.t. $\mathcal{M}$ can be diagnosed by searching an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$. The notion of the active execution tree for $\mathcal{P}$ and an operational symptom A_o of $\mathcal{P}$ is also introduced. Section 7.2 presents the diagnoser and in section 7.3 we prove its soundness and completeness.

In section 7.4, we show the generality of our inconsistency diagnosis procedure and insufficiency diagnosis procedure and compares them with those in the literature with respect to the size of the search space. Section 7.5 concludes the chapter.

7.1 Search Space

This section formally introduces the concepts of $\mathcal{M}$-*cpp* and $\mathcal{M}$-*cps* and shows that an inconsistency symptom A_w of $\mathcal{P}$ w.r.t. $\mathcal{M}$ can be diagnosed by searching an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w and an insufficiency symptom A_m of $\mathcal{P}$ w.r.t. $\mathcal{M}$ can be diagnosed by searching an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$. We also introduce the notion of the active execution tree for $\mathcal{P}$ and an operational symptom A_o of $\mathcal{P}$ and prove that an inadmissibility symptom A_o of $\mathcal{P}$ can be diagnosed by searching the active execution tree for $\mathcal{P}$ and A_o.

Let T be a tree, r the root of T, and v a node of T. v is a branch node if v is neither the root of T nor a leaf of T. The height of T, written as $h(T)$, is the length of the longest path of T. $n(T)$ denotes the number of nodes of T. The level of v in T, denoted by $l(v, T)$, is the length of the path from r to v. T_v denotes the sub-tree of T that is rooted at v. Notice that $T = T_r$.

Definition 7.1.1 An ordered tree T is a literal-labelled tree if each node of T is a literal. Let L be a node of T and $L_1, L_2, \cdots, L_k$ the children of L in that order. We say that $L \leftarrow L_1, L_2, \cdots, L_k$ is the root implication of T_L and write it as $RI(T, L)$. ∎

Definition 7.1.2 Let $\mathcal{P}$ be a normal program, $\mathcal{M}$ an interpretation and A an atom.

(1) A partial proof tree T for $\mathcal{P}$ and A is a literal-labelled tree satisfying the following two conditions. (i) The root of T is A. (ii) For each non-leaf node L' of T, either $RI(T, L')$ is a clause instance of $\mathcal{P}$ or $RI(T, L') = (\neg A' \leftarrow \mathsf{true})$ where A' is an atom on which $\mathcal{P}$ finitely fails.

(2) T is a proof tree for $\mathcal{P}$ and A if T is a partial proof tree for $\mathcal{P}$ and A, and every leaf node of T is true.

(3) A partial proof tree T for $\mathcal{P}$ and A is an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A if every leaf node of T is a literal that is valid in $\mathcal{M}$. ∎

A proof tree T for $\mathcal{P}$ and A is an $\mathcal{M}$-*cpp* T for $\mathcal{P}$ and A because true is valid in $\mathcal{M}$. This definition of a proof tree is similar to that of [29]. The leaves of a proof tree defined in [29] are instances of unit clauses of $\mathcal{P}$ while they are **true** according to the above definition that treats a unit clause as having body **true**.

Example 7.1.3 Let $\mathcal{P}$ be the following buggy quicksort program. $\mathcal{P}$ has a bug that is indicated by a comment.

```
qs([X|L],L0):-
  pt(L,X,L1,L2),
  qs(L1,L3),
  qs(L2,L4),
  ap([X|L3],L4,L0).
  %ap(L3,[X|L4],L0)
qs([],[]).

pt([X|L],Y,L1,[X|L2]) :-
  Y=<X,
  pt(L,Y,L1,L2).
pt([X|L],Y,[X|L1],L2) :-
  Y>X,
  pt(L,Y,L1,L2).
pt([],X,[],[]).

ap([X|L1],L2,[X|L3]) :-
  ap(L1,L2,L3).
ap([],L,L).
```

Let the intended interpretation $\mathcal{M}$ be as usual. $qs([2,3,1],[2,1,3])$ is an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$. Suppose that the acquired part $\mathcal{M}'$ of $\mathcal{M}$ consists of the knowledge of built-in predicates. Then the following is an $\mathcal{M}'$-*cpp* for $\mathcal{P}$ and $qs([2,3,1],[2,1,3])$ and hence an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and $qs([2,3,1],[2,1,3])$.

```
qs([2,3,1],[2,1,3])------------------|                          (1)
|--pt([3,1],2,[1],[3])   (2)         |--qs([3],[3])             (18)
|  |--2=<3               (3)         |  |--pt([],3,[],[])       (19)
|  |--pt([1],2,[1],[])   (4)         |  |  |--true              (20)
|     |--2>1             (5)         |  |--qs([],[])            (21)
|     |--pt([],2,[],[])  (6)         |  |  |--true              (22)
|        |--true         (7)         |  |--qs([],[])            (23)
|--qs([1],[1])           (8)         |  |  |--true              (24)
   |--pt([],1,[],[])     (9)         |  |--ap([3],[],[3])       (25)
   |  |--true            (10)        |     |--ap([],[],[])      (26)
   |--qs([],[])          (11)        |        |--true           (27)
   |  |--true            (12)        |--ap([2,1],[3],[2,1,3])   (28)
   |--qs([],[])          (13)           |--ap([1],[3],[1,3])    (29)
   |  |--true            (14)              |--ap([],[3],[3])    (30)
   |--ap([1],[],[1])     (15)                 |--true           (31)
      |--ap([],[],[])    (16)
         |--true         (17)
```

The above tree is smaller than a proof tree for $\mathcal{P}$ and $qs([2,3,1],[2,1,3])$ because in a proof tree for $\mathcal{P}$ and $qs([2,3,1],[2,1,3])$ each of node (3) (labelled with $2 =< 3$) and node (5) (labelled with $2 > 1$) has a child labelled with **true**. ∎

Example 7.1.4 Let $\mathcal{P}$ and $\mathcal{M}$ be the same as in example 7.1.3. Suppose that at some stage during debugging, the acquired part $\mathcal{M}'$ of $\mathcal{M}$ consists of the knowledge of built-in predicates and the knowledge that $qs([X],[X])$ is valid in $\mathcal{M}$ for all possible natural numbers X. Then removing nodes 9-17 and 19-27 from the tree in example 7.1.3 will result in an $\mathcal{M}'$-*cpp* for $\mathcal{P}$ and $qs([2,3,1],[2,1,3])$ and hence an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and $qs([2,3,1],[2,1,3])$. This tree and the tree in example 7.1.3 are both $\mathcal{M}$-*cpp* for $\mathcal{P}$ and $qs([2,3,1],[2,1,3])$. But the former is much smaller than the latter. This example shows that, given the same inconsistency symptom A_w

of $\mathcal{P}$ w.r.t. $\mathcal{M}$, our inconsistency diagnosis procedure is able to construct a smaller and smaller $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w as $\mathcal{M}'$ increases during debugging. ∎

We now establish the relation between an inconsistency symptom A_w of $\mathcal{P}$ w.r.t. $\mathcal{M}$ and an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w.

Lemma 7.1.5 Let $\mathcal{M}$ be an interpretation and T a literal-labelled tree. If the root of T is invalid in $\mathcal{M}$ and each leaf of T is valid in $\mathcal{M}$, then there is a node L' of T such that $RI(T, L')$ is invalid $\mathcal{M}$.

PROOF: See p. 179. ∎

Lemma 7.1.6 Let $\mathcal{P}$ be a normal program, $\mathcal{M}$ an interpretation, A_w an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$, and T an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w. If L' is a node of T such that $RI(T, L')$ is invalid in $\mathcal{M}$, then either $RI(T, L')$ is an inconsistent clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$, or $RI(T, L') = (\neg A_m \leftarrow \textsf{true})$ and A_m is an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

PROOF: The proof follows from definitions 2.3.2 and 7.1.2 ∎

The consequence of lemmas 7.1.5 and 7.1.6 is that an inconsistency symptom A_w of $\mathcal{P}$ w.r.t. $\mathcal{M}$ can be diagnosed by searching an $\mathcal{M}$-*cpp* T for $\mathcal{P}$ and A_w to find a node L' of T such that $C = RI(T, L')$ is invalid in $\mathcal{M}$. Either C is an inconsistent clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$, or $C = (\neg A_m \leftarrow \textsf{true})$ and A_m is an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

Definition 7.1.7 Let $\mathcal{P}$ be a normal program, R a computation rule, G and G' normal goals, and $\mathcal{M}$ an interpretation.

(1) We say that G' is derived from G and $\mathcal{P}$ via R and write $G \xrightarrow{\mathcal{P},R} G'$ if $G = \leftarrow L_1, \cdots, L_i, \cdots, L_m$, L_i is the selected literal of G by R, and either (i) L_i is positive, there is a clause $A \leftarrow W$ of $\mathcal{P}$ such that L_i and A unify with a *mgu* θ, and $G' = \leftarrow (L_1, \cdots, L_{i-1}, W, L_{i+1}, \cdots, L_m)\theta$, or (ii) L_i is negative with $L_i = \neg A_i$, $\mathcal{P}$ finitely fails on A_i, and $G' = \leftarrow (L_1, \cdots, L_{i-1}, L_{i+1}, \cdots, L_m)$.

(2) $\mathcal{C}(G, \mathcal{P}, R) \stackrel{def}{=} \{G' | G \xrightarrow{\mathcal{P},R} G'\}$ is the set of all the goals that are derived from G and $\mathcal{P}$ via R.

(3) $\mathcal{E}(G, \mathcal{P}, R, \mathcal{M}) \stackrel{def}{=} \{\leftarrow W' \mid (\leftarrow W' \in \mathcal{C}(G, \mathcal{P}, R))\ \mathit{and}\ W'\ \mathit{is\ satisfiable\ in}\ \mathcal{M}\}$ is the subset of $\mathcal{C}(G, \mathcal{P}, R)$ such that for each goal $G' = \leftarrow W'$ in $\mathcal{E}(G, \mathcal{P}, R, \mathcal{M})$, W' is satisfiable in $\mathcal{M}$. ∎

An ordered tree T is a goal-labelled tree if each node of T is a goal.

Definition 7.1.8 Let $\mathcal{P}$ be a normal program, R a computation rule, G a normal goal, T a goal-labelled tree such that the root of T is G, and $\mathcal{M}$ an interpretation.

(1) T is a partial SLDNF-tree for $\mathcal{P} \cup \{G\}$ via R if any two nodes G^1 and G^2 of T such that G^2 is a child of G^1 satisfy $G^2 \in \mathcal{C}(G^1, \mathcal{P}, R)$.

(2) T is an SLDNF-tree for $\mathcal{P} \cup \{G\}$ via R if T is a partial SLDNF-tree for $\mathcal{P} \cup \{G\}$ via R such that if G^1 is a node of T, then each G^2 in $\mathcal{C}(G^1, \mathcal{P}, R)$ is also a node of T and G^2 is a child of G^1.

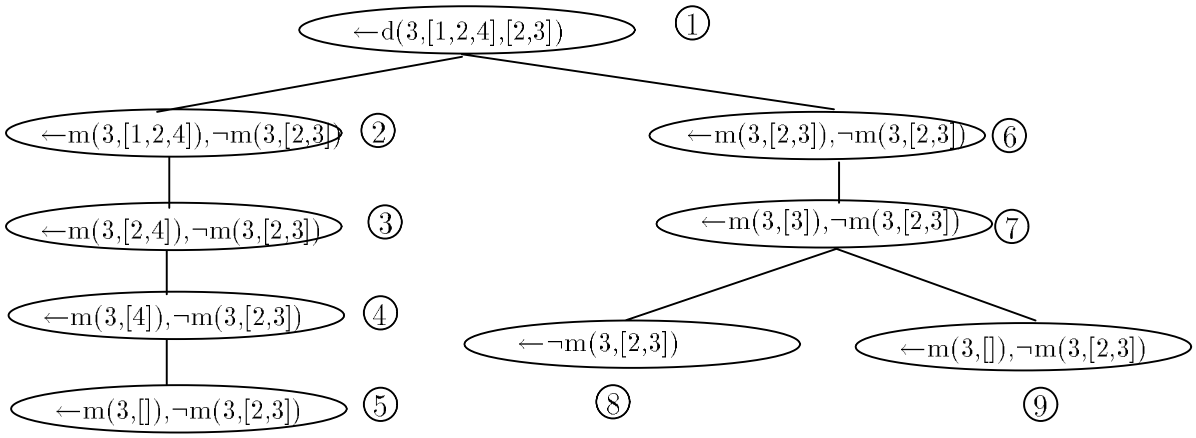
(3) T is an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{G\}$ via R if T is a partial SLDNF-tree for $\mathcal{P} \cup \{G\}$ via R such that if G^1 is a node of T, then each G^2 in $\mathcal{E}(G^1, \mathcal{P}, R, \mathcal{M})$ is also a node of T and G^2 is a child of G^1. ∎

The definition of an SLDNF-tree is equivalent to that given in the literature such as [74]. The notion of an $\mathcal{M}$-*cps* T for $\mathcal{P} \cup \{G\}$ via R captures the idea that if $\mathcal{P}$ is correct with respect to $\mathcal{M}$, then any successful derivation of $\mathcal{P} \cup \{G\}$ via R corresponds to a path from the root of T to a leaf of T which is $\Box$. It follows from definition 7.1.8 that an SLDNF-tree for $\mathcal{P} \cup \{G\}$ via R is an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{G\}$ via R for any $\mathcal{M}$. If A_m is an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$, then an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ via a fair computation rule R is a finite tree, and none of its leaves is $\Box$ because any derivation of $\mathcal{P} \cup \{\leftarrow A_m\}$ terminated with $\Box$ corresponds to a proof tree for $\mathcal{P}$ and $A_m\theta$ for some θ.

Example 7.1.9 Let $\mathcal{P}$ be the following buggy program. The intended interpretation for d(X,Ys,Zs) is that either X is in list Ys but not in list Zs or X is in list Zs but not in list Ys. The intended interpretation for m(X,L) is that X is in list L.

```
d(X,Ys,Zs):- m(X,Ys),\+ m(X,Zs).          m(X,[X|Xs]).
d(X,Ys,Zs):- m(X,Zs),\+ m(X,Zs).          m(X,[Y|Ys]):- m(X,Ys).
                     %\+ m(X,Ys)
```

$d(3,[1,2,4],[2,3])$ is an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$. Suppose that the acquired part $\mathcal{M}'$ of $\mathcal{M}$ is empty. Then the following is an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow d(3,[1,2,4],[2,3])\}$ via the left-to-right computation rule. $\leftarrow$ is written as ':-'. This tree is also an SLDNF-tree for $\mathcal{P} \cup \{\leftarrow d(3,[1,2,4],[2,3])\}$ via the left-to-right computation rule.



■

Example 7.1.10 Let $\mathcal{P}$ and $\mathcal{M}$ be the same as in example 7.1.9. $Q \wedge \neg Q$ is unsatisfiable in $\mathcal{M}'$ for any Q and any consistent $\mathcal{M}'$. So, removing nodes 6-9 from the tree in example 7.1.9 will result in an $\mathcal{M}'$-*cps* for $\mathcal{P} \cup \{\leftarrow d(3,[1,2,4],[2,3])\}$ via the left-to-right computation rule. This tree and the tree in example 7.1.9 are both $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow d(3,[1,2,4],[2,3])\}$ via the left-to-right computation rule. But the former is much smaller than the latter. This example shows that, given the same insufficiency symptom A_m of $\mathcal{P}$ w.r.t. $\mathcal{M}$, our insufficiency diagnosis procedure is able to construct a smaller and smaller $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ via a fixed computation rule as $\mathcal{M}'$ increases during debugging. ■

Definition 7.1.11 Let $\mathcal{M}$ be an interpretation, T a goal-labelled tree and $\leftarrow W$ a node of T. We say $\leftarrow W$ is a critical node of T w.r.t. $\mathcal{M}$ if W is satisfiable in $\mathcal{M}$ and for each child $\leftarrow W'$ of $\leftarrow W$, W' is unsatisfiable in $\mathcal{M}$. ■

We now establish the relation between an insufficiency symptom A_m of $\mathcal{P}$ w.r.t. $\mathcal{M}$ and an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ via a computation rule R.

Lemma 7.1.12 Let $\mathcal{P}$ be a normal program, G^1 a normal goal, R a computation rule, $\mathcal{M}$ an interpretation, T an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{G^1\}$ via R, $G = \leftarrow W = \leftarrow W_1, L, W_2 \neq \Box$ a critical node of T w.r.t. $\mathcal{M}$, and L the selected literal of G by R.

(1) If L is positive, then L is an incompletely covered atom of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

(2) If $L = \neg A_w$ is negative, then A_w is an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

PROOF: See p. 179. ■

Lemma 7.1.13 Let $\mathcal{M}$ be an interpretation, T a goal-labelled tree, the root of T be $\leftarrow W$. If W is satisfiable in $\mathcal{M}$, then there is a critical node of T w.r.t. $\mathcal{M}$.

PROOF: See p. 180. ∎

The consequence of lemmas 7.1.12 and 7.1.13 is that an insufficiency symptom A_m of $\mathcal{P}$ w.r.t. $\mathcal{M}$ can be diagnosed by searching an $\mathcal{M}$-*cps* T for $\mathcal{P} \cup \{\leftarrow A_m\}$ via a fair computation rule R to find a critical node G of T w.r.t. $\mathcal{M}$. $G \neq \Box$ since $\mathcal{P}$ finitely fails on A_m. Let L be the selected literal of G by R. By lemma 7.1.12, either L is an incompletely covered atom of $\mathcal{P}$ w.r.t. $\mathcal{M}$, or $L = \neg A_w$ and A_w is an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

Let $\mathcal{P}$ be a program and A_o be an operational symptom of $\mathcal{P}$. There is a program point p and a substitution σ such that an unexpected behaviour of the execution of $\mathcal{P}$ on A_o was detected when the execution reached p with current substitution σ. Let $B_{(i_0,k_0)} = A$ and $\sigma_0 = \epsilon$. There must be a clause C_{i_1} in $\mathcal{P}$ that has been used to solve $B_{(i_0,k_0)}\sigma_0$, a program point (i_1, k_1) in C_{i_1} and a substitution σ_1 such that $\mathcal{P}$ succeeds on $L_{(i_1,1)}\sigma_1, L_{(i_1,2)}\sigma_1, \cdots, L_{(i_1,k_1-1)}\sigma_1$ and either (a) $p = (i_1, k_1)$ and $\sigma = \sigma_1$ or (b) the unexpected behaviour of the execution of $\mathcal{P}$ on A_o was detected during the execution of $\mathcal{P}$ on $B_{(i_1,k_1)}\sigma_1$. In case (b), there must be a clause C_{i_2} of $\mathcal{P}$ that has been used to solve $B_{(i_1,k_1)}\sigma_1$, a program point (i_2, k_2) in C_{i_2} and a substitution σ_2 such that $\mathcal{P}$ succeeds on $L_{(i_2,1)}\sigma_2, L_{(i_2,2)}\sigma_2, \cdots, L_{(i_2,k_2-1)}\sigma_2$ and either (b.1) $p = (i_2, k_2)$ and $\sigma = \sigma_2$ or (b.2) the unexpected behaviour of the execution of $\mathcal{P}$ on A_o was detected during the execution of $\mathcal{P}$ on $B_{(i_2,k_2)}\sigma_2$. The same argument applied to case (b) can be applied to case (b.2) and so on. Therefore, there must be clauses $C_{i_1}, C_{i_2}, \cdots, C_{i_s}$ of $\mathcal{P}$, program points $(i_1, k_1), (i_2, k_2), \cdots, (i_s, k_s)$ in $C_{i_1}, C_{i_2}, \cdots, C_{i_s}$ respectively, and substitutions $\sigma_1, \sigma_2, \cdots, \sigma_s$ such that C_{i_j} has been used to solve $B_{(i_{j-1},k_{j-1})}\sigma_{j-1}$ and $\mathcal{P}$ succeeds on $L_{(i_j,1)}\sigma_j, L_{(i_j,2)}\sigma_j, \cdots, L_{(i_j,k_j-1)}\sigma_j$ for $1 \leq j \leq s$, and $p = (i_s, k_s)$ and $\sigma = \sigma_s$.

The *active execution tree* for $\mathcal{P}$ and A_o is a tree that models the execution of $\mathcal{P}$ on A_o until the unexpected behaviour of the execution is detected. It is constructed as follows. For each $1 \leq u \leq s$, a one-level tree as shown in figure 7.1 is constructed for (i_{u-1}, k_{u-1}), σ_{u-1}, C_{i_u}, (i_u, k_u) and σ_u. The root is labelled with $< (i_{u-1}, k_{u-1}), \sigma_{u-1} >$. The root has k_u children. For each $1 \leq l < k_u$, the root has a child labelled with $L_{(i_u,l)}\sigma_u$ with the leftmost child corresponding to $l = 1$ and the other children being ordered accordingly. The rightmost child of the root is labelled with $< (i_u, k_u), \sigma_u >$.

The active execution tree for $\mathcal{P}$ and A_o results from superimposing, for $2 \leq u \leq s$, the root of the one-level tree for (i_{u-1}, k_{u-1}), σ_{u-1}, C_{i_u}, (i_u, k_u) and σ_u onto the rightmost leaf of the one-level tree for (i_{u-2}, k_{u-2}), σ_{u-2}, $C_{i_{u-1}}$, (i_{u-1}, k_{u-1}) and σ_{u-1}. Each node labelled with

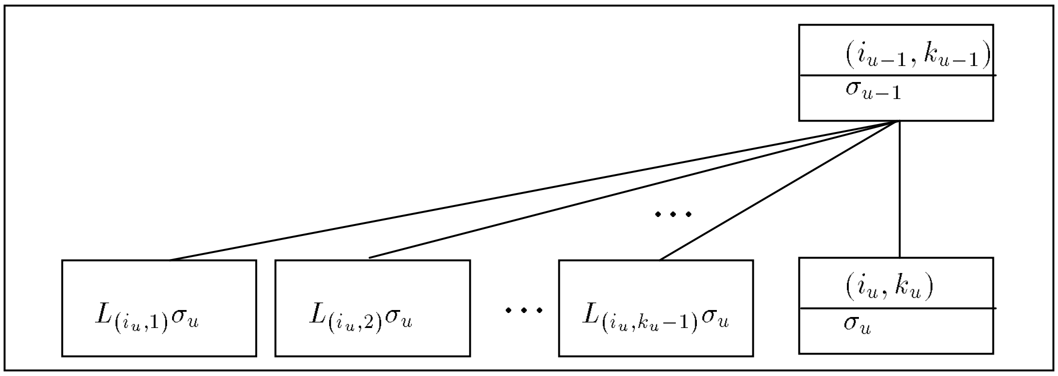
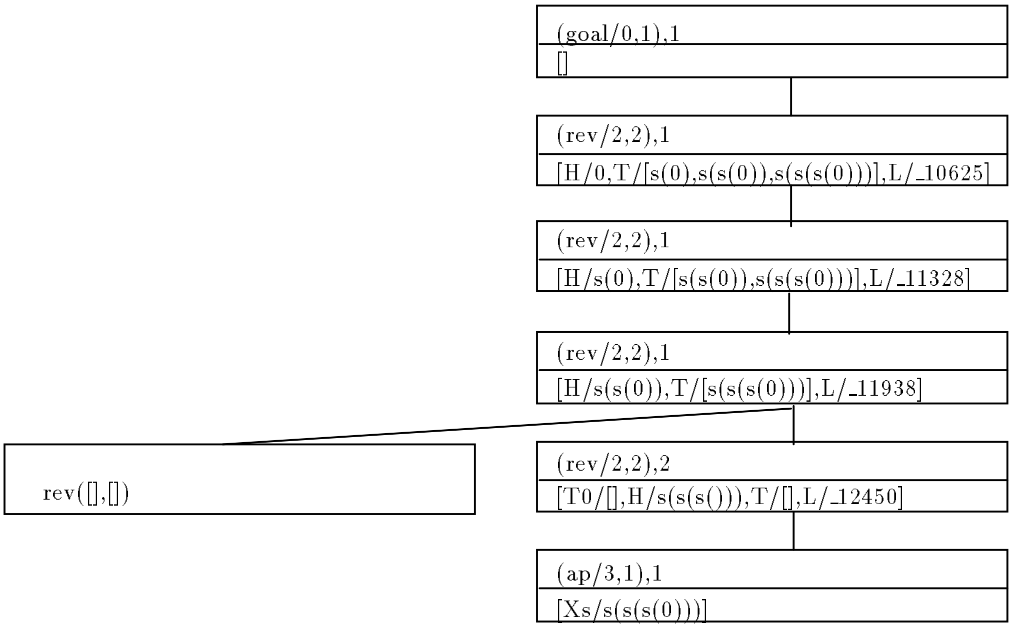


Figure 7.1: One-level Tree for (i_{u-1}, k_{u-1}), σ_{u-1}, C_{i_u}, (i_u, k_u) and σ_u.

$< (i_u, k_u), \sigma_u >$, for $0 \leq u \leq s-1$, corresponds to a procedure call which was being evaluated when the unexpected behaviour of the execution was detected while the node labelled with $< (i_s, k_s), \sigma_s >$ corresponds to the program point and the current substitution at the time the unexpected behaviour of the execution was detected. We call such nodes *active nodes*. The other nodes are all labelled with literals on which $\mathcal{P}$ succeeds and are called *evaluated nodes*.

Example 7.1.14 Let $\mathcal{P}$ be the program in example 1.0.3 and $(type, \{Xs/list(any)\})@v$. *rev*([0,s(0),s(s(0)),s(s(s(0)))],L) is an inadmissibility symptom of $\mathcal{P}$ since the execution of $\mathcal{P}$ on *rev*([0,s(0),s(s(0)),s(s(s(0)))],L) violates $(type, \{Xs/list(any)\})@v$. The following is the active execution tree for $\mathcal{P}$ and *rev*([0,s(0),s(s(0)),s(s(s(0)))],L).



∎

Lemma 7.1.15 Let $\mathcal{P}$ be a program, $\mathcal{M}$ an interpretation and A_o be an operational symptom of $\mathcal{P}$. If T is the active execution tree for $\mathcal{P}$ and A_o and there is an active node in T which is inadmissible then there is a one-level tree S in T such that either (a) $(< p, \sigma >, < q, \theta >)$ is a diverging invocation of $\mathcal{P}$ w.r.t. $\mathcal{M}$ where $< p, \sigma >$ is the root of S and $< q, \theta >$ is the rightmost child of $< p, \sigma >$ or (b) S has a leaf labelled with a negative literal $\neg A_m$ and A_m is an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ or (c) S has a leaf labelled with a positive literal A_w and A_w is an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

PROOF: See p. 180. ∎

Lemma 7.1.15 says that an inadmissibility symptom A_o of $\mathcal{P}$ can be diagnosed by searching the active execution tree for $\mathcal{P}$ and A_o.

7.2 The Diagnoser

We now present our declarative diagnoser for normal programs. This declarative diagnoser will be referred to as π and is presented in Edinburgh Prolog [54].

Let T be a tree and v a node in T. We use $T - T_v$ to denote the tree resulting from deleting T_v, the sub-tree rooted at v, from T and $T \backslash T_v$ to denote the tree resulting from replacing T_v of T with a node v.

The inconsistency diagnosis procedure of π is *inconsistency*$(+A_w, -D)$. When called with an inconsistency symptom A_w of $\mathcal{P}$ w.r.t. $\mathcal{M}$, *inconsistency*/2 succeeds with D being a declarative bug of $\mathcal{P}$ w.r.t. $\mathcal{M}$. *inconsistency*/2 first calls *cpp_1*/2 to construct an $\mathcal{M}$-*cpp* T for $\mathcal{P}$ and A_w. Then *inconsistency*/2 calls *invalid_impl*/2 to find a node L' of T such that $C = RI(T, L')$ is invalid. If C is an instance of a clause of $\mathcal{P}$, *inconsistency*/2 returns with C as its output. Otherwise $C = (\neg A_m \leftarrow \textsf{true})$ with A_m being an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$. In this case, *inconsistency*/2 calls *insufficiency*/2 to diagnose the insufficiency symptom A_m of $\mathcal{P}$ w.r.t. $\mathcal{M}$. The specification for *cpp_1*/2 is that *cpp_1*(A_w, T) succeeds once and only once with T being an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w.

```
inconsistency(Aw,D) :-
    cpp_1(Aw,T),
    invalid_impl(T,C),
    (
        C = (\+Am:-true)
    ->
        insufficiency(Am,D)
    ;
        D = C
    ).
```

```
invalid_impl(T,C) :-
    height(T,1),
    root_impl(T,C).
invalid_impl(T,C) :-
    \+ height(T,1),
    branch_node_1(T,L),
    (
        valid(L)
    ->
        '\'(T,L,T1)
    ;
        '%'(T,L,T1)
    ),
    invalid_impl(T1,C).
```

invalid_impl$(+T,-C)$ succeeds with $C = RI(T,L')$ being invalid in $\mathcal{M}$ for some node L' of T if T is a literal-labelled tree whose root is invalid in $\mathcal{M}$ and each of whose leaves is valid in $\mathcal{M}$. *height*$(+T,-H)$ succeeds with $h(T) = H$. The specification for *branch_node_1*/2 is that *branch_node_1*(T,N) succeeds once and only once with N being a branch node of T. *root_impl*$(+T,-C)$ succeeds with C being the root implication of T. *valid*$(+L)$ succeeds iff L is valid in $\mathcal{M}$. $'\%'(+T,+L,-T1)$ succeeds with $T1 = T_L$ if L is a node of T. $'\backslash'(+T,+L,-T1)$ succeeds with $T1 = T\backslash T_L$ if L is a node of T.

The insufficiency diagnosis procedure of π is *insufficiency*$(+A_m,-D)$. When called with an insufficiency symptom A_m of $\mathcal{P}$ w.r.t. $\mathcal{M}$, *insufficiency*/2 succeeds with D being a declarative bug $\mathcal{P}$ w.r.t. $\mathcal{M}$. *insufficiency*/2 first calls *cps_1*/2 to construct an $\mathcal{M}$-*cps* T for $\mathcal{P} \cup \{\leftarrow A_m\}$. Then *insufficiency*/2 calls *critical*/2 to find a G node of T such that G is a critical node of T w.r.t. $\mathcal{M}$. Let L be the selected literal of G by R. If L is positive, *insufficiency*/2 returns with L as its output. Otherwise, $L = \neg A_w$ with A_w being an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$. In this case *insufficiency*/2 calls *inconsistency*/2 to diagnose inconsistency symptom A_w of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

```
insufficiency(Am,D) :-
    cps_1(':-'(Am),T),
    critical(T,G),
    selected(G,L),
    (
        L = \+Aw
    ->
        inconsistency(Aw,D)
    ;
        D = L
    ).
```

```
critical(T,G) :-
    height(T,0),
    root(T,G).
critical(T,G) :-
    \+ height(T,0),
    non_root_1(T,':-'(W)),
    (
        satisfiable(W)
    ->
        '%'(T,':-'(W),T1)
    ;
        '-'(T,':-'(W),T1)
    ),
    critical(T1,G).
```

cps_1$(+G, -T)$ succeeds with T being an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{G\}$ via a fixed fair computation rule R. *selected*$(+G, -L)$ succeeds with L is the selected literal of G by R.

critical$(+T, -G)$ succeeds with G being a critical node of T w.r.t. $\mathcal{M}$ if T is a goal-labelled tree such that W is satisfiable in $\mathcal{M}$ where $\leftarrow W$ is the root of T. *root*$(+T, -G)$ succeeds with G being the root of T. *non_root_1*$(+T, -G)$ succeeds once and only once with G being a node of T other than the root of T. '-'$(+T, +G, -T1)$ succeeds with $T1 = T - T_G$ if G is a node T. *satisfiable*$(+W)$ succeeds iff W is satisfiable in $\mathcal{M}$.

At a certain stage of diagnosis, $\mathcal{M}'$ embodies the knowledge about the intended interpretation $\mathcal{M}$ that has been acquired by the diagnoser. When the diagnoser constructs an $\mathcal{M}$-*cpp* (or an $\mathcal{M}$-*cps*), it uses $\mathcal{M}'$ to decide if a literal L is *known* to be valid in $\mathcal{M}$ (or if a conjunction W of literals is *known* to be unsatisfiable in $\mathcal{M}$). These judgements are made by a set of rules based on $\mathcal{M}'$ and other knowledge that the diagnoser has acquired. The following are some example rules where A is an atom, L is a literal and W is a conjunction of literals.

- A is valid in $\mathcal{M}$ if A is an instance of another atom that is valid in $\mathcal{M}$.
- $\neg A$ is valid in $\mathcal{M}$ if A is unsatisfiable in $\mathcal{M}$.
- W is unsatisfiable in $\mathcal{M}$ if W contains both A and $\neg A$ for some A.
- W is unsatisfiable in $\mathcal{M}$ if W contains L and L is unsatisfiable in $\mathcal{M}$.

It is possible that the validity of a literal L in $\mathcal{M}$ or the unsatisfiability of a conjunction W of literals in $\mathcal{M}$ may not be detected by the set of rules at a certain stage either because $\mathcal{M}'$ does not have enough information or because the set of rules are incomplete. This does not affect the soundness and the completeness of π which are proved in the next section.

Example 7.2.1 We now show a session using the diagnoser π. The program is a quicksort program that has three bugs. A bug is indicated by a comment which also indicates the correction of the bug. When a query is imposed on the oracle by *satisfiable*/1 or *valid*/1, variables in the atom concerned are replaced with generated mnemonic names that should be understood to be local to the query. Before the diagnosis session begins, the acquired part $\mathcal{M}'$ of the intended interpretation $\mathcal{M}$ contains the following. (1) any call to *ap*/3 or any built-in procedure does not result in any bug symptom, and (2) $qs([X],[X])$ is valid in $\mathcal{M}$. The top-down zooming strategy [81] is used to search both $\mathcal{M}$-*cpp* and $\mathcal{M}$-*cps* .

```
qs([X|L],L0) :-
  pt(L,X,L1,L2),
  qs(L1,L3),
  qs(L2,L4),
  ap([X|L3],L4,L0). %ap(L3,[X|L4],L0)
qs([],[]).

pt([X|L],Y,L1,[X|L2]) :-
  %Y<X,
  pt(L,Y,L1,L2).
pt([X|L],Y,[X|L1],L2) :-
  Y=<X,          %Y>=X,
  pt(L,Y,L1,L2).
pt([],X,[],[]).

ap([X|L1],L2,[X|L3]) :-
  ap(L1,L2,L3).
ap([],L,L).

| ?- qs([2,1],L).
     L = [2,1]
yes
| ?- inconsistency(qs([2,1],[2,1]),D).
Is qs([],[]) valid? y.
Is pt([1],2,[],[1]) valid? n.
Is pt([],2,[],[]) valid? y.

D=(pt([1],2,[],[1]):-pt([],2,[],[]))
yes
| ?-
% D is an instance of the first clause
% for pt/4; the user corrects the  bug
% in that clause.

| ?- qs([2,1],L).
no
| ?- insufficiency(qs([2,1],L),D).
Is pt([1],2,A,B) satisfiable? y.
     A = ? [1].
     B = ? [].

D=pt([1],2,_12938,_12939)
yes
| ?-
% The  user  corrects  the  bug in the
% second clause of pt/4.

| ?- qs([2,1],L).
     L = [2,1]
yes
| ?- inconsistency(qs([2,1],[2,1]),D).

D=(qs([2,1],[2,1]) :-
     pt([1],2,[1],[]),
     qs([1],[1]),
     qs([],[]),
     ap([2,1],[],[2,1]))
yes
| ?-
% D is an instance of the first clause
% of  qs/2. The user corrects the  bug
% in that clause.

| ?- qs([2,1],L).
     L = [1,2]
yes
| ?-
```

∎

The diagnosis procedure of π for operational symptoms is *operational*$(+A_o, -D)$. Given an operational symptom A_o of $\mathcal{P}$, *operational*/2 first calls *aet*/2 which simulates the execution of $\mathcal{P}$ on A_o up to the detection of an unexpected behaviour of the execution and constructs the active execution tree T for $\mathcal{P}$ and A_o. *operational*/2 then calls *segment*/2 to find in T a one-level segment S whose root is admissible and whose rightmost child is inadmissible. If *segment*/2 finds such a segment S then *operational*/2 calls *diverge*/2 which finds either an inconsistent clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$ or an incompletely covered atom $\mathcal{P}$ w.r.t. $\mathcal{M}$ or a diverging invocation of $\mathcal{P}$ w.r.t. $\mathcal{M}$. If T does not contain any one-level segment whose root is admissible and whose rightmost child is inadmissible, then *operational*/2 returns *no_diagnosis*, indicating that no bug has been found as a result of the diagnosis.

```
operational(Ao,D) :-
    aet(Ao,T),
    (
        segment(T,S)
    ->
        diverge(S,D)
    ;
        D = no_diagnosis
    ).

segment(T,S)            :-
    \+ height(T,0),
    rightmost(T,PS1),
    (
        admissible(PS1)
    ->
        '%'(T,PS1,T1),
        segment(T1,S)
    ;
        '\'(T,PS1,S)
    ).

diverge(S,(PS0,PS1)) :-
    degree(S,1),
    root(S,PS0),
    rightmost(S,PS1).

diverge(S,D)            :-
    \+ degree(S,1),
    leftmost(S,\+Am),
    (
        satisfiable(Am)
    ->
        insufficiency(Am,D)
    ;
        '-'(S,\+Am,S1),
        diverge(S1,D)
    ).
diverge(S,D)            :-
    \+ degree(S,1),
    leftmost(S,Aw),
    \+ Aw = \+(_),
    (
        valid(Aw)
    ->
        '-'(S,Aw,S1),
        diverge(S1,D)
    ;
        inconsistency(Aw,D)
    ).
```

The specification for *aet*/2 is that if A_o is an operational symptom of $\mathcal{P}$ then $aet(A_o, T)$ simulates the execution of $\mathcal{P}$ on A_o up to the detection of an unexpected behaviour of the execution, and constructs and returns the active execution tree T for $\mathcal{P}$ and A_o. The specification for *leftmost*/2 is that if T is a tree then $leftmost(T, C)$ succeeds with C being the leftmost child of the root of T. The specification for *rightmost*/2 is that if T is a tree then $rightmost(T, C)$ succeeds with C being the rightmost child of the root of T. The specification for *degree*/2 is that if T is a tree then $degree(T, N)$ succeeds with N being the number of children of the root of T.

Example 7.2.2 This example shows a session of using π for diagnosing the inadmissibility symptom in example 7.1.14.

```
| ?- operational(rev([0,s(0),s(s(0)),s(s(s(0)))],L),D).

Look at the following program state.
    rev([H|T],L) :-
              H <- 0  T <- [s(0),s(s(0)),s(s(s(0)))]  L <- _10652
        rev(T,T0),
        ap(T0,H,L).
Is this state admissible? y
Look at the following program state.
    rev([H|T],L) :-
              H <- s(0)  T <- [s(s(0)),s(s(s(0)))]  L <- _11328
        rev(T,T0),
        ap(T0,H,L).
Is this state admissible? y
Look at the following program state.
    rev([H|T],L) :-
              H <- s(s(0))  T <- [s(s(s(0)))]  L <- _11938
        rev(T,T0),
        ap(T0,H,L).
Is this state admissible? y
Look at the following program state.
    rev([H|T],L) :-
        rev(T,T0),
              T0 <- []  H <- s(s(s(0)))  T <- []  L <- _12450
        ap(T0,H,L).
Is this state admissible? y
Look at the following program state.
    ap([],Xs,Xs).
              Xs <- s(s(s(0)))
Is this state admissible? n

D = ((((rev/2,2),2),[T0/[],H/s(s(s(0))),T/[],L/_12450])
     ,
     (((ap/3,1),1),[Xs/s(s(s(0)))])
    )
```

∎

7.3 Soundness and Completeness

In this section we establish the soundness and the completeness of π. We first consider the soundness and the completeness of *invalid_impl*/2.

Lemma 7.3.1 Let $\mathcal{M}$ be an interpretation, T a literal-labelled tree such that the root of T is invalid in $\mathcal{M}$ and each leaf of T is valid in $\mathcal{M}$, and L^1 a node of T.

(1) If L^1 is invalid in $\mathcal{M}$, then there is a node L' of T_{L^1} such that $RI(T_{L^1}, L')$ is invalid in $\mathcal{M}$.

(2) If L^1 is valid in $\mathcal{M}$, then there is a node L' of $T \backslash T_{L^1}$ such that $RI(T \backslash T_{L^1}, L')$ is invalid in $\mathcal{M}$.

PROOF: In case (1), T_{L^1} is a literal-labelled tree such that the root of T_{L^1} is invalid in $\mathcal{M}$ and each leaf of T_{L^1} is valid in $\mathcal{M}$. In case (2), $T \backslash T_{L^1}$ is a literal-labelled tree such that the root of $T \backslash T_{L^1}$ is invalid in $\mathcal{M}$ and each leaf of $T \backslash T_{L^1}$ is valid in $\mathcal{M}$. So, the lemma follows from lemma 7.1.5. ∎

Lemma 7.3.2 Let $\mathcal{M}$ be an interpretation and T a literal-labelled tree whose root is invalid in $\mathcal{M}$ and each of whose leaves is valid in $\mathcal{M}$. If $invalid\_impl(T, C) \in SS(\pi)$, then C is invalid in $\mathcal{M}$ and $C = RI(T, L')$ for a node L' of T.

PROOF: See p. 181. ∎

invalid_impl/2 is determinate since *height*/2, *root_impl*/2, *branch_node_1*/2 and *valid*/1 are determinate. So, for any literal-labelled tree whose root is invalid in $\mathcal{M}$ and each of whose leaves is valid in $\mathcal{M}$, there is at most one C such that $invalid\_impl(T, C) \in SS(\pi)$. What remains to be addressed is whether there is a node L' of T such that $C = RI(T, L')$ and $invalid\_impl(T, C) \in SS(\pi)$.

Lemma 7.3.3 Let $\mathcal{M}$ be an interpretation. If T is a literal-labelled tree whose root is invalid in $\mathcal{M}$ and each of whose leaves is valid in $\mathcal{M}$, then there is a node L' of T such that $C = RI(T, L')$ and $invalid\_impl(T, C) \in SS(\pi)$.

PROOF: See p. 181. ∎

We now address the soundness and the completeness of *critical*/2.

Lemma 7.3.4 Let $\mathcal{M}$ be an interpretation, T a goal-labelled tree, $\leftarrow W$ the root of T, W be satisfiable in $\mathcal{M}$, and $\leftarrow W^1$ a node of T.

(1) If W^1 is satisfiable in $\mathcal{M}$, then there is node $\leftarrow W'$ of $T_{\leftarrow W^1}$ such that $\leftarrow W'$ is a critical node of T w.r.t. $\mathcal{M}$.

(2) If W^1 is unsatisfiable in $\mathcal{M}$, then there is a node $\leftarrow W'$ of $T - T_{\leftarrow W^1}$ such that $\leftarrow W'$ is a critical node of T w.r.t. $\mathcal{M}$.

PROOF: See p. 182. ∎

Lemma 7.3.5 Let $\mathcal{M}$ be an interpretation, T a goal-labelled tree, $\leftarrow W$ the root of T, and W be satisfiable in $\mathcal{M}$. If $critical(T, G) \in SS(\pi)$ then G is a critical node of T w.r.t. $\mathcal{M}$.

PROOF: See p. 182. ∎

critical/2 is determinate since *height*/2, *root*/2, *non_root_1*/2 and *satisfiable*/1 are determinate. For any goal-labelled tree T with root $\leftarrow W$ such that W is satisfiable in $\mathcal{M}$, there is at most one critical node G of T w.r.t. $\mathcal{M}$ such that $critical(T,G) \in SS(\pi)$. What remains to be addressed is whether or not there is a critical node G of T w.r.t. $\mathcal{M}$ such that $critical(T,G) \in SS(\pi)$.

Lemma 7.3.6 Let $\mathcal{M}$ be an interpretation. If T is a goal-labelled tree, $\leftarrow W$ is the root of T and W is satisfiable in $\mathcal{M}$, then there is a critical node G of T w.r.t. $\mathcal{M}$ such that $critical(T,G) \in SS(\pi)$.

PROOF: See p. 183. ∎

The following lemma states that *segment*/2 is sound and complete for diagnosing inadmissibility symptoms.

Lemma 7.3.7 Let A be an operational symptom of $\mathcal{P}$ and T be the active execution tree for $\mathcal{P}$ and A.

(1) If $segment(T,S) \in SS(\pi)$ then S is a one-level tree whose root $< p, \sigma >$ is admissible, whose righmost leaf $< q, \theta >$ is inadmissible and whose other leaves are literals on which $\mathcal{P}$ succeeds.

(2) If there is an active node in T which is inadmissible then there is some S such that $segment(T,S) \in SS(\pi)$ and S is a one-level tree whose root $< p, \sigma >$ is admissible, whose righmost leaf $< q, \theta >$ is inadmissible and whose other leaves are literals on which $\mathcal{P}$ succeeds.

PROOF: Immediate. ∎

The soundness and the completeness of *diverge*/2 are stated by the following lemma.

Lemma 7.3.8 Let S be a one-level tree whose root $< p, \sigma >$ is admissible, whose rightmost child $< q, \theta >$ of S is inadmissible and whose other leaves are literals on which $\mathcal{P}$ succeeds.

(1) If $diverge(S,D) \in SS(\pi)$ then either D is a declarative bug of $\mathcal{P}$ w.r.t. $\mathcal{M}$ or $D = (< p, \sigma >, < q, \theta >)$ and every leaf of S other than $< q, \theta >$ is valid in $\mathcal{M}$.

(2) There is some D such that $diverge(S,D) \in SS(\pi)$ and either D is a declarative bug of $\mathcal{P}$ w.r.t. $\mathcal{M}$ or $D = (< p, \sigma >, < q, \theta >)$ and every leaf of S other than $< q, \theta >$ is valid in $\mathcal{M}$.

PROOF: See p. 183. ∎

Finally, we prove the soundness and the completeness of π.

Theorem 7.3.9 (Soundness of π) *Let $\mathcal{P}$ be a normal program, A an atom and $\mathcal{M}$ an interpretation.*

(1) If A is an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ and inconsistency$(A, D) \in SS(\pi)$, then D is a declarative bug of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

(2) If A is an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ and insufficiency$(A, D) \in SS(\pi)$, then D is a declarative bug of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

(3) If A is an operational symptom of $\mathcal{P}$ and operational$(A, D) \in SS(\pi)$ and $D \neq$ no_diagnosis then D is either a declarative bug of $\mathcal{P}$ w.r.t. $\mathcal{M}$ or a diverging invocation of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

PROOF: See p. 184. ∎

Theorem 7.3.10 (Completeness of π) *Let $\mathcal{P}$ be a normal program, A an atom and $\mathcal{M}$ an interpretation.*

(1) If A is an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$, then there is some D such that D is a declarative bug of $\mathcal{P}$ w.r.t. $\mathcal{M}$ and inconsistency$(A, D) \in SS(\pi)$.

(2) If A is an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$, then there is some D such that D is a declarative bug of $\mathcal{P}$ w.r.t. $\mathcal{M}$ and insufficiency$(A, D) \in SS(\pi)$.

(3) If A is an inadmissibility symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ then there is some D such that operational$(A, D) \in SS(\pi)$ and D is either a declarative bug of $\mathcal{P}$ w.r.t. $\mathcal{M}$ or a diverging invocation of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

PROOF: See p. 185. ∎

7.4 Discussion

This section compares our diagnoser with the diagnosers reported in the literature with respect to the search strategy and the search space, two major factors that affect the quantity of queries.

7.4.1 Generality

When called with an inconsistency symptom A_w of $\mathcal{P}$ w.r.t. $\mathcal{M}$, *inconsistency*/2 calls *invalid_impl*/2 to search an $\mathcal{M}$-*cpp* T for $\mathcal{P}$ and A_w to find a root implication C of a sub-tree of T such that C

is invalid in $\mathcal{M}$. When called with an insufficiency symptom A_m of $\mathcal{P}$ w.r.t. $\mathcal{M}$, *insufficiency*/2 calls *critical*/2 to search an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ to find a node that is a critical node of T w.r.t. $\mathcal{M}$.

The specification for *branch_node_1*/2 is that *branch_node_1*(T, N) succeeds once and only once with N being a branch node of T. The different implementations of *branch_node_1*/2 will result in different performances of the inconsistency diagnosis procedure *inconsistency*/2 in terms of the quantity of queries.

A top-down inconsistency diagnosis procedure based on tree search [33, 81, 125] can be obtained by using an implementation of *branch_node_1*/2 such that *branch_node_1*(T, N) succeeds once and only once with N being a child of the root of T. A bottom-up inconsistency diagnosis procedure based on tree search [125] can be obtained by using an implementation of *branch_node_1*/2 such that *branch_node_1*(T, N) succeeds with N being the parent node of a leaf node of T. The divide-and-query inconsistency diagnosis procedure [121, 122] can be obtained through an implementation of *branch_node_1*/2 such that *branch_node_1*(T, N) succeeds once and only once with N being a node of T such that $|n(T_N) - n(T)/2| \leq |n(T_{N'}) - n(T)/2|$ for any other node N' of T. The top-down zooming inconsistency diagnosis procedure [81] can be obtained by using an implementation of *branch_node_1*/2 such that *branch_node_1*(T, N) succeeds once and only once with N being a node of T satisfying either (1) N is a node of T other than the root of T, and (2) N and the root of T have the same predicate name, and (3) N is not subordinate to any node of T that satisfies (1) and (2), or N is a child of the root of T when no node of T satisfies (1) and (2).

The specification for *non_root_1*/2 is that *non_root_1*(T, N) succeeds once and only once such that N is a node of T other than the root node of T. The different implementations of *non_root_1*/2 will result in different performances of the insufficiency diagnosis procedure *insufficiency*/2 in terms of the quantity of queries.

A top-down insufficiency diagnosis procedure can be obtained by using an implementation of *non_root_1*/2 such that *non_root_1*(T, N) succeeds once and only once with N being a child of the root of T. A bottom-up insufficiency diagnosis procedure can be obtained by using an implementation of *non_root_1*/2 such that *non_root_1*(T, N) succeeds with N being a leaf node of T.

insufficiency/2 can be specialised resulting in a divide-and-query insufficiency diagnosis procedure through an implementation of *non_root_1*/2 such that *non_root_1*(T, N) succeeds with N being a node of T such that $|n(T_N) - n(T)/2| \leq |n(T_{N'}) - n(T)/2|$ for any other node N' of

T.

The formulation of π not only enables standard tree search strategies such as top-down, bottom-up and divide-and-query to be used, but also allows more flexible strategies to be used as long as these strategies conform to the specifications for *branch_node_1*/2 and *non_root_1*(T, N). This provides us with a platform for evaluating various strategies as well as tailoring the declarative diagnoser to a user who prefers a particular search strategy.

7.4.2 Search Space

The search space of a declarative diagnoser is one of the major factors that affects the quantity of queries. We briefly compare the search space of our declarative diagnoser π with the search spaces of the declarative diagnosers in the literature.

Suppose that A_w is the inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ that is to be diagnosed. An inconsistency diagnosis procedure based on tree search [33, 81, 121, 122, 125], including the divide-and-query diagnoser [121, 122], is a specialised version of *inconsistency*/2. These inconsistency diagnosis procedures search for an inconsistent clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$ in the set of all the clause instances that are used in one successful derivation of $\mathcal{P} \cup \{\leftarrow A_w\}$, that is, the clause instances that are used in one proof tree for $\mathcal{P}$ and A_w. The search space of *inconsistency*/2 is the set of all the clause instances that are used in one $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w. Because an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w is smaller than a proof tree for $\mathcal{P}$ and A_w, *inconsistency*/2 has a smaller search space.

There are inconsistency diagnosis procedures that are not based on tree search [43, 73, 74, 121, 122, 144]. Such inconsistency diagnosis procedures search a larger space than *inconsistency*/2. We exemplify this through the single stepping inconsistency diagnosis procedure [121, 122]. The single stepping inconsistency diagnosis procedure simulates Prolog's execution of $\mathcal{P}$ on A_w. Whenever a call A' has been executed successfully with a computed answer θ, the oracle is asked if $A'\theta$ is valid in $\mathcal{M}$. If $A'\theta$ is valid in $\mathcal{M}$, the single stepping inconsistency diagnosis procedure continues to simulate Prolog's execution of the remaining calls. Otherwise, $A'\theta$ is invalid in $\mathcal{M}$. Let $C \equiv H' \leftarrow L'_1, L'_2, \cdots, L'_m$ be the clause instance used to solve A'. $(L'_1, L'_2, \cdots, L'_m)\theta$ is already known to be valid in $\mathcal{M}$. Therefore, $C\theta$ is an inconsistent clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$. Let R' be the left-to-right computation rule. The single stepping inconsistency diagnosis procedure searches for an inconsistent clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$ in the set of all the clause instances used in the first successful SLDNF-derivation of $\mathcal{P} \cup \{\leftarrow A_w\}$ via R' and all the clause instances used in all the unsuccessful SLDNF-derivations of $\mathcal{P} \cup \{\leftarrow A_w\}$ via R' that are previous to the successful SLDNF-derivation of $\mathcal{P} \cup \{\leftarrow A_w\}$. The other inconsistency diagnosis procedures that are not based on tree search can be shown to have larger spaces than *inconsistency*/2 as well.

Suppose that A_m is the insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ to be diagnosed. *insufficiency*/2 searches for an incompletely covered atom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ in the set of all the selected atoms of all the nodes of an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ via a fixed computation rule R. The search space of the insufficiency diagnosis procedures presented in [33, 81, 105, 106] is the set of all the selected atoms of all the nodes of an SLDNF-tree for $\mathcal{P} \cup \{\leftarrow A_m\}$. Because an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ is smaller than an SLDNF-tree for $\mathcal{P} \cup \{\leftarrow A_m\}$, *insufficiency*/2 has a smaller search space than these insufficiency diagnosis procedures.

The insufficiency diagnosis procedures presented in [43, 73, 74, 144] have larger search spaces than *insufficiency*/2 because, given an insufficiency symptom A_m of $\mathcal{P}$ w.r.t. $\mathcal{M}$, their search spaces are larger than the set of all the selected atoms of all the nodes of an SLDNF-tree for $\mathcal{P} \cup \{\leftarrow A_m\}$. See [99] for a detailed analysis of the search spaces of some insufficiency diagnosis procedures in the literature.

7.4.3 Search Space Pruning versus Oracle Automation

The objective of reducing the quantity of queries has also been pursued by fully or partly automating the oracle. Diagnosers in [30, 31, 37, 70] use a full specification. A full specification makes it possible to completely avoid querying the user because any query about the intended interpretation $\mathcal{M}$ can be answered by using the specification. Diagnosers in [32, 33, 66] use assertions about the intended interpretation $\mathcal{M}$ to answer queries. Whenever a query is necessary, these diagnosers will try to answer the query by using only the assertions. Those queries that cannot be answered this way are imposed on the oracle. The assertions are descriptions of the acquired part $\mathcal{M}'$ of the intended interpretation $\mathcal{M}$.

Given an inconsistency symptom A_w of $\mathcal{P}$ w.r.t. $\mathcal{M}$, our diagnoser constructs an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w using $\mathcal{M}'$. The effect is equivalent to pruning a proof tree for $\mathcal{P}$ and A_w before it is searched. Similarly, given an insufficiency symptom A_m of $\mathcal{P}$ w.r.t. $\mathcal{M}$, our diagnoser constructs an $\mathcal{M}$-*cps* for $\mathcal{P} \cup \{\leftarrow A_m\}$ using $\mathcal{M}'$. The effect is equivalent to pruning an SLDNF-tree for $\mathcal{P} \cup \{\leftarrow A_m\}$ before it is searched.

Using $\mathcal{M}'$ to prune the search space before it is searched rather than to answer queries makes sense. Firstly, a smaller search space means a smaller upper bound for the quantity of queries [74]. Secondly, given an inconsistency symptom A_w of $\mathcal{P}$ w.r.t. $\mathcal{M}$, if a proof tree for

$\mathcal{P}$ and A_w is searched, then a search strategy may select a node L and query the oracle about the validity of L if the validity of L cannot be decided by using $\mathcal{M}'$. If L is subordinate to another node L' in the proof tree such that L' is valid in $\mathcal{M}'$, then an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A_w that is constructed using $\mathcal{M}'$ will not contain node L because L' is valid in $\mathcal{M}'$. Therefore, this query can be spared. A similar argument applies to insufficiency diagnosis. This does not apply to the top-down search strategy. However, the top-down strategy may not be either the optimal strategy for the diagnosis problem at hand or the strategy preferred by the user. We share with [34] the opinion that the user should be allowed to choose his own strategy.

7.4.4 Diagnosis of Operational Symptoms

Work on logic program diagnosis has been focused on the diagnosis of inconsistency and insufficiency symptoms with little effort on the diagnosis of operational symptoms. Shapiro developed a diagnosis procedure for non-termination symptoms [122]. The diagnosis is based on a well-founded relation among procedure calls. Given a program $\mathcal{P}$ and a non-termination symptom A_o of $\mathcal{P}$, the diagnosis procedure searches, among the sequence of program clauses used in the execution of $\mathcal{P}$ and A_o up to the detection of the unexpected behaviour of the execution, for two consecutive procedure calls - heads of two consecutive clauses which violate the well-founded relation. The oracle is required to decide if it is well-founded that the execution of one procedure call leads to the execution of another procedure call. Our procedure *operational*/2 is an adaptation of Shapiro's diagnosis procedure for non-termination symptoms. Instead of querying on a well-founded relation among procedure calls, it queries on the admissibility of a substitution at a program point. Due to this difference, *operational*/2 returns a diverging invocation of $\mathcal{P}$ w.r.t. $\mathcal{M}$ whilst Shapiro's diagnosis procedure returns a diverging clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$ and the well-founded relation. A diverging clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$ and the well-founded relation is a clause instance $H \leftarrow L_1, \cdots, L_i, \cdots, L_n$ of $\mathcal{P}$ such that H is not in the well-founded relation to L_i and $L_1, \cdots, L_{i-1}$ are valid in $\mathcal{M}$. Shapiro's diagnosis procedure is applicable only if the unexpected behaviour of the execution of $\mathcal{P}$ on A_o is not detected at the exit point of a clause, which may be the case for inadmissibility symptoms.

7.5 Summary

We have presented the declarative diagnoser π for normal logic programs and established its soundness and its completeness. The inconsistency diagnosis procedure and the insufficiency diagnosis procedure in π have a smaller search space than the those in the literature when diagnosing an inconsistency or insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$.

Various tree search strategies can be used with the inconsistency diagnosis procedure and the insufficiency diagnosis procedure in π. The proof tree search strategy and the SLDNF tree search strategy affect the quantity of the queries imposed on the oracle.

Chapter 8

Conclusion and Further Work

8.1 Conclusion

We have so far reported our work on abstract interpretation, bug detection and bug diagnosis in normal logic programs. Abstract interpretation is used as an approach to program analysis the results of which are used in the detection of declarative and operational bug symptoms. Bug detection is accomplished by the combination of static analyses and program testing. Bug diagnosis is performed in a declarative manner in the sense that the oracle has only to know the intended interpretation for the program. We have made some achievements in this work.

In respect of abstract interpretation, we have developed two fixed-point generic abstract semantics $F^{\flat}_{\mathcal{P},\mathcal{G}}$ and $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ for forward analyses of normal logic programs under the SLDNF via the left-to-right computation rule. $F^{\flat}_{\mathcal{P},\mathcal{G}}$ infers data descriptions for edges in the program graph. The existing generic abstract semantics are optimisation-oriented and usually infer data descriptions for program points, data descriptions for entry points and success points or patterns of procedure calls and procedure successes. $F^{\flat}_{\mathcal{P},\mathcal{G}}$ and $F^{\diamond}_{\mathcal{P},\mathcal{G}}$ deal explicitly with negation as failure while the existing generic abstract semantics do not. The work on abstract interpretation at an early stage of this research was presented at the *IEEE Computer Society 1992 International Conference on Computer Languages* [77] and the *4th UK Conference on Logic Programming* [78].

We have also developed a generic abstract domain and a generic abstract unification algorithm for a class of analyses which infer pair sharing and aliasing information together with descriptions of the values of variables. The generic abstract domain and the generic abstract unification algorithm have been specialised for mode and type analyses. Mode information is derived together with pair sharing and aliasing information. All the mode descriptions but **iu** in the literature can be expressed in mode descriptions in $\wp(\triangle)$. **e** is expressed by $\emptyset$, **f** by $\{ⓕ\}$,

g by $\{ⓖ\}$, **nf** by $\{ⓖ, ⓞ\}$ and **a** by $\triangle$. Type information is also derived together with pair sharing and aliasing information. The abstract domain we have presented is richer than that used by Horiuchi and Kanamori [52, 62] who used a mono-variable-typing to approximate a set of substitutions. Horiuchi and Kanamori [52, 62] do not address the problem of an infinite abstract domain. We use a normalisation function choosing from two classes to enforce a finite abstract domain. This problem can also be resolved by designing a pair of widening/narrowing operators [22]. Horiuchi and Kanamori [52, 62] required that each object function symbol is typed by a type definition. We relax this requirement by allowing default type definitions. The default type definitions and choice of normalisation functions can be used to perform depth and stump analyses.

In respect of bug detection, we have developed a method for the detection of bug symptoms in normal logic programs. An earlier version of this method was presented at the *1993 International Conference on Formal Methods and Their Applications* [79]. The method combines static analysis and program testing to detect bug symptoms. Logic program testing is an area which has rarely been looked at. There are only a few publications in this area. It does not appear that there has been any other work on using abstract interpretation in logic program testing. Yan [143] studies declarative testing using a reliable test data set. What is not covered by Yan is how a reliable test data set is selected. Informally, a test data set T is reliable for a program $\mathcal{P}$ and a specification S if that $\mathcal{P}$ computes the same value as S upon each input in T implies that $\mathcal{P}$ computes the same value as S upon each input at which $\mathcal{P}$ is defined. A finite reliable test data set T for program $\mathcal{P}$ and specification S exists, but there is no effective procedure to either generate or recognise a reliable test data set [53, 91]. Dershowitz *et al.* [30, 31, 70] study logic program testing when an executable specification for the program is available. The program is exhaustively compared with the specification when a goal to be tested has been given. This is impractical because both the intended interpretation for the program specified by the specification and the actual interpretation of the program are usually infinite. We generate a finite test data set with respect to a set of regularity and uniformity hypotheses.

In respect of bug diagnosis, we have developed a declarative diagnoser for normal logic programs. The diagnoser uses the acquired part of the intended interpretation for the program to prune the search spaces for the diagnosis of declarative symptoms. The part of the diagnoser devoted to the diagnosis of the declarative symptoms was presented at the *5th International Conference on Logic Programming and Automated Reasoning* [76]. The inconsistency diagnosis procedure and the insufficiency diagnosis procedure in π have a smaller search space than the those in the literature when diagnosing an inconsistency or insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$. Various tree search strategies can be used with the inconsistency diagnosis procedure and the insufficiency diagnosis procedure in π. The proof tree search strategy and the SLDNF-tree search strategy affect the quantity of the queries imposed on the oracle.

In conclusion, the objective of this work, to develop an automated bug detection method for normal logic programs and to improve the declarative bug diagnosis method for normal logic programs, has been achieved through the methods presented in Chapters 4-7.

8.2 Further Work

The work in our research can be extended in several dimensions. In the language dimension, the prototype debugging system is to be adapted to cover more language constructs such as those found in Prolog. To accommodate these constructs, the generic abstract semantics and the diagnosis procedures need to be adapted accordingly.

In the program analysis dimension, one direction would be to apply the generic abstract semantics for more analyses that may be useful in bug detection, such as complexity [26], termination [133], and determinacy [114] analyses. To specialise a generic abstract semantics, say, $F^{\flat}_{\mathcal{P},\mathcal{G}}$ or $F^{\diamond}_{\mathcal{P},\mathcal{G}}$, for an analysis, it is necessary to investigate if the generic abstract semantics is capable of performing the analysis. If so, it is necessary to develop an abstract domain and an abstract unification algorithm for the analysis. It is desirable to specialise the generic abstract domain and the generic abstract unification algorithm in Chapter 5 for the analysis. For the purpose of bug detection, it is necessary to develop an algorithm for test data generation corresponding to the analysis. Another direction is to integrate different analyses, for instance, the mode and type analyses in Chapter 5.

In the bug detection dimension, we are interested in the automated generation of assertions in the presence of specifications for modes and/or types of procedure calls and procedure successes. A procedure is very often written in a way that it is expected to be called with arguments of particular modes and/or types and to succeed with arguments of particular modes and/or types. The built-in *sort*/2, for instance, is expected to be called with its first argument being a list of natural numbers. An assertion to be associated with each program point preceding each call to *sort*/2 should be generated to check statically if the expectation is satisfied. *sort*/2 is also expected to succeed with both arguments being lists of natural numbers. Similarly, an assertion to be associated with each program point following each call to *sort*/2 should be generated to check statically if the expectation is satisfied. Automated generation of assertions will relieve the burden on the programmer of associating assertions with program points.

In the bug diagnosis dimension, a direction for further research would be to study the use of the results of program analyses to reduce the number of queries imposed on the oracle. It would be interesting to use the results of program analyses to adapt the proof tree search strategy and the SLDNF-tree search strategy.

Appendix A

Proofs

This appendix contains proofs of lemmas and theorems appearing in the thesis that were considered to be too long to appear in the body of the thesis as well as some basic results on unification. Section A.1 contains the basic results on unification that are used in the body of the thesis and in the appendix. The long proofs of the lemmas and theorems in Chapter 4, 5 and 7 are given in sections A.2-A.4 respectively.

A.1 Basic Results on Unification

Lemma A.1.1 Let a and b be terms or atoms, and ρ a renaming substitution such that $(vars(a\rho) \cup vars(\rho(\phi))) \cap (vars(b) \cup vars(\psi)) = \emptyset$. If $(a\rho)(\rho(\phi))$ and $b\psi$ unify then $a\rho$ and b unify.

Proof: Let $a\rho \equiv a'$ and $\rho(\phi) \equiv \phi'$. If $a'\phi'$ and $b\psi$ unify then there is a substitution θ such that $a'\phi'\theta \equiv b\psi\theta$. We have that $vars(a') \cap dom(\psi) = \emptyset$ and $rang(\psi) \cap dom(\phi') = \emptyset$ and $vars(b\psi) \cap dom(\phi') = \emptyset$. Hence, $a'\psi\phi'\theta \equiv b\psi\phi'\theta$. Therefore, $a\rho$ and b unify. ∎

Lemma A.1.2 Let A and B be two atoms, and ρ_1 and ρ_2 be two renaming substitutions such that

$$dom(\rho_1) = dom(\rho_2) \supseteq vars(B) \tag{A.1}$$

$$rang(\rho_1) \cap vars(A) = \emptyset \tag{A.2}$$

$$rang(\rho_2) \cap vars(A) = \emptyset \tag{A.3}$$

Then

(a) A and $B\rho_1$ unify iff A and $B\rho_2$ unify.

(b) $mgu(A, B\rho_1)\restriction vars(A) \cong mgu(A, B\rho_2)\restriction vars(A)$.

(c) $\rho_1 \circ mgu(A, B\rho_1)\restriction dom(\rho_1) \cong \rho_2 \circ mgu(A, B\rho_2)\restriction dom(\rho_2)$.

PROOF: Let $vars(A) = \{X_1, \cdots, X_k\}$, $dom(\rho_1) = dom(\rho_2) = \{V_1, \cdots, V_l\}$, $\rho_1 = \{V_1/Y_1, \cdots, V_l/Y_l\}$ and $\rho_2 = \{V_1/Z_1, \cdots, V_l/Z_l\}$. Define

$$\rho_3 \stackrel{def}{=} \{Z_1/Y_1, \cdots, Z_l/Y_l\}$$

$$\rho_4 \stackrel{def}{=} \{Y_1/Z_1, \cdots, Y_l/Z_l\}$$

$$\mathcal{Y} \stackrel{def}{=} \{Y_1, \cdots, Y_l\}$$

$$\mathcal{Z} \stackrel{def}{=} \{Z_1, \cdots, Z_l\}$$

$$\mathcal{V} \stackrel{def}{=} \{V_1, \cdots, V_l\}$$

We have

$$\rho_1 = \rho_2 \circ \rho_3\restriction\mathcal{V} \tag{A.4}$$

$$\rho_2 = \rho_1 \circ \rho_4\restriction\mathcal{V} \tag{A.5}$$

Suppose that A and $B\rho_1$ unify with mgu θ_1. Let

$$\theta_1 = \{X_{i_1}/x_{i_1}, \cdots, X_{i_s}/x_{i_s}, Y_{j_1}/y_{j_1}, \cdots, Y_{j_t}/y_{j_t}\} \tag{A.6}$$

with $1 \leq i_1 \leq \cdots \leq i_s \leq k$ and $1 \leq j_1 \leq \cdots \leq j_t \leq l$. Define

$$y_h \stackrel{def}{=} \begin{cases} Y_h & If\ h \notin \{j_1, j_2, \cdots, j_t\} \\ y_h & If\ h \in \{j_1, j_2, \cdots, j_t\} \end{cases} \tag{A.7}$$

By equations A.6 and A.7, we have that

$$\rho_1 \circ \theta_1\restriction\mathcal{V} = \{V_1/y_1, \cdots, V_l/y_l\} \tag{A.8}$$

$A\rho_3\theta_1 = A\theta_1 = B\rho_1\theta_1 = B(\rho_2 \circ \rho_3\restriction\mathcal{V})\theta_1 = B\rho_2\rho_3\theta_1$ by equations A.1, A.3, A.4, A.6 and A.7. So, A and $B\rho_2$ unify with $\rho_3\theta_1$ being one of their unifiers if A and $B\rho_1$ unify with mgu θ_1.

Suppose that A and $B\rho_2$ unify with mgu θ_2. Let

$$\theta_2 = \{X_{u_1}/\overline{x}_{u_1}, \cdots, X_{u_p}/\overline{x}_{u_p}, Z_{v_1}/z_{v_1}, \cdots, Z_{v_q}/z_{v_q}\} \tag{A.9}$$

with $1 \leq u_1 \leq \cdots \leq u_p \leq k$ and $1 \leq v_1 \leq \cdots \leq v_q \leq l$. Define

$$z_h \stackrel{def}{=} \begin{cases} Z_h & If\ h \notin \{v_1, v_2, \cdots, v_q\} \\ z_h & If\ h \in \{v_1, v_2, \cdots, v_q\} \end{cases} \tag{A.10}$$

By equations A.9 and A.10, we have that

$$\rho_2 \circ \theta_2 |\!\!\upharpoonright \mathcal{V} = \{V_1/z_1, \cdots, V_l/z_l\} \tag{A.11}$$

$A\rho_4\theta_2 = A\theta_2 = B\rho_2\theta_2 = B(\rho_1 \circ \rho_4 |\!\!\upharpoonright \mathcal{V})\theta_2 = B\rho_1\rho_4\theta_2$ by equations A.1, A.2, A.5, A.9 and A.10. So, A and $B\rho_1$ unify with $\rho_4\theta_2$ being one of their unifiers if A and $B\rho_2$ unify with *mgu* θ_2. Therefore, (a) holds.

The following equation results from equation A.6 and the definition of ρ_3.

$$\begin{aligned} \rho_3\theta_1 = & \quad \{X_{i_1}/x_{i_1}, \cdots, X_{i_s}/x_{i_s}\} \\ & \cup \ \{Y_{j_o}/y_{j_o} \mid 1 \le o \le t \wedge Y_{j_o} \notin \mathcal{Z}\} \\ & \cup \ \{Z_1/y_1, \cdots, Z_l/y_l\} \end{aligned} \tag{A.12}$$

The following equation results from equation A.9 and the definition of ρ_4.

$$\begin{aligned} \rho_4\theta_2 = & \quad \{X_{u_1}/\overline{x}_{u_1}, \cdots, X_{u_p}/\overline{x}_{u_p}\} \\ & \cup \ \{Z_{v_o}/z_{v_o} \mid 1 \le o \le q \wedge Z_{v_o} \notin \mathcal{Y}\} \\ & \cup \ \{Y_1/z_1, \cdots, Y_l/z_l\} \end{aligned} \tag{A.13}$$

Since $\rho_4\theta_2$ is a unifier of A and $B\rho_1$, there is a substitution δ_1 such that $\rho_4\theta_2 = \theta_1\delta_1$. By equations A.6 and A.13, we have that

$$\left(\begin{array}{c} \{X_{u_1}/\overline{x}_{u_1}, \cdots, X_{u_p}/\overline{x}_{u_p}\} \\ \cup \\ \{Z_{v_o}/z_{v_o} \mid 1 \le o \le q \wedge Z_{v_o} \notin \mathcal{Y}\} \\ \cup \\ \{Y_1/z_1, \cdots, Y_l/z_l\} \end{array} \right) = \left(\begin{array}{c} \{X_{i_1}/x_{i_1}, \cdots, X_{i_s}/x_{i_s}\} \\ \cup \\ \{Y_{j_1}/y_{j_1}, \cdots, Y_{j_t}/y_{j_t}\} \end{array} \right) \delta_1 \tag{A.14}$$

Since $\rho_3\theta_1$ is a unifier of A and $B\rho_2$, there is a substitution δ_2 such that $\rho_3\theta_1 = \theta_2\delta_2$. By equations A.9 and A.12, we have that

$$\left(\begin{array}{c} \{X_{i_1}/x_{i_1}, \cdots, X_{i_s}/x_{i_s}\} \\ \cup \\ \{Y_{j_o}/y_{j_o} \mid 1 \le o \le t \wedge Y_{j_o} \notin \mathcal{Z}\} \\ \cup \\ \{Z_1/y_1, \cdots, Z_l/y_l\} \end{array} \right) = \left(\begin{array}{c} \{X_{u_1}/\overline{x}_{u_1}, \cdots, X_{u_p}/\overline{x}_{u_p}\} \\ \cup \\ \{Z_{v_1}/z_{v_1}, \cdots, Z_{v_q}/z_{v_q}\} \end{array} \right) \delta_2 \tag{A.15}$$

By equation A.14, $\{X_{i_1}, \cdots, X_{i_s}\} \subseteq \{X_{u_1}, \cdots, X_{u_p}\}$ and, by equation A.15, $\{X_{u_1}, \cdots, X_{u_p}\} \subseteq \{X_{i_1}, \cdots, X_{i_s}\}$. So, $\{X_{i_1}, \cdots, X_{i_s}\} = \{X_{u_1}, \cdots, X_{u_p}\}$ and thus we have that $s = p$ and $i_o = u_o$ for $1 \leq o \leq s$. By equations A.14 and A.15, we have that

$$x_{i_o} = \overline{x}_{i_o}\delta_2 \;\wedge\; \overline{x}_{i_o} = x_{i_o}\delta_1 \tag{A.16}$$

$x_{i_o} \cong \overline{x}_{i_o}$ for $1 \leq o \leq s$ from equation A.16. Therefore, (b) holds.

By equation A.14, we have that

$$Y_h/z_h \in \delta_1 \quad If\ h \notin \{j_1, \cdots, j_t\} \tag{A.17}$$

$$z_h = y_h\delta_1 \quad If\ h \in \{j_1, \cdots, j_t\} \tag{A.18}$$

$y_h = Y_h$ if $h \notin \{j_1, \cdots, j_t\}$ by equation A.7. So, $z_h = y_h\delta_1$ for $h \notin \{j_1, \cdots, j_t\}$ by equation A.17. This and equation A.18 imply that, for all $1 \leq h \leq l$,

$$z_h = y_h\delta_1 \tag{A.19}$$

By equation A.15, we have that

$$Z_h/y_h \in \delta_2 \quad If\ h \notin \{v_1, \cdots, v_q\} \tag{A.20}$$

$$y_h = z_h\delta_2 \quad If\ h \in \{v_1, \cdots, v_q\} \tag{A.21}$$

$z_h = Z_h$ if $h \notin \{v_1, \cdots, v_q\}$ by equation A.10. So, $y_h = z_h\delta_2$ for $h \notin \{v_1, \cdots, v_q\}$ by equation A.20. This and equation A.21 imply that for all $1 \leq h \leq l$

$$y_h = z_h\delta_2 \tag{A.22}$$

By equations A.8, A.11, A.19 and A.22, $\rho_1 \circ \theta_1 \upharpoonright \mathcal{V} \cong \rho_2 \circ \theta_2 \upharpoonright \mathcal{V}$. Therefore, (c) holds. ∎

Corollary A.1.3 Let A, B_1 and B_2 be atoms. If $B_1 \cong B_2$, $vars(A) \cap vars(B_1) = \emptyset$ and $vars(A) \cap vars(B_2) = \emptyset$ then A and B_1 unify iff A and B_2 unify, and $mgu(A, B_1)\upharpoonright vars(A) \cong mgu(A, B_2)\upharpoonright vars(A)$.

Proof: Because $B_1 \cong B_2$, there is a renaming substitution ρ_2 such that $B_2 = B_1\rho_2$ with $dom(\rho_2) = vars(B_1)$. Let ρ_1 be the renaming substitution on $vars(B_1)$ such that $X\rho_1 = X$ for each $X \in vars(B_1)$. By lemma A.1.2.(a) and lemma A.1.2.(b), A and B_1 unify iff A and B_2 unify and $mgu(A, B_1)\upharpoonright vars(A) = mgu(A, B_1\rho_1)\upharpoonright vars(A) \cong mgu(A, B_1\rho_2)\upharpoonright vars(A) = mgu(A, B_2)\upharpoonright vars(A)$. ∎

Corollary A.1.4 Let A and B be two atoms and ρ be a renaming substitution such that $dom(\rho) \supseteq vars(B)$. If $vars(A) \cap vars(B) = \emptyset$ and $vars(A) \cap vars(B\rho) = \emptyset$ then A and B unify iff A and $B\rho$ unify, and $mgu(A, B)\upharpoonright vars(B) \cong (\rho \circ mgu(A, B\rho))\upharpoonright vars(B)$.

PROOF: The proof results immediately from lemma A.1.2.(a) and (c) by letting $\rho_2 = \rho$ and ρ_1 be the renaming substitution on $vars(B)$ such that $X\rho_1 = X$ for each $X \in vars(B)$. ∎

Corollary A.1.5 Let A and B be two atoms, ρ_A and ρ_B be renaming substitutions. If $dom(\rho_A) \supseteq vars(A)$, $dom(\rho_B) \supseteq vars(B)$, $vars(A\rho_A) \cap vars(B) = \emptyset$ and $vars(B\rho_B) \cap vars(A) = \emptyset$ then $A\rho_A$ and B unify iff A and $B\rho_B$ unify, and $(\rho_A \circ mgu(A\rho_A, B))\upharpoonright dom(\rho_A) \cong mgu(A, B\rho_B)\upharpoonright vars(A)$.

PROOF: We prove the *if* part. The *only if* part is a dual case of the *if* part. Let ρ'_B be a renaming substitution such that $dom(\rho'_B) = dom(\rho_B)$, $vars(B\rho'_B) \cap vars(A) = \emptyset$ and $vars(A\rho_A) \cap vars(B\rho'_B) = \emptyset$.

Suppose that A and $B\rho_B$ unify. By lemma A.1.2.(a), A and $B\rho'_B$ unify, and $mgu(A, B\rho'_B)\upharpoonright vars(A) \cong mgu(A, B\rho_B)\upharpoonright vars(A)$ by lemma A.1.2.(b). By corollary A.1.4, $A\rho_A$ and $B\rho'_B$ unify, and $\rho_A \circ mgu(A\rho_A, B\rho'_B)\upharpoonright vars(A) \cong mgu(A, B\rho'_B)\upharpoonright vars(A)$. So, $\rho_A \circ mgu(A\rho_A, B\rho'_B)\upharpoonright vars(A) \cong mgu(A, B\rho_B)\upharpoonright vars(A)$. By corollary A.1.3, $A\rho_A$ and B unify, and $mgu(A\rho_A, B)\upharpoonright vars(A\rho_A) \cong mgu(A\rho_A, B\rho'_B)\upharpoonright vars(A\rho_A)$. Hence, $\rho_A \circ mgu(A\rho_A, B)\upharpoonright vars(A) \cong \rho_A \circ mgu(A\rho_A, B\rho'_B)\upharpoonright vars(A)$. Therefore, $\rho_A \circ mgu(A\rho_A, B)\upharpoonright vars(A) \cong mgu(A, B\rho_B)\upharpoonright vars(A)$. It now suffices to prove $\rho_A \circ mgu(A\rho_A, B)\upharpoonright dom(\rho_A) \cong \rho_A \circ mgu(A\rho_A, B)\upharpoonright vars(A)$. Let $\rho_A^1 = \rho_A\upharpoonright vars(A)$ and $\rho_A^2 = \rho_A\upharpoonright(dom(\rho_A) \setminus vars(A))$. We thus have that

$$\begin{aligned}
\rho_A &= \rho_A^1 \cup \rho_A^2 \\
\rho_A \circ mgu(A\rho_A, B)\upharpoonright dom(\rho_A) &= (\rho_A^1 \cup \rho_A^2) \circ mgu(A(\rho_A^1 \cup \rho_A^2), B)\upharpoonright dom(\rho_A) \\
&= \rho_A^1 \circ mgu(A\rho_A^1, B)\upharpoonright vars(A) \cup \rho_A^2 \\
\rho_A \circ mgu(A\rho_A, B)\upharpoonright vars(A) &= (\rho_A^1 \cup \rho_A^2) \circ mgu(A(\rho_A^1 \cup \rho_A^2), B)\upharpoonright vars(A) \\
&= \rho_A^1 \circ mgu(A\rho_A^1, B)\upharpoonright vars(A)
\end{aligned}$$

We also have that $rang(\rho_A^1 \circ mgu(A\rho_A^1, B)\upharpoonright vars(A)) \cap dom(\rho_A^2) = \emptyset$ and $dom(\rho_A^2) \cap vars(A) = \emptyset$. So,

$$\begin{aligned}
(\rho_A \circ mgu(A\rho_A, B)\upharpoonright vars(A)) \circ \rho_A^2 &= (\rho_A^1 \circ mgu(A\rho_A^1, B)\upharpoonright vars(A)) \circ \rho_A^2 \\
&= \rho_A^1 \circ mgu(A\rho_A^1, B)\upharpoonright vars(A) \cup \rho_A^2 \\
&= \rho_A \circ mgu(A\rho_A, B)\upharpoonright dom(\rho_A)
\end{aligned}$$

Therefore, $\rho_A \circ mgu(A\rho_A, B)\restriction dom(\rho_A) \cong \rho_A \circ mgu(A\rho_A, B)\restriction vars(A)$ since ρ_A^2 is a renaming substitution. ∎

Lemma A.1.6 Let θ_1 and θ_2 be two substitutions and $\mathcal{V}$ a set of variables. Then $\theta_1 \circ \theta_2\restriction\mathcal{V} = (\theta_1\restriction\mathcal{V}) \circ \theta_2\restriction\mathcal{V}$.

Proof: Let $(X/t) \in \theta_1 \circ \theta_2\restriction\mathcal{V}$. Then either $X \in dom(\theta_1)$ or $X \notin dom(\theta_1) \wedge X \in dom(\theta_2)$. If $X \in dom(\theta_1)$ then there is a t_1 such that $((X/t_1) \in \theta_1 \wedge t = t_1\theta_2)$. Since $X \in \mathcal{V}$, $(X/t_1) \in \theta_1\restriction\mathcal{V}$ and hence $X/(t_1\theta_2) = (X/t) \in (\theta_1\restriction\mathcal{V}) \circ \theta_2\restriction\mathcal{V}$. Otherwise, $X \in dom(\theta_2)$, $(X/t) \in \theta_2$ and $(X/t) \in (\theta_1\restriction\mathcal{V}) \circ \theta_2\restriction\mathcal{V}$.

Let $(X/t) \in (\theta_1\restriction\mathcal{V}) \circ \theta_2\restriction\mathcal{V}$. Then $X \in \mathcal{V}$. Either $X \in dom(\theta_1\restriction\mathcal{V})$ or $X \notin dom(\theta_1\restriction\mathcal{V}) \wedge X \in dom(\theta_2)$. If $X \in dom(\theta_1\restriction\mathcal{V})$ then there is a t_2 such that $((X/t_2) \in \theta_1\restriction\mathcal{V} \wedge t = t_2\theta_2)$. $(X/t_2) \in \theta_1$ and $(X/t) \in \theta_1 \circ \theta_2$. So, $(X/t) \in \theta_1 \circ \theta_2\restriction\mathcal{V}$. Otherwise, $(X/t) \in \theta_2$ and $X \notin dom(\theta_1) \cap \mathcal{V}$. So, $(X/t) \in \theta_1 \circ \theta_2\restriction\mathcal{V}$. ∎

Lemma A.1.7 Let E_1 and E_2 be two sets of equations, and θ_1 and θ_2 be two substitutions. If $\theta_1 = mgu(E_1)$ and $\theta_2 = mgu(E_2\theta_1)$ then $\theta_1 \circ \theta_2 = mgu(E_1 \cup E_2)$.

Proof: See [15] ∎

A.2 Proofs for Chapter 4

Proof of lemma 4.1.1 (p. 76): VSLDNF and SLDNF deal with negative literals in the same manner. Therefore, the cases where the leftmost literals are positive remain to be proved. The proof has two parts. The first part corresponds to procedure-entry and the second part to procedure-exit.

We first prove that if $\sigma_{(j,k)}\restriction\mathcal{V}_{i_j} \cong \rho_j \circ \tau_{(j,k)}\restriction\mathcal{V}_{i_j}$ then R2 (p. 74) is derived from R1 (p. 73) iff R2' (p. 75) is derived from R1' (p. 75) and $\sigma_{(j+1,1)}\restriction\mathcal{V}_{i_{j+1}} \cong \rho_{j+1} \circ \tau_{(j+1,1)}\restriction\mathcal{V}_{i_{j+1}}$.

Let $\sigma_{(j,k)}\restriction\mathcal{V}_{i_j} \cong \rho_j \circ \tau_{(j,k)}\restriction\mathcal{V}_{i_j}$. Then there is a renaming substitution δ such that

$$(\sigma_{(j,k)}\restriction\mathcal{V}_{i_j}) \circ \delta = \rho_j \circ \tau_{(j,k)}\restriction\mathcal{V}_{i_j} \tag{A.23}$$

By choosing the same clause $C_{i_{j+1}}$ to be resolved with both R1 and R1', we have that R2 is derived from R1 iff R2' is derived from R1' by corollary A.1.5 (p. 168). Suppose that R2 were derived from R1 and R2' from R1'.

$$
\begin{aligned}
B_{(i_j,k)}\rho_j\tau_{(j,k)} &= B_{(i_j,k)}(\rho_j \circ \tau_{(j,k)}) \\
&= B_{(i_j,k)}(\rho_j \circ \tau_{(j,k)}\restriction\mathcal{V}_{i_j}) \quad (\because vars(B_{(i_j,k)}) \subseteq \mathcal{V}_{i_j}) \\
&= B_{(i_j,k)}((\sigma_{(j,k)}\restriction\mathcal{V}_{i_j}) \circ \delta) \quad (\because \textit{equation A.23}) \\
&= B_{(i_j,k)}\sigma_{(j,k)}\delta \quad (\because vars(B_{(i_j,k)}) \subseteq \mathcal{V}_{i_j}) \qquad \text{(A.24)}
\end{aligned}
$$

$$
\begin{aligned}
\rho_{j+1} \circ \tau_{(j+1,1)}\restriction\mathcal{V}_{i_{j+1}} &= \rho_{j+1} \circ \tau_{(j,k)} \circ \eta\restriction\mathcal{V}_{i_{j+1}} \quad (\because \textit{equation 4.2}) \\
&= \rho_{j+1} \circ \eta\restriction\mathcal{V}_{i_{j+1}} \quad (\because \textit{equation 4.1}) \\
&= \rho_{j+1} \circ mgu(H_{i_{j+1}}\rho_{j+1}, B_{(i_j,k)}\rho_j\tau_{(j,k)})\restriction\mathcal{V}_{i_{j+1}} \quad (\because \textit{equation 4.3}) \\
&= \rho_{j+1} \circ mgu(H_{i_{j+1}}\rho_{j+1}, B_{(i_j,k)}\sigma_{(j,k)}\delta)\restriction\mathcal{V}_{i_{j+1}} \quad (\because \textit{equation A.24}) \\
&\qquad \text{(A.25)}
\end{aligned}
$$

Let δ^{-1} be the inverse of δ.

$$
\begin{aligned}
\sigma_{(j+1,1)}\restriction\mathcal{V}_{i_{j+1}} &= mgu(H_{i_{j+1}}, B_{(i_j,k)}\sigma_{(j,k)}\psi_{j+1})\restriction\mathcal{V}_{i_{j+1}} \quad (\because \textit{equation 4.6}) \\
&= mgu(H_{i_{j+1}}, B_{(i_j,k)}\sigma_{(j,k)}\delta\delta^{-1}\psi_{j+1})\restriction\mathcal{V}_{i_{j+1}} \quad (\because \delta\delta^{-1} \textit{ is identity}) \\
&= mgu(H_{i_{j+1}}, (B_{(i_j,k)}\sigma_{(j,k)}\delta)(\delta^{-1} \circ \psi_{j+1}))\restriction\mathcal{V}_{i_{j+1}} \\
&= mgu(H_{i_{j+1}}, (B_{(i_j,k)}\sigma_{(j,k)}\delta)(\delta^{-1} \circ \psi_{j+1}))\restriction vars(H_{i_{j+1}}) \qquad \text{(A.26)}
\end{aligned}
$$

$\sigma_{(j+1,1)}\restriction\mathcal{V}_{i_{j+1}} \cong \rho_{j+1} \circ \tau_{(j+1,1)}\restriction\mathcal{V}_{i_{j+1}}$ by corollary A.1.5 and equations A.25 and A.26. This completes the first part of the proof.

We now prove that if $\sigma_{(j+1,m[i_{j+1}]+1)}\restriction\mathcal{V}_{i_{j+1}} \cong \rho_{j+1} \circ \tau_{j,k+1}\restriction\mathcal{V}_{i_{j+1}}$ then $\sigma_{(j,k+1)}\restriction\mathcal{V}_{i_j} \cong \rho_j \circ \tau_{(j,k+1)}\restriction\mathcal{V}_{i_j}$. Let $\tilde{\delta}$ be a renaming substitution such that

$$
\sigma_{(j+1,m[i_{j+1}]+1)}\restriction\mathcal{V}_{i_{j+1}} = (\rho_{j+1} \circ \tau_{j,k+1}\restriction\mathcal{V}_{i_{j+1}}) \circ \tilde{\delta}
$$

and $\tilde{\delta}^{-1}$ be the inverse of $\tilde{\delta}$. $\sigma_{(j+1,m[i_{j+1}]+1)} = \rho_{j+1} \circ \tau_{j,k+1} \circ \tilde{\delta}\restriction\mathcal{V}_{i_{j+1}}$. Therefore,

$$
\begin{aligned}
H_{i_{j+1}}\sigma_{(j+1,m[i_{j+1}]+1)}\phi_{j+1} &= H_{i_{j+1}}\rho_{j+1}\tau_{(j,k+1)}\tilde{\delta}\phi_{j+1} \\
&= H_{i_{j+1}}\rho_{j+1}\tau_{(j,k)}\eta\theta\tilde{\delta}\phi_{j+1} \quad (\because \textit{equation 4.4}) \\
&= H_{i_{j+1}}\rho_{j+1}\eta\theta\tilde{\delta}\phi_{j+1} \quad (\because vars(C_{i_{j+1}}\rho_{j+1}) \cap vars(C_{i_j}\rho_j) = \emptyset) \\
&= B_{(i_j,k)}\rho_j\tau_{(j,k)}\eta\theta\tilde{\delta}\phi_{j+1} \quad (\because \textit{equation 4.3}) \\
&\qquad \text{(A.27)}
\end{aligned}
$$

By equation A.23,

$$
\sigma_{(j,k)})\restriction\mathcal{V}_{i_j} = \rho_j \circ \tau_{(j,k)} \circ \delta^{-1}\restriction\mathcal{V}_{i_j} \qquad \text{(A.28)}
$$

So,

$$
\begin{aligned}
B_{(i_j,k)}\sigma_{(j,k)} &= B_{(i_j,k)}(\rho_j \circ \tau_{j,k} \circ \delta^{-1}\restriction\mathcal{V}_{i_j}) \quad (\because \textit{equation A.28}) \\
&= B_{(i_j,k)}\rho_j\tau_{j,k}\delta^{-1} \quad (\because vars(B_{(i_j,k)}) \subseteq \mathcal{V}_{i_j}) \qquad \text{(A.29)}
\end{aligned}
$$

Substituting equations A.27-A.29 into equation 4.7 and letting $A = B_{(i_j,k)}\rho_j\tau_{j,k}$, we have that

$$\begin{array}{rcll}
\sigma_{(j,k+1)}\restriction\mathcal{V}_{i_j} & = & \rho_j \circ \tau_{j,k} \circ \delta^{-1} \circ mgu(A\delta^{-1}, A\eta\theta\tilde{\delta}\phi_{j+1})\restriction\mathcal{V}_{i_j} & \\
& = & \rho_j \circ \tau_{j,k} \circ (\delta^{-1} \circ mgu(A\delta^{-1}, A\eta\theta\tilde{\delta}\phi_{j+1})\restriction vars(A))\restriction\mathcal{V}_{i_j} & (\because\ lemma\ A.1.6) \\
& \cong & \rho_j \circ \tau_{j,k} \circ (mgu(A, A\eta\theta\tilde{\delta}\phi_{j+1})\restriction vars(A))\restriction\mathcal{V}_{i_j} & (\because\ lemma\ A.1.4) \\
& = & \rho_j \circ \tau_{j,k} \circ mgu(A, A\eta\theta\tilde{\delta}\phi_{j+1})\restriction\mathcal{V}_{i_j} & (\because\ lemma\ A.1.6) \\
& = & \rho_j \circ \tau_{j,k} \circ \eta \circ \theta \circ \tilde{\delta} \circ \phi_{j+1}\restriction\mathcal{V}_{i_j} & \\
& \cong & \rho_j \circ \tau_{j,k} \circ \eta \circ \theta\restriction\mathcal{V}_{i_j} & (\because\ \tilde{\delta}, \phi_{j+1}\ are\ renaming\ substitutions) \\
& = & \rho_j \circ \tau_{(j,k+1)}\restriction\mathcal{V}_{i_j} & (\because\ equation\ 4.4)
\end{array}$$

∎

Proof of lemma 4.2.3 (p. 84): It is sufficient to prove that $F_{\mathcal{P},\mathcal{G}} \uparrow k \subseteq \gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k)$ for any ordinal k. The proof is done by transfinite induction.

Basis. $F_{\mathcal{P},\mathcal{G}} \uparrow 0 = \emptyset \subseteq \{\$\} = \gamma^\sharp(\perp^\sharp) = \gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow 0)$.

Induction. Let $F_{\mathcal{P},\mathcal{G}} \uparrow k' \subseteq \gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k')$ for any $k' < k$. If k is a limit ordinal then $F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k = \sqcup^\sharp\{F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k' \mid k' < k\}$. Therefore, $\gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k) \supseteq \gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k')$ for any $k' < k$ by equation 4.15. By the induction hypothesis, $\gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k) \supseteq F_{\mathcal{P},\mathcal{G}} \uparrow k'$ for any $k' < k$. So, $F_{\mathcal{P},\mathcal{G}} \uparrow k \subseteq \gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k)$.

Let k not be a limit ordinal. Let $\| p_1 \leftarrow\!\bullet q_1, \theta_1 \| \cdots \| p_n \leftarrow\!\bullet q_n, \theta_n \| \cdot \$ \in F_{\mathcal{P},\mathcal{G}} \uparrow k$. By equation 4.15, it is sufficient to prove that $p_i \leftarrow\!\bullet q_i \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \wedge \theta_i \in [F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k]_{p_i \leftarrow\bullet q_i}$ for any $1 \leq i \leq n$.

There is a $0 \leq \jmath \leq 3$, such that $\| p_1 \leftarrow\!\bullet q_1, \theta_1 \| \cdots \| p_n \leftarrow\!\bullet q_n, \theta_n \| \cdot \$ \in F^\jmath_{\mathcal{P},\mathcal{G}}(F_{\mathcal{P},\mathcal{G}} \uparrow (k-1))$ by equation 4.10.

Let $\jmath = 0$. By equation 4.11, $n = 1$ and $p_1 \leftarrow\!\bullet q_1 \in \mathcal{E}^0_{\mathcal{P},\mathcal{G}} \wedge \theta_1 \in \Theta_{p[1]}$. So, by equation 4.17, $\| p_1 \leftarrow\!\bullet q_1, \theta_1 \| \cdots \| p_n \leftarrow\!\bullet q_n, \theta_n \| \cdot \$ \in \gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k)$.

Let $\jmath = 1$. By equation 4.12, $p_1 \leftarrow\!\bullet q_1 \in \mathcal{E}^1_{\mathcal{P},\mathcal{G}}$, $p_2 = q_1$ and $\| p_2 \leftarrow\!\bullet q_2, \theta_2 \| \cdots \| p_n \leftarrow\!\bullet q_n, \theta_n \| \cdot \$ \in F_{\mathcal{P},\mathcal{G}} \uparrow (k-1)$ and $\theta_1 = unify(B_{q_1}, \sigma, H_{p_1[1]}, \epsilon) \neq fail$. $\| p_2 \leftarrow\!\bullet q_2, \theta_2 \| \cdots \| p_n \leftarrow\!\bullet q_n, \theta_n \| \cdot \$ \in \gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow (k-1))$ by the induction hypothesis. $\| p_1 \leftarrow\!\bullet q_1, \theta_1 \| \cdots \| p_n \leftarrow\!\bullet q_n, \theta_n \| \cdot \$ \in \gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k)$ by equation 4.18 and the monotonicity of $F^\sharp_{\mathcal{P},\mathcal{G}}$.

Let $\jmath = 2$. By equation 4.13, $p_1 \leftarrow\!\bullet q_1 \in \mathcal{E}^2_{\mathcal{P},\mathcal{G}}$ and there are two stack items $\| q_1 \leftarrow\!\bullet u, \sigma \|$ and $\| p_{\bar{1}} \leftarrow\!\bullet v, \eta \|$ such that

$$\| q_1 \leftarrow\!\bullet u, \sigma \| \cdot \| p_{\bar{1}} \leftarrow\!\bullet v, \eta \| \cdot \| p_2 \leftarrow\!\bullet q_2, \theta_2 \| \cdots \| p_n \leftarrow\!\bullet q_n, \theta_n \| \cdot \$ \in F_{\mathcal{P},\mathcal{G}} \uparrow (k-1)$$

$$\theta_1 = unify(H_{q_1[1]}, \sigma, B_{p_{\bar{1}}}, \eta) \neq fail$$

$\| p_2 \leftarrow\!\!\bullet q_2, \theta_2 \| \cdots \| p_n \leftarrow\!\!\bullet q_n, \theta_n \| \cdot \$ \in \gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow (k-1))$ by the induction hypothesis. Therefore, $\| p_1 \leftarrow\!\!\bullet q_1, \theta_1 \| \cdots \| p_n \leftarrow\!\!\bullet q_n, \theta_n \| \cdot \$ \in \gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k)$ by equation 4.19 and the monotonicity of $F^\sharp_{\mathcal{P},\mathcal{G}}$.

Let $\jmath = 3$. By equation 4.14, $p_1 \leftarrow\!\!\bullet q_1 \in \mathcal{E}^3_{\mathcal{P},\mathcal{G}}$, $p_2 = q_1$, and $\| p_2 \leftarrow\!\!\bullet q_2, \theta_2 \| \cdots \| p_n \leftarrow\!\!\bullet q_n, \theta_n \| \cdot \$ \in F_{\mathcal{P},\mathcal{G}} \uparrow (k-1)$. By the induction hypothesis, $\| p_2 \leftarrow\!\!\bullet q_2, \theta_2 \| \cdots \| p_n \leftarrow\!\!\bullet q_n, \theta_n \| \cdot \$ \in \gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow (k-1))$. Therefore, $\| p_1 \leftarrow\!\!\bullet q_1, \theta_1 \| \cdots \| p_n \leftarrow\!\!\bullet q_n, \theta_n \| \cdot \$ \in \gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k)$ by equation 4.20 and the monotonicity of $F^\sharp_{\mathcal{P},\mathcal{G}}$.

Therefore, $F_{\mathcal{P},\mathcal{G}} \uparrow k \subseteq \gamma^\sharp(F^\sharp_{\mathcal{P},\mathcal{G}} \uparrow k)$ for any ordinal k. ∎

Proof of theorem 4.3.3 (p. 86): (C4) implies that $F^\flat_{\mathcal{P},\mathcal{G}}$ is monotonic and therefore $lfp F^\flat_{\mathcal{P},\mathcal{G}}$ exists. By theorem 2.1.9, it suffices to prove that, for any $X^\flat \in \mathbf{D}^\flat$, $F^\sharp_{\mathcal{P},\mathcal{G}} \cdot \gamma^\flat(X^\flat) \sqsubseteq^\sharp \gamma^\flat \cdot F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)$.

Let $\sigma \in [F^\sharp_{\mathcal{P},\mathcal{G}} \cdot \gamma^\flat(X^\flat)]_{p \leftarrow\bullet q}$. We need to prove $\sigma \in [\gamma^\flat \cdot F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{p \leftarrow\bullet q}$. Let $p \leftarrow\!\!\bullet q \in \mathcal{E}^\jmath_{\mathcal{P},\mathcal{G}}$ for some $0 \le \jmath \le 3$.

Let $\jmath = 0$. By equation 4.17, $\sigma \in \gamma[\![\mathcal{V}_{p[1]}]\!](\theta^\flat_{p[1]})$. By equation 4.22, $\sigma \in \gamma[\![\mathcal{V}_{p[1]}]\!]([F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{p \leftarrow\bullet q}) = [\gamma^\flat \cdot F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{p \leftarrow\bullet q}$.

Let $\jmath = 1$. By equation 4.18, there is a $u \in \mathcal{N}_{\mathcal{P},\mathcal{G}}$ such that $q \leftarrow\!\!\bullet u \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$ and $\sigma \in unify^\sharp(B_q, [\gamma^\flat(X^\flat)]_{q \leftarrow\bullet u}, H_{p[1]}, \{\epsilon\})$. By equations 4.18 and 4.21, the monotonicity of function $unify^\sharp$ in its fourth argument, and C3,

$$\begin{array}{rcl} \sigma & \in & unify^\sharp(B_q, \gamma[\![\mathcal{V}_{q[1]}]\!](X^\flat_{q \leftarrow\bullet u}), H_{p[1]}, \gamma[\![\mathcal{V}_{p[1]}]\!](\epsilon^\flat[\![\mathcal{V}_{p[1]}]\!])) \\ & \subseteq & \gamma[\![\mathcal{V}_{p[1]}]\!] \cdot unify^\flat(B_q, X^\flat_{q \leftarrow\bullet u}, H_{p[1]}, \epsilon^\flat[\![\mathcal{V}_{p[1]}]\!]) \end{array}$$

So, by equations 4.21 and 4.23, and the monotonicity of $\gamma[\![\mathcal{V}_{p[1]}]\!]$

$$\sigma \in \gamma[\![\mathcal{V}_{p[1]}]\!]([F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{p \leftarrow\bullet q}) \subseteq \gamma[\![\mathcal{V}_{p[1]}]\!]([F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{p \leftarrow\bullet q}) = [\gamma^\flat \cdot F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{p \leftarrow\bullet q}$$

Let $\jmath = 2$. There are $u, v \in \mathcal{N}_{\mathcal{P},\mathcal{G}}$ such that $p^- \leftarrow\!\!\bullet v, q \leftarrow\!\!\bullet u \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$ and

$$\sigma \in unify^\sharp(H_{q[1]}, [\gamma^\flat(X^\flat)]_{q \leftarrow\bullet u}, B_{p^-}, [\gamma^\flat(X^\flat)]_{p^- \leftarrow\bullet v})$$

by equation 4.19. By equations 4.21 and 4.24,

$$\begin{array}{rcl} \sigma & \in & unify^\sharp(H_{q[1]}, \gamma[\![\mathcal{V}_{q[1]}]\!](X^\flat_{q \leftarrow\bullet u}), B_{p^-}, \gamma[\![\mathcal{V}_{p^-[1]}]\!](X^\flat_{p^- \leftarrow\bullet v})) \\ & \subseteq & \gamma[\![\mathcal{V}_{p^-[1]}]\!] \cdot unify^\flat(H_{q[1]}, X^\flat_{q \leftarrow\bullet u}, B_{p^-}, X^\flat_{p^- \leftarrow\bullet v}) \\ & = & \gamma[\![\mathcal{V}_{p[1]}]\!] \cdot unify^\flat(H_{q[1]}, X^\flat_{q \leftarrow\bullet u}, B_{p^-}, X^\flat_{p^- \leftarrow\bullet v}) \\ & \subseteq & \gamma[\![\mathcal{V}_{p[1]}]\!]([F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{p \leftarrow\bullet q}) \\ & = & [\gamma^\flat \cdot F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{p \leftarrow\bullet q} \end{array}$$

Let $\jmath = 3$. By equation 4.20, there is a $u \in \mathcal{N}_{\mathcal{P},\mathcal{G}}$ such that $\sigma \in [\gamma^\flat(X^\flat)]_{q\leftarrow\bullet u}$. By equations 4.21 and 4.25,

$$\begin{array}{rcl} \sigma & \in & [\gamma^\flat(X^\flat)]_{q\leftarrow\bullet u} \\ & = & \gamma[\![\mathcal{V}_{q[1]}]\!](X^\flat_{q\leftarrow\bullet u}) \\ & \subseteq & \gamma[\![\mathcal{V}_{q[1]}]\!]([F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{p\leftarrow\bullet q}) \\ & = & \gamma[\![\mathcal{V}_{p[1]}]\!]([F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{p\leftarrow\bullet q}) \\ & = & [\gamma^\flat \cdot F^\flat_{\mathcal{P},\mathcal{G}}(X^\flat)]_{p\leftarrow\bullet q} \end{array}$$

∎

Proof of theorem 4.4.1 (p. 88): It suffices to prove that, for any $p\leftarrow\bullet q \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$ and any $X^\diamond \in \mathbf{D}^\diamond$, $[F^\flat_{\mathcal{P},\mathcal{G}} \cdot \gamma^\diamond(X^\diamond)]_{p\leftarrow\bullet q} = [\gamma^\diamond \cdot F^\diamond_{\mathcal{P},\mathcal{G}}(X^\diamond)]_{p\leftarrow\bullet q}$. By equation 4.31,

$$\forall p\leftarrow\bullet q \in \mathcal{E}_{\mathcal{P},\mathcal{G}}.([\gamma^\diamond(X^\diamond)]_{p\leftarrow\bullet q} = X^\diamond_p) \tag{A.30}$$

Let $p\leftarrow\bullet q \in \mathcal{E}^0_{\mathcal{P},\mathcal{G}}$. $[F^\flat_{\mathcal{P},\mathcal{G}} \cdot \gamma^\diamond(X^\diamond)]_{p\leftarrow\bullet q} = \theta^\flat_{p[1]} = [\gamma^\diamond \cdot F^\diamond_{\mathcal{P},\mathcal{G}}(X^\diamond)]_{p\leftarrow\bullet q}$ by equations 4.22, 4.32 and A.30.

Let $p\leftarrow\bullet q \in \mathcal{E}^1_{\mathcal{P},\mathcal{G}}$. By equations 4.23, 4.33 and A.30,

$$[F^\flat_{\mathcal{P},\mathcal{G}} \cdot \gamma^\diamond(X^\diamond)]_{p\leftarrow\bullet q} = \mathit{unify}^\flat(B_q, X^\diamond_q, H_{p[1]}, \epsilon^\flat[\![\mathcal{V}_{p[1]}]\!]) = [\gamma^\diamond \cdot F^\diamond_{\mathcal{P},\mathcal{G}}(X^\diamond)]_{p\leftarrow\bullet q}$$

Let $p\leftarrow\bullet q \in \mathcal{E}^2_{\mathcal{P},\mathcal{G}}$. By equation 4.24, 4.34 and A.30,

$$[F^\flat_{\mathcal{P},\mathcal{G}} \cdot \gamma^\diamond(X^\diamond)]_{p\leftarrow\bullet q} = \mathit{unify}^\flat(H_{q[1]}, X^\diamond_q, B_{p\text{-}}, X^\diamond_{p\text{-}}) = [\gamma^\diamond \cdot F^\diamond_{\mathcal{P},\mathcal{G}}(X^\diamond)]_{p\leftarrow\bullet q}$$

Let $p\leftarrow\bullet q \in \mathcal{E}^3_{\mathcal{P},\mathcal{G}}$. By equations 4.25, 4.35 and A.30,

$$[F^\flat_{\mathcal{P},\mathcal{G}} \cdot \gamma^\diamond(X^\diamond)]_{p\leftarrow\bullet p\text{-}} = X^\diamond_{p\text{-}} = [\gamma^\diamond \cdot F^\diamond_{\mathcal{P},\mathcal{G}}(X^\diamond)]_{p\leftarrow\bullet p\text{-}}$$

Therefore, $[F^\flat_{\mathcal{P},\mathcal{G}} \cdot \gamma^\diamond(X^\diamond)]_{p\leftarrow\bullet q} = [\gamma^\diamond F^\diamond_{\mathcal{P},\mathcal{G}}(X^\diamond)]_{p\leftarrow\bullet q}$. ∎

Proof of theorem 4.5.3 (p. 91): $\theta^\flat\Psi \cup \sigma^\flat \in Sub^\flat[\![\mathcal{U}\Psi \cup \mathcal{V}]\!]$. Let $\zeta \in \gamma[\![\mathcal{U}\Psi \cup \mathcal{V}]\!](\theta^\flat\Psi \cup \sigma^\flat)$ and $Y \in \mathit{downwards}(E_0, \theta^\flat\Psi \cup \sigma^\flat)$. Then either $Y \in \theta^\flat\Psi \cup \sigma^\flat$ or there is an X and t such that $X \in \theta^\flat\Psi \cup \sigma^\flat$, $(X = t) \in E_0$ and $Y \in \mathit{vars}(t)$. So, $Y(\zeta \circ \mathit{mgu}(E_0\zeta))$ is ground if $\mathit{mgu}(E_0\zeta) \neq \mathit{fail}$. It is true that if every variable in a term is ground under a substitution then that term is ground under the same substitution. Therefore, if $Z \in \mathit{upwards}(E_0, \mathit{downwards}(E_0, \theta^\flat\Psi \cup \sigma^\flat))$ then Z is ground under $\zeta \circ \mathit{mgu}(E_0\zeta)\restriction(\mathcal{U}\Psi \cup \mathcal{V})$. This and lemmas A.1.1 and A.1.7 complete the proof of C4. ∎

A.3 Proofs for Chapter 5

PROOF OF THEOREM 5.3.2 (p. 105): Let Ψ be also a renaming substitution such that $(vars(A\Psi)\cup vars(\Psi(\theta)))\cap(vars(B)\cup vars(\sigma)) = \emptyset$. Let $\mathcal{W} = \mathcal{U}\Psi\cup\mathcal{V}$, $\eta = \sigma\cup\Psi(\theta)$ and $E_0 = eq(mgu(\{A\Psi = B\}))$. $\zeta \in \gamma[\![\mathcal{W}]\!](\zeta^\flat) \leftrightarrow \zeta\upharpoonright\mathcal{W} \in \gamma[\![\mathcal{W}]\!](\zeta^\flat)$ for any $\zeta \in Sub$ and $\zeta^\flat \in Sub^\flat[\![\mathcal{W}]\!]$. Therefore, by lemma A.1.2,

$$\begin{array}{lrcl} & unify(A,\theta,B,\sigma) & \in & \gamma[\![\mathcal{V}]\!](unify^\flat(A,\theta^\flat,B,\sigma^\flat)) \\ \leftrightarrow & \sigma\circ mgu(\{(A\Psi)\Psi(\theta) = B\sigma\}) & \in & \gamma[\![\mathcal{V}]\!](unify^\flat(A,\theta^\flat,B,\sigma^\flat)) \\ \leftrightarrow & (\sigma\cup\Psi(\theta))\circ mgu(\{(A\Psi)(\sigma\cup\Psi(\theta)) = B(\sigma\cup\Psi(\theta))\}) & \in & \gamma[\![\mathcal{V}]\!](unify^\flat(A,\theta^\flat,B,\sigma^\flat)) \\ \leftrightarrow & \eta\circ mgu(E_0\eta) & \in & \gamma[\![\mathcal{V}]\!](unify^\flat(A,\theta^\flat,B,\sigma^\flat)) \end{array} \tag{A.31}$$

By equation 5.2, $\eta \in \gamma[\![\mathcal{W}]\!](\theta^\flat\Psi \uplus \sigma^\flat)$. $\eta\circ mgu(E_0\eta) \in \gamma[\![\mathcal{W}]\!](solve(E_0, \theta^\flat\Psi \uplus \sigma^\flat))$ by lemma 5.3.3. So, by equations 5.1, 5.5 and A.31, $unify(A,\theta,B,\sigma) \in \gamma[\![\mathcal{V}]\!](unify^\flat(A,\theta^\flat,B,\sigma^\flat))$. ∎

PROOF OF LEMMA 5.3.3 (p. 105): The proof is done by induction on the size of E. If $E = \emptyset$ then $\sigma\circ mgu(E\sigma) = \sigma\circ\emptyset = \sigma \in \gamma[\![\mathcal{W}]\!](\sigma^\flat) = \gamma[\![\mathcal{W}]\!](solve(E,\sigma^\flat))$ by equation 5.3.

Suppose $E \neq \emptyset$. Let $E = \{X = t\}\cup E'$ and $\eta = mgu(X\sigma, t\sigma)$. $\sigma\circ\eta \in \gamma[\![\mathcal{W}]\!](soln(X = t,\sigma^\flat))$ by lemma 5.3.4. Let $\zeta = mgu(E'(\sigma\circ\eta))$. By the induction hypothesis and equation 5.3,

$$\sigma\circ\eta\circ\zeta \in \gamma[\![\mathcal{W}]\!](solve(E', soln(X = t,\sigma^\flat))) = \gamma[\![\mathcal{W}]\!](solve(E,\sigma^\flat))$$

$\sigma\circ\eta\circ\zeta = \sigma\circ(\eta\circ\zeta) = \sigma\circ mgu(E\sigma)$. So, $\sigma\circ mgu(E\sigma) \in \gamma[\![\mathcal{W}]\!](solve(E,\sigma^\flat))$. ∎

PROOF OF LEMMA 5.3.4 (p. 105): $soln(X = t, <\mathcal{I}_i,\mathcal{S}_i,\mathcal{A}_i>) = <\mathcal{I}_o,\mathcal{S}_o,\mathcal{A}_o>$, by equation 5.6, if

$$\begin{array}{rcl} unieq(X = t, <\mathcal{I}_i,\mathcal{S}_i,\mathcal{A}_i>) & = & <<\mathcal{I}',\mathcal{S}',\mathcal{A}'>,\mathcal{Z}'> \\ compose(<\mathcal{I}_i,\mathcal{S}_i,\mathcal{A}_i>, <<\mathcal{I}',\mathcal{S}',\mathcal{A}'>,\mathcal{Z}>, \{X\}\cup vars(t)) & = & <\mathcal{I}'',\mathcal{S}'',\mathcal{A}''> \\ reduce(<\mathcal{I}'',\mathcal{S}'',\mathcal{A}''>) & = & <\mathcal{I}_o,\mathcal{S}_o,\mathcal{A}_o> \end{array}$$

$\gamma[\![\mathcal{W}]\!](<\mathcal{I}'',\mathcal{S}'',\mathcal{A}''>) = \gamma[\![\mathcal{W}]\!](<\mathcal{I}_o,\mathcal{S}_o,\mathcal{A}_o>)$ by equation 5.12 and C5', C6', So, it is sufficient to prove that, for any $U, V \in \mathcal{W}$,

(a) if $U\eta \neq V\eta$ then $(U,V) \notin \mathcal{A}''$ and

(b) if $U \neq V \wedge vars(U\eta)\cap vars(V\eta) \neq \emptyset$ then $(U,V) \in \mathcal{S}''$ and if $\chi(U\eta) = 2$ then $(U,U) \in \mathcal{S}''$ and

(c) $U\eta \in \check{\gamma}(\mathcal{I}''(U))$.

(a) If $(U,V) \in \mathcal{A}''$ then either (a.1) $(U,V) \in \mathcal{A}_i$ or (a.2) $t \in \mathcal{W}$ and $\exists W, Z \in \{X, t\}.((U,W) \in \mathcal{A}_i \wedge (Z,V) \in \mathcal{A}_i)$. In case (a.1), $U\sigma = V\sigma$ and hence $U\eta = V\eta$. In case (a.2), $U\sigma = W\sigma$ and $V\sigma = Z\sigma$, so, $U\eta = V\eta$ because $\eta = \sigma \circ mgu(\{X\sigma = t\sigma\})$ and $W, Z \in \{X, t\}$.

(b) results from Lemma 2.2.(3)-(4) in [15], Lemma 5(2) in [68] and Lemma 4.4.(2) in [15]. The part of our algorithm devoted to pair sharing analysis is similar to that in [15] except that the part of our algorithm for pair sharing analysis makes use of Lemma 5(2) in [68] instead of Lemma 2.2.(5) in [15] which is erroneous [1].

(c) By C8' and equation 5.9, $U\eta \in \check{\gamma}(\mathcal{I}''(U))$ for any variable $U \in \{X\} \cup vars(t)$. By lemmas 5.2.1 and 5.2.2, $\mathcal{Z}'$ is a subset of $vars(t)$ such that, for any variable $U \in \mathcal{W}\backslash vars(t)$, if U does not share with any variable in $\mathcal{Z}'$ under σ then $U\eta \in \check{\gamma}(\mathcal{I}''(U))$ by equation 5.9. $U\eta \in \check{\gamma}(\mathcal{I}''(U))$ for any variable $U \in \mathcal{W}$ by equation 5.9, C4'-C7' and lemmas 5.2.1 and 5.2.2. ∎

Proof of lemma 5.4.6 (p. 107): (1) Since $U = t$, $mo(U\sigma\eta) \in \{mo(X\sigma)\}\nabla\{mo(t\sigma)\}$ by lemma 5.4.4. (2) Since $mo(X\sigma) = ⓖ \vee mo(t\sigma) = ⓖ$, $mo(t\sigma\eta) = ⓖ$. Hence, $mo(U\sigma\eta) = ⓖ$. (3) $mo(U\sigma)$ may assume one of three values in $\triangle$. If $mo(U\sigma) = ⓕ$, we have that $\triangle\nabla\{mo(U\sigma)\} = \triangle$ according to figure 5.1 and hence $mo(t\sigma\eta) \in \triangle\nabla\{mo(U\sigma)\}$. If $mo(U\sigma) = ⓞ$ then $mo(U\sigma\eta) \in \{ⓖ, ⓞ\} = \triangle\nabla\{mo(U\sigma)\}$ according to figure 5.1. Otherwise, $mo(U\sigma) = ⓖ$ and $mo(U\sigma\eta) = ⓖ \in \{ⓖ\} = \triangle\nabla\{mo(U\sigma)\}$ according to figure 5.1. ∎

Proof of lemma 5.5.10 (p. 114): Equation 5.15 can be written as $\check{\gamma}^t(M) = \bigcup_{k\geq 0}\{\check{\gamma}^t_k(M)\}$ where

$$\check{\gamma}^t_k(M) = \begin{cases} \mathcal{TERMS} & if\ M = any \\ \emptyset & if\ M = empty \vee k = 0 \\ \left\{ f(t_1, \cdots, t_n) \middle| \begin{array}{c} f(X_1,\cdots,X_n){:}c(\beta_1,\cdots,\beta_m) \leftarrow X_1{:}T_1,\cdots,X_n{:}T_n \\ \kappa = \{\beta_j/M_j | 1 \leq j \leq m\} \\ \forall 1 \leq i \leq n.(t_i \in \check{\gamma}^t_{k-1}(T_i\kappa)) \end{array} \right\} & if\ M = c(M_1, \cdots, M_m) \end{cases}$$

It suffices to prove that, for any $k \geq 0$,

$$M \check{\sqsubseteq}^t M' \rightarrow \check{\gamma}^t_k(M) \subseteq \check{\gamma}^t_k(M') \tag{A.32}$$

If either $M = empty$ or $M' = any$ then equation A.32 holds for any $k \geq 0$. The proof for the case where $M \neq empty$ and $M' \neq any$ is done by induction on k.

[1] We owe our gratitude to Dr. Andy King who pointed out that to us.

Basis. $\check{\gamma}_0^t(M) = \check{\gamma}_0^t(M') = \emptyset$. So, equation A.32 holds for $k = 0$.

Induction. If $M \check{\sqsubseteq}^t M'$ then, by definition 5.5.1, there are a type constructor c/m and mono-types $M_1, \cdots, M_m$, $M_1', \cdots, M_m'$ such that $M = c(M_1, \cdots, M_m)$, $M' = c(M_1', \cdots, M_m')$ and $M_j \check{\sqsubseteq}^t M_j'$ for $1 \leq j \leq m$. Let $f(t_1, \cdots, t_n) \in \check{\gamma}_k^t(M)$ be an arbitrary term. It remains to prove that $f(t_1, \cdots, t_n) \in \check{\gamma}_k^t(M')$. Let

$$f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$$

Since $f(t_1, \cdots, t_n) \in \check{\gamma}_k^t(M)$, $t_i \in \check{\gamma}_{k-1}^t(T_i\{\beta_j/M_j \mid 1 \leq j \leq m\})$. Since $M_j \check{\sqsubseteq}^t M_j'$ for $1 \leq j \leq m$, $T_i\{\beta_j/M_j \mid 1 \leq j \leq m\} \check{\sqsubseteq}^t T_i\{\beta_j/M_j' \mid 1 \leq j \leq m\}$. Thus, according to the induction hypothesis, $t_i \in \check{\gamma}_{k-1}^t(T_i\{\beta_j/M_j' \mid 1 \leq j \leq m\})$. Therefore, $f(t_1, \cdots, t_n) \in \check{\gamma}_k^t(M')$. ∎

PROOF OF LEMMA 5.5.11 (p. 114): The proof is by induction on the height $h(t)$ of t.

Basis. $h(t) = 0$. t must be a constant for otherwise $M = any$ by equation 5.15.

Induction. $h(t) > 0$. There are a function symbol $f/n \in \Sigma$ and terms $t_1, \cdots, t_n$ such that $t = f(t_1, \cdots, t_n)$. Let

$$f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$$

By equation 5.15, there are $M_1, \cdots, M_m$ such that $M = c(M_1, \cdots, M_m)$ and $t_i \in \check{\gamma}^t(T_i\{\beta_j/M_j \mid 1 \leq j \leq m\})$. any does not occur in $T_i\{\beta_j/M_j \mid 1 \leq j \leq m\}$ for $1 \leq i \leq n$ since any does not occur in M. $h(t_i) < h(t)$. By the induction hypothesis, t_i is a ground term for $1 \leq i \leq n$. Therefore, t is a ground term. ∎

PROOF OF LEMMA 5.5.12 (p. 114): The proof is by induction on the height $h(t)$ of t.

Basis. $h(t) = 0$. t is either a constant or a variable. If t is a constant then $t\theta = t$. If t is a variable then $M = any$ by equation 5.15. Therefore, $t \in \check{\gamma}^t(M) \rightarrow t\theta \in \check{\gamma}^t(M)$ when $h(t) = 0$.

Induction. $h(t) > 0$. There are a function symbol $f/n \in \Sigma$ and terms $t_1, \cdots, t_n$ such that $t = f(t_1, \cdots, t_n)$. Let

$$f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$$

By equation 5.15, there are $M_1, \cdots, M_m$ such that $M = c(M_1, \cdots, M_m)$ and $t_i \in \check{\gamma}^t(T_i\{\beta_j/M_j \mid 1 \leq j \leq m\})$. $h(t_i) < h(t)$. By the induction hypothesis, $t_i\theta \in \check{\gamma}^t(T_i\{\beta_j/M_j \mid 1 \leq j \leq m\})$. Therefore, $t\theta = f(t_1\theta, \cdots, t_n\theta) \in \check{\gamma}^t(M)$ by equation 5.15. So, $t \in \check{\gamma}^t(M) \rightarrow t\theta \in \check{\gamma}^t(M)$ when $h(t) > 0$. ∎

PROOF OF LEMMA 5.5.20 (p. 117): The proof is done by induction on the height $h(t)$ of t.

Base. $h(t) = 0$ implies that t is either is variable or a constant. If t is a variable then $\mu(t) \check{\sqsubseteq}^t \nu(t)$ and, by equation 5.17, $\check{A}^t(t,\mu) \check{\sqsubseteq}^t \check{A}^t(t,\nu)$. If t is a constant then $\check{A}^t(t,\mu) = \check{A}^t(t,\nu)$. So, $\check{A}^t(t,\mu) \check{\sqsubseteq}^t \check{A}^t(t,\nu)$ for $h(t) = 0$.

Induction. $h(t) > 0$. Let $t = f(t_1, \cdots, t_n)$. $\forall 1 \leq i \leq n.\check{A}^t(t_i,\mu) \check{\sqsubseteq}^t \check{A}^t(t_i,\nu)$ by the induction hypothesis. Let

$$f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$$

$$\kappa_i = lmts(\check{A}^t(t_i,\mu), T_i)$$

$$\kappa_i' = lmts(\check{A}^t(t_i,\nu), T_i)$$

If $\check{A}^t(t,\mu) = any$ then $\exists 1 \leq i \leq n.\kappa_i = \mathit{fail}$ by equation 5.17. $\kappa_i' = \mathit{fail}$ by lemma 5.5.17.(b), $\check{A}^t(t,\nu) = any$ by equation 5.17 and therefore $\check{A}^t(t,\mu) \check{\sqsubseteq}^t \check{A}^t(t,\nu)$ when $\check{A}^t(t,\mu) = any$. If $\check{A}^t(t,\mu) \neq any$ then $\forall 1 \leq i \leq n.\kappa_i \neq \mathit{fail}$ and $\check{A}^t(t,\mu) = c(\beta_1, \cdots, \beta_m)\kappa$ where

$$\kappa = \bigcup_{1 \leq j \leq m} \{\beta_j / (\check{\bigsqcup}^t{}_{\substack{1 \leq i \leq n \\ \beta_j \in dom(\kappa_i)}} \{\beta_j \kappa_i\})\}$$

If $\exists 1 \leq i \leq n.\kappa_i' = \mathit{fail}$ then $\check{A}^t(t,\nu) = any$ and hence $\check{A}^t(t,\mu) \check{\sqsubseteq}^t \check{A}^t(t,\nu)$. Otherwise, $\check{A}^t(t,\nu) = c(\beta_1, \cdots, \beta_m)\kappa'$ where

$$\kappa' = \bigcup_{1 \leq j \leq m} \{\beta_j / (\check{\bigsqcup}^t{}_{\substack{1 \leq i \leq n \\ \beta_j \in dom(\kappa_i')}} \{\beta_j \kappa_i'\})\}$$

$\forall 1 \leq j \leq m.\beta_j\kappa \check{\sqsubseteq}^t \beta_j\kappa'$ since $\forall 1 \leq i \leq n.\forall \beta \in vars(T_i).\beta\kappa_i \check{\sqsubseteq}^t \beta\kappa_i'$ by lemma 5.5.17.(c). So, $\check{A}^t(t,\mu) \check{\sqsubseteq}^t \check{A}^t(t,\nu)$ when $\check{A}^t(t,\mu) \neq any$. Therefore, $\check{A}^t(t,\mu) \check{\sqsubseteq}^t \check{A}^t(t,\nu)$ for $h(t) > 0$. ∎

Proof of lemma 5.5.22 (p. 117): Since both $t \in \check{\gamma}^t(M)$ and $\check{\alpha}^t(t) \check{\sqsubseteq}^t M$ hold when $M = any$, we assume that $M \neq any$ and accomplish the proof by induction on the height $h(t)$ of t.

Base. $h(t) = 0 \wedge M \neq any$ implies that t is a constant. Let

$$t : c(\beta_1, \cdots, \beta_m) \leftarrow$$

Then

$$\begin{array}{rcll} \check{\alpha}^t(t) \check{\sqsubseteq}^t M & \leftrightarrow & c(empty, \cdots, empty) \check{\sqsubseteq}^t M & \textit{by equation } 5.16 \\ & \leftrightarrow & \exists M_1, \cdots, M_m.M = c(M_1, \cdots, M_m) & \textit{by definition } 5.5.1 \\ & \leftrightarrow & t \in \check{\gamma}^t(M) & \textit{by equation } 5.15 \end{array}$$

So, $t \in \check{\gamma}^t(M) \leftrightarrow \check{\alpha}^t(t) \check{\sqsubseteq}^t M$ when $h(t) = 0$.

Induction. $h(t) > 0$ implies that there are a function symbol $f/n \in \Sigma$ and terms $t_1, \cdots, t_n$ such that $t = f(t_1, \cdots, t_n)$. Let

$$f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$$

$$\kappa_i = lmts(\check{\alpha}^t(t_i), T_i)$$

$$M = c(\beta_1, \cdots, \beta_m)\kappa''$$

We have that

$$\begin{array}{rcll} t \in \check{\gamma}^t(M) & \leftrightarrow & \forall 1 \leq i \leq n.(t_i \in \check{\gamma}^t(T_i\kappa'')) & (\textit{by equation } 5.15) \\ & \leftrightarrow & \forall 1 \leq i \leq n.\check{\alpha}^t(t_i) \check{\sqsubseteq}^t T_i\kappa'' & (\textit{by the induction hypothesis}) \\ & \leftrightarrow & \forall 1 \leq i \leq n.(\kappa_i \neq \mathit{fail} \wedge \forall \beta \in vars(T_i).\beta\kappa_i \check{\sqsubseteq}^t \beta\kappa'') & (\textit{by lemma } 5.5.17.(d)) \\ & \leftrightarrow & \forall 1 \leq j \leq m.(\check{\bigsqcup}^t_{\substack{1 \leq i \leq n \\ \beta_j \in dom(\kappa_i)}} \{\beta_j\kappa_i\} \check{\sqsubseteq}^t \beta_j\kappa'') & \\ & \leftrightarrow & \check{\alpha}^t(t) \check{\sqsubseteq}^t M & (\textit{by equation } 5.16) \end{array}$$

Therefore, $t \in \check{\gamma}^t(M) \leftrightarrow \check{\alpha}^t(t) \check{\sqsubseteq}^t M$ when $h(t) > 0$. ∎

Proof of lemma 5.5.26 (p. 118): If $M = any$ then $\forall X \in vars(t).\nu(X) = any$ by equation 5.18 and hence $\forall X \in vars(t).(\check{\alpha}^t(X\theta) \check{\sqsubseteq}^t \nu(X))$ by lemma 5.5.3. The proof for the case $M \neq any$ is done by induction on the height $h(t)$ of t.

Basis. $h(t) = 0$ implies either t is a variable or t is a constant. If t is a variable then $\check{\alpha}^t(t\theta) \check{\sqsubseteq}^t M \rightarrow \forall X \in vars(t).(\check{\alpha}^t(X\theta) \check{\sqsubseteq}^t \nu(X))$ since $\nu(t) = M$ by equation 5.18. If t is a constant then $vars(t) = \emptyset$ and therefore $\forall X \in vars(t).\check{\alpha}^t(X\theta) \check{\sqsubseteq}^t \nu(X)$. So,

$$\check{\alpha}^t(t\theta) \check{\sqsubseteq}^t M \rightarrow \forall X \in vars(t).\check{\alpha}^t(X\theta) \check{\sqsubseteq}^t \nu(X)$$

when $h(t) = 0$.

Induction. $h(t) > 0$ implies that there are a function symbol $f/n \in \Sigma$ and terms $t_1, \cdots, t_n$ such that $t = f(t_1, \cdots, t_n)$. Let

$$f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$$

$$\kappa_i = lmts(\check{\alpha}^t(t_i\theta), T_i)$$

$$\kappa = \bigcup_{1 \leq j \leq m} \{\beta_j / (\check{\bigsqcup}^t_{\substack{1 \leq i \leq n \\ \beta_j \in dom(\kappa_i)}} \{\beta_j\kappa_i\})\}$$

Since $M \neq any$, $\check{\alpha}^t(t\theta) = c(\beta_1, \cdots, \beta_m)\kappa$ and $\kappa_i \neq \mathit{fail}$ for $1 \leq i \leq n$ by equations 5.16 and 5.17. $M \neq any$ and $\check{\alpha}^t(t\theta) = c(\beta_1, \cdots, \beta_m)\kappa \check{\sqsubseteq}^t M$ imply that there is a type-substitution κ' such that $M = c(\beta_1, \cdots, \beta_m)\kappa'$ and $\forall 1 \leq j \leq m.\beta_j\kappa \check{\sqsubseteq}^t \beta_j\kappa'$. Let

$$\nu_i = gmvt(T_i\kappa', t_i)$$

We have that

$$\begin{array}{rcll}
\kappa_i = lmts(\check{\alpha}^t(t_i\theta), T_i) & \to & \check{\alpha}^t(t_i\theta) \check{\sqsubseteq}^t T_i\kappa_i & \textit{by lemma } 5.5.17.(a) \\
& \to & \check{\alpha}^t(t_i\theta) \check{\sqsubseteq}^t T_i\kappa & \textit{by definition of } \kappa \\
& \to & \check{\alpha}^t(t_i\theta) \check{\sqsubseteq}^t T_i\kappa' & \textit{by the transitivity of } \check{\sqsubseteq}^t \\
& \to & \forall X \in vars(t_i).(\check{\alpha}^t(X\theta) \check{\sqsubseteq}^t \nu_i(X)) & \textit{by the induction hypothesis} \\
& \to & \forall X \in vars(t).(\check{\alpha}^t(X\theta) \check{\sqsubseteq}^t \nu(X)) & \textit{by equation } 5.18
\end{array}$$

So, by equation 5.16, $\check{\alpha}^t(t\theta) \check{\sqsubseteq}^t M \to \forall X \in vars(t).(\check{\alpha}^t(X\theta) \check{\sqsubseteq}^t \nu(X))$ when $h(t) > 0$. ∎

Proof of lemma 5.5.28 (p. 119): A mono-variable-typing whose domain is $\mathcal{V}$ is an abstract instantiation state on $\mathcal{V}$. Let $\mathcal{I}_1 = gmvt(\mathcal{I}(X), t)$ and $\mathcal{I}_2 = \{Y/(\mathcal{I}(Y)\check{\sqcap}^t\mathcal{I}_1(Y)) \mid Y \in vars(t)\}$. Since $\sigma \in \gamma_i[\![\mathcal{V}]\!](\mathcal{I})$, $X\sigma \in \check{\gamma}^t(\mathcal{I}(X))$ and $Y\sigma \in \check{\gamma}^t(\mathcal{I}(Y))$ for $Y \in vars(t)$. By lemma 5.5.12, $t\sigma\eta = X\sigma\eta \in \check{\gamma}^t(\mathcal{I}(X))$. Therefore, $Y\sigma\eta \in \check{\gamma}^t(\mathcal{I}_1(Y))$ for $Y \in vars(t)$ by lemmas 5.5.22 and 5.5.26. By lemma 5.5.12, $Y\sigma\eta \in \check{\gamma}^t(\mathcal{I}(Y))$. Therefore, $Y\sigma\eta \in \check{\gamma}^t(\mathcal{I}_2(Y)) = \check{\gamma}^t(\mathcal{I}_0(Y))$ by lemma 5.5.26 and equation 5.19. By lemma 5.5.21.(a) and lemma 5.5.26, $X\sigma\eta = t\sigma\eta \in \check{\gamma}^t(\check{A}^t(t, \mathcal{I}_2))$. So, $X\sigma\eta \in \check{\gamma}^t(\mathcal{I}(X)\check{\sqcap}^t\check{A}^t(t,\mathcal{I}_2)) \subseteq \check{\gamma}^t(\mathcal{I}_0(X))$ by equation 5.19 and lemma 5.5.7.(b). ∎

A.4 Proofs for Chapter 7

Proof of lemma 7.1.5 (p. 141): Let L be the root of T and $RI(T, L) = L \leftarrow L_1, L_2, \cdots, L_k$. The proof is done by induction on $h(T)$.

Basis. $h(T) = 1$. $L_1, L_2, \cdots, L_k$ are valid in $\mathcal{M}$ since they are leaves of T. So, $RI(T, L) = L \leftarrow L_1, L_2, \cdots, L_k$ is invalid in $\mathcal{M}$ since L is invalid in $\mathcal{M}$. So, the lemma holds when $h(T) = 1$.

Induction. Now suppose the lemma holds for any T' with $h(T') < k$. If $L_1, L_2, \cdots, L_k$ are valid in $\mathcal{M}$, then $RI(T, L)$ is invalid in $\mathcal{M}$ since L is invalid in $\mathcal{M}$. Otherwise, at least one of them, say L_i, is invalid in $\mathcal{M}$. $h(T_{L_i}) \leq k - 1 < k$, L_i is invalid in $\mathcal{M}$ and each leaf of T_{L_i} is valid in $\mathcal{M}$. So, by the induction hypothesis, there is a node L' of T_{L_i} such that $RI(T_{L_i}, L')$ is invalid in $\mathcal{M}$. Therefore, there is a node L' of T such that $RI(T, L')$ is invalid in $\mathcal{M}$ since T_{L_i} is a sub-tree of T. ∎

Proof of lemma 7.1.12 (p. 143): Let $L = A$ in case (1). W is satisfiable in $\mathcal{M}$ by definition 7.1.11. Let η be a substitution such that $W\eta$ is valid in $\mathcal{M}$. By definition 2.3.2, it suffices to prove that, for every clause $A'' \leftarrow W''$ of $\mathcal{P}$ such that $A\eta$ and A'' unify with mgu θ, $W''\theta$ is unsatisfiable in $\mathcal{M}$. If A and A'' unify with mgu θ_1 then $W' = (W_1, W'', W_2)\theta_1$ is unsatisfiable in $\mathcal{M}$ by definition 7.1.11. Due to renaming, $dom(\eta) \cap vars(A'') = \emptyset$ and hence $A\eta\theta = A''\theta = A''\eta\theta$. Therefore, $\eta\theta$ is a unifier of A and A'' and there is some substitution ξ such that $\eta\theta = \theta_1\xi$. Because W' is unsatisfiable in $\mathcal{M}$, $W'\xi = (W_1, W'', W_2)\eta\theta$ is unsatisfiable in $\mathcal{M}$. Because $W\eta$ is valid in $\mathcal{M}$, $W\eta\theta$ is valid in $\mathcal{M}$. Therefore, $W''\eta\theta$ is unsatisfiable in $\mathcal{M}$. So, $W''\theta$ is unsatisfiable in $\mathcal{M}$ because $dom(\eta) \cap vars(W'') = \emptyset$ due to the process of standardisation-apart.

We prove by contradiction that $\mathcal{P}$ must succeed on A_w without instantiating A_w in case (2). If $\mathcal{P}$ does not succeed on A_w without instantiating A_w, then, by the weaker safeness condition for the negation as failure rule, $\mathcal{P}$ finitely fails on A_w. The only child of G is $G'' = \leftarrow W'' = \leftarrow (W_1, W_2)$. W is satisfiable in $\mathcal{M}$ by definition 7.1.11, which implies that W'' is satisfiable in $\mathcal{M}$. This is in contradiction with G being a critical node of T w.r.t. $\mathcal{M}$ by definition 7.1.11. Therefore, $\mathcal{P}$ must succeed on A_w without instantiating A_w. That W is satisfiable in $\mathcal{M}$ implies that $\neg A_w$ is satisfiable in $\mathcal{M}$. So, A_w is invalid in $\mathcal{M}$. By definition 2.3.1, A_w is an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$. ∎

Proof of lemma 7.1.13 (p. 144): The proof is done by induction on $h(T)$. Let $h(T) = 0$. $\leftarrow W$ is a critical node of T w.r.t. $\mathcal{M}$ because W is satisfiable in $\mathcal{M}$ and $\leftarrow W$ is a leaf of T.

Now suppose that the lemma holds for any T' with $h(T') < k$. Let $h(T) = k$. If W' is unsatisfiable in $\mathcal{M}$ for each child $\leftarrow W'$ of $\leftarrow W$, then $\leftarrow W$ is a critical node of T w.r.t. $\mathcal{M}$. Otherwise, there is at least one child, say $\leftarrow W'$, of $\leftarrow W$ such that W' is satisfiable in $\mathcal{M}$. $T_{\leftarrow W'}$ is a goal-labelled tree, the root of $T_{\leftarrow W'}$ is $\leftarrow W'$, W' is satisfiable in $\mathcal{M}$, and $h(T_{\leftarrow W'}) < k$. By the induction hypothesis, these is a node of $T_{\leftarrow W'}$ that is a critical node of $T_{\leftarrow W'}$ w.r.t. $\mathcal{M}$. This node is also a critical node of T w.r.t. $\mathcal{M}$ because $T_{\leftarrow W'}$ is a subtree of T. ∎

Proof of lemma 7.1.15 (p. 145): Since the root of T is admissible and there is an active node in T which is inadmissible, there must be a one-level tree S in T such that, letting $< p, \sigma >$ be the root of S and $< q, \theta >$ be the the rightmost child of $< p, \sigma >$, $< p, \sigma >$ is admissible and $< q, \theta >$ is inadmissible. Either each evaluated leaf of S is valid in $\mathcal{M}$ or S has an evaluated leaf L that is invalid in $\mathcal{M}$. In the former case, $(< p, \sigma >, < q, \theta >)$ is a diverging invocation of $\mathcal{P}$ w.r.t. $\mathcal{M}$ by definition 2.3.6. In the latter case, if $L = \neg A_m$ for some atom then A_m is an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ by definition 2.3.1.(2) as A_m is satisfiable in $\mathcal{M}$ and $\mathcal{P}$ finitely fails on A_m. If $L = A_w$ is positive then A_w is an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$

by definition 2.3.1.(1) for $A_w \in SS(\mathcal{P})$. ∎

PROOF OF LEMMA 7.3.2 (p. 152): The proof is done by induction on $(h(T), n(T))$. Let L be the root of T. Define $(h_1, n_1) <' (h_2, n_2) \stackrel{def}{=} (h_1 < h_2 \wedge n_1 \leq n_2) \vee (h_1 \leq h_2 \wedge n_1 < n_2)$. Relation $<'$ is a partial ordering on binary tuples of natural numbers.

Let $h(T) = 1$. $invalid\_impl(T, C) \in SS(\pi)$ implies $root\_impl(T, C) \in SS(\pi)$. So, $C = RI(T, L)$ is invalid in $\mathcal{M}$ since $h(T) = 1$. Thus the lemma holds for any T with $(h(T), n(T)) = (1, k)$ where k is a natural number.

Suppose the lemma holds for any T' with $(h(T'), n(T')) <' (k_h, k_n)$. Let $h(T) = k_h \neq 1$ and $n(T) = k_n$. $height(T, 1) \notin SS(\pi)$ since $h(T) \neq 1$. So, $invalid\_impl(T, C) \in SS(\pi)$ implies either (a) $valid(L^1) \in SS(\pi)$ and $invalid\_impl(T \backslash T_{L^1}, C) \in SS(\pi)$ or (b) $invalid\_impl(T_{L^1}, C) \in SS(\pi)$ and $valid(L^1) \notin SS(\pi)$. In case (b), T_{L^1} is a literal-labelled tree, L^1 is invalid in $\mathcal{M}$ and each leaf of T_{L^1} is valid in $\mathcal{M}$, and $invalid\_impl(T_{L^1}, C) \in SS(\pi)$. Because $(h(T_{L^1}), n(T_{L^1})) <' (k_h, k_n)$, by the induction hypothesis, C is invalid in $\mathcal{M}$ and $C = RI(T_{L^1}, L')$ for a node L' of T_{L^1}. In case (a), $T \backslash T_{L^1}$ is a literal-labelled tree, the root of $T \backslash T_{L^1}$ is invalid in $\mathcal{M}$, each leaf of $T \backslash T_{L^1}$ is valid in $\mathcal{M}$, and $invalid\_impl(T \backslash T_{L^1}, C) \in SS(\pi)$. Because $(h(T \backslash T_{L^1}), n(T \backslash T_{L^1})) <' (k_h, k_n)$, by the induction hypothesis, C is invalid in $\mathcal{M}$ and $C = RI(T \backslash T_{L^1}, L')$ for a node L' of $T \backslash T_{L^1}$. ∎

PROOF OF LEMMA 7.3.3 (p. 152): $invalid\_impl/2$ searches T by constructing a sequence of literal-labelled trees: $T^0, T^1, \cdots$. For $j \geq 1$, $invalid\_impl/2$ selects a node L^1 of T^j. If L^1 is invalid in $\mathcal{M}$, then $T^{j+1} = T^j_{L^1}$. Otherwise, $T^{j+1} = T^j \backslash T^j_{L^1}$. $invalid\_impl/2$ continues this process until, for some q, $h(T^q) = 1$. Letting L' be the root of T^q, $C = RI(T^q, L') = RI(T, L')$ is invalid in $\mathcal{M}$.

According to the way that $T^0, T^1, \cdots$ are constructed, each T^j for $0 \leq j \leq q$ is a literal-labelled tree whose root is invalid in $\mathcal{M}$ and each of whose leaves is valid in $\mathcal{M}$. For any $k > j$, $h(T^k) \leq h(T^j)$. Therefore, it suffices to prove that, for any T^j with $h(T^j) > 1$, there is a $k > j$ such that $h(T^k) < h(T^j)$. If a node L^1, with L^1 being invalid in $\mathcal{M}$, is selected in constructing $T^{j'}$ where $j' > j$, then $h(T^{j'}) < h(T^j)$. Therefore, it causes no loss of generality to assume that $invalid\_impl/2$ always selects literals that are valid in $\mathcal{M}$ when constructing $T^{j+1}, T^{j+2}, \cdots$ until, for some k, $h(T^k) < h(T^j)$. For any $j' > j$, $h(T^{j'}) = h(T^j)$ iff none of the nodes on the longest path of T^j is selected by $invalid\_impl/2$. Because T^j has only a finite number of nodes, $invalid\_impl/2$ has eventually to select a node on the longest path. So, there is some $k > j$ such that $h(T^k) < h(T^j)$. This implies that there is some q such that $h(T^q) = 1$. Letting L' be the root of T^q, $C = RI(T^q, L') = RI(T, L')$ is invalid in $\mathcal{M}$. ∎

PROOF OF LEMMA 7.3.4 (p. 152): In case (1), $T_{\leftarrow W^1}$ is a goal-labelled tree, the root of $T_{\leftarrow W^1}$ is $\leftarrow W^1$ and W^1 is satisfiable in $\mathcal{M}$. By lemma 7.1.13, there is a node $\leftarrow W'$ of $T_{\leftarrow W^1}$ such that $\leftarrow W'$ is a critical node of $T_{\leftarrow W^1}$ w.r.t. $\mathcal{M}$. $\leftarrow W'$ is a critical node of T w.r.t. $\mathcal{M}$ since $T_{\leftarrow W^1}$ is a sub-tree of T. So, (1) holds.

We prove (2) by induction on $l(\leftarrow W^1, T)$. If $l(\leftarrow W^1, T) = 1$, $\leftarrow W^1$ is a child of $\leftarrow W$. Let $\leftarrow W_1, \cdots, \leftarrow W_n$ be the children of $\leftarrow W$. Then $W^1 = W_j$ for some $1 \le j \le n$. If W_k is unsatisfiable in $\mathcal{M}$ for all $1 \le k \le n \wedge k \ne j$, then $\leftarrow W$ is a critical node of T w.r.t. $\mathcal{M}$. Otherwise, there is at least one child of $\leftarrow W$ other than $\leftarrow W^1$, say $\leftarrow W_i$, such that W_i is satisfiable in $\mathcal{M}$. By lemma 7.1.13, there is a node $\leftarrow W'$ of $T_{\leftarrow W_i}$ such that $\leftarrow W'$ is a critical node of $T_{\leftarrow W_i}$ w.r.t. $\mathcal{M}$. $\leftarrow W'$ is a critical node of T w.r.t. $\mathcal{M}$ since $T_{\leftarrow W_i}$ is a sub-tree of T. Therefore, the lemma holds for $l(\leftarrow W^1, T) = 1$.

Suppose that the lemma holds for any node $\leftarrow W''$ of T with $l(\leftarrow W'', T) < k$. Let $l(\leftarrow W^1, T) = k$, $k > 1$ and $\leftarrow W^2$ be the parent node of $\leftarrow W^1$. $l(\leftarrow W^2, T) \le k - 1 < k$. If W^2 is unsatisfiable in $\mathcal{M}$ then by the induction hypothesis there is a node $\leftarrow W'$ of $T - T_{\leftarrow W^2}$ such that $\leftarrow W'$ is a critical node of T w.r.t. $\mathcal{M}$. If W^2 is satisfiable in $\mathcal{M}$, then, because $l(\leftarrow W^1, \leftarrow W^2) = 1$, it is proved by the preceding paragraph that there is a node $\leftarrow W'$ of $T_{\leftarrow W^2} - T_{\leftarrow W^1}$ such that $\leftarrow W'$ is a critical node of T w.r.t. $\mathcal{M}$. So, for any node $\leftarrow W^1$ of T with $l(\leftarrow W^1, T) = k$, if W^1 is unsatisfiable in $\mathcal{M}$, then there is a node $\leftarrow W'$ of $T - T_{\leftarrow W^1}$ such that $\leftarrow W'$ is a critical node of T w.r.t. $\mathcal{M}$. ∎

PROOF OF LEMMA 7.3.5 (p. 152): The proof is done by induction on $(h(T), n(T))$. If $h(T) = 0$, then $critical(T, G) \in SS(\pi)$ implies $G = \leftarrow W$. $G = \leftarrow W$ is a critical node of T w.r.t. $\mathcal{M}$ since W is satisfiable in $\mathcal{M}$ and $G = \leftarrow W$ is a leaf of T. Therefore, the lemma holds when $h(T) = 0$.

Suppose the lemma holds for T' with $(h(T'), n(T')) <' (k_h, k_n)$. Let $h(T) = k_h \ne 0$ and $n(T) = k_n$. $critical(T, N) \in SS(\pi)$ implies either (a) $critical(T_{\leftarrow W^1}, G) \in SS(\pi)$ and $satisfiable(W^1) \in SS(\pi)$ for some W^1 or (b) $critical(T - T_{\leftarrow W^1}, G) \in SS(\pi)$ and $satisfiable(W^1) \notin SS(\pi)$ for some W^1. In case (a), $T_{\leftarrow W^1}$ is a goal-labelled tree, $\leftarrow W^1$ is the root of $T_{\leftarrow W^1}$, W^1 is satisfiable in $\mathcal{M}$, and $(h(T_{\leftarrow W^1}), n(T_{\leftarrow W^1})) <' (k_h, k_n)$. By the induction hypothesis, G is a critical node of $T_{\leftarrow W^1}$ w.r.t. $\mathcal{M}$ and hence a critical node of T w.r.t. $\mathcal{M}$. In case (b), $T - T_{\leftarrow W^1}$ is a goal-labelled tree, $\leftarrow W$ be the root of T, W is satisfiable in $\mathcal{M}$ and $(h(T - T_{\leftarrow W^1}), n(T - T_{\leftarrow W^1})) <' (k_h, k_n)$. By the induction hypothesis, G is a critical node of

$T - T_{\leftarrow W^1}$ w.r.t. $\mathcal{M}$ and hence a critical node of T w.r.t. $\mathcal{M}$. ∎

PROOF OF LEMMA 7.3.6 (p. 153): The proof is done by induction on $n(T)$ using lemma 7.3.4. *critical*/2 constructs a sequence of goal-labelled trees $T^0, T^1, \cdots$. $T^0 = T$. For each $j \geq 1$, *critical*/2 selects a non-root node $\leftarrow W^1$ of T^j and constructs T^{j+1} as follows. If W^1 is unsatisfiable in $\mathcal{M}$, then $T^{j+1} = T^j - T^j_{\leftarrow W^1}$. Otherwise, $T^{j+1} = T^j_{\leftarrow W^1}$. *critical*/2 continues this process until, for some q, $h(T^q) = 0$. The root of T^q is a critical node of T^j w.r.t. $\mathcal{M}$ for each $0 \leq j \leq q$ by lemma 7.3.4.

For $j > 1$, a node that is a critical node of T^{j+1} w.r.t. $\mathcal{M}$ is also a critical node of T^j w.r.t. $\mathcal{M}$. Therefore, a node that is a critical node of T^q w.r.t. $\mathcal{M}$ is a critical node of T^0 w.r.t. $\mathcal{M}$. It remains to show that there is a q such that $h(T^q) = 0$. This is equivalent to proving that, for any T^j with $h(T^j) > 1$, there is a $k > j$ such that $h(T^k) < h(T^j)$. If a node $\leftarrow W^1$ such that W^1 is satisfiable in $\mathcal{M}$ is selected in constructing $T^{j'}$ where $j' > j$, then $h(T^{j'}) < h(T^j)$. Therefore, it causes no loss of generality to assume that *critical*/2 always selects a node $\leftarrow W^1$ such that W^1 is unsatisfiable in $\mathcal{M}$ when constructing $T^{j'}$ for $j' = j+1, j+2, \cdots$ until, for some $j' = k$ such that $h(T^k) < h(T^j)$. For any $j' > j$, $h(T^{j'}) = h(T^j)$ iff any node on the longest path of T^j is not selected by *critical*/2 when constructing $T^{j+1}, T^{j+2}, \cdots, T^{j'}$. Because T^j has only finite number of nodes, *critical*/2 has eventually to select a node on the longest path. So, there is some $k > j$ such that $h(T^k) < h(T^j)$. This implies that there is some q such that $h(T^q) = 0$. ∎

PROOF OF LEMMA 7.3.8 (p. 153):

Proof of (1) is done by induction on the number of leaves in S. Let the number of leaves in S be 1. Since $diverge(S, D) \in SS(\pi)$, $D = (< p, \sigma >, < q, \theta >)$. (1) holds since $< q, \theta >$ is the only leaf in S.

Let the number of leaves in S be $k \neq 1$ and suppose that (1) holds for any S' such that the number of leaves in S' is less than k. Since $diverge(S, D) \in SS(\pi)$, either (a) $leftmost(S, \neg A_m) \in SS(\pi)$, $satisfiable(A_m) \in SS(\pi)$ and $insufficiency(A_m, D)$ or (b) $leftmost(S, \neg A_m) \in SS(\pi)$, $satisfiable(A_m) \notin SS(\pi)$ and $diverge(S \backslash T_{\neg A_m}, D) \in SS(\pi)$ or (c) $leftmost(S, A_w) \in SS(\pi)$, A_w is positive, $valid(A_w) \in SS(\pi)$ and $diverge(S \backslash T_{A_w}, D) \in SS(\pi)$ or (d) $leftmost(S, A_w) \in SS(\pi)$, A_w is positive, $valid(A_w) \notin SS(\pi)$ and $inconsistency(A_w, D) \in SS(\pi)$. In case (a), A_m is an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ since $\mathcal{P}$ finitely fails on A_m. Hence, (1) follows from theorem 7.3.9. In case (b), (1) follows from the induction hypothesis since A_m is unsatisfiable in $\mathcal{M}$ and the number of leaves in $S \backslash T_{\neg A_m}$

is $k-1$. In case (c), (1) follows from the induction hypothesis since A_w is valid in $\mathcal{M}$ and the number of leaves in $S \backslash T_{A_w}$ is $k-1$. In case (d), A_w is an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ since since $\mathcal{P}$ succeeds on A_w. Therefore, (1) follows from theorem 7.3.9.

Proof of (2). Without loss of generality, we assume that the number of leaves in S is greater than 1. Let $S^0 = S$. *diverge*/2 repeatedly constructs a one-level tree S^{j+1} for $j = 0, 1, \cdots$ by discarding the leftmost leaf L of S^j unless either (a) $< q, \theta >$ is the leftmost leaf of S or (b) L is positive and L is invalid in $\mathcal{M}$ or (c) $L = \neg A_m$ for some A_m and A_m is satisfiable in $\mathcal{M}$. In case (a), $diverge(S, (< p, \sigma >, < q, \theta >)) \in SS(\pi)$ and every leaf of S other than $< q, \theta >$ is valid in $\mathcal{M}$. In the case (b), L is an inconsistency symptom of of $\mathcal{P}$ w.r.t. $\mathcal{M}$ since $\mathcal{P}$ succeeds on L. In the case (c), A_m is an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ since $\mathcal{P}$ finitely fails on A_m. In either case (b) or case (c), (2) follows from theorem 7.3.10. ∎

Proof of theorem 7.3.9 (p. 154): We first prove (1) and (2). Let n be the sum of the number of calls to *inconsistency*/2 and the number of calls to *insufficiency*/2 in the refutation of $\pi \cup \{\leftarrow inconsistency(A, D)\}$ or $\pi \cup \{\leftarrow insufficiency(A, D)\}$. The proof is by induction on n by using lemmas 7.3.2 and 7.3.5.

Let $n = 1$. If A is an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ and $inconsistency(A, D) \in SS(\pi)$, then $cpp\_1(A, T) \in SS(\pi)$ and $invalid\_impl(T, D) \in SS(\pi)$ for some T, and D is an instance of a clause of $\mathcal{P}$. D is invalid in $\mathcal{M}$ by lemma 7.3.2 because $invalid\_impl(T, D) \in SS(\pi)$. Therefore, D is an inconsistent clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$. So, (1) holds for $n = 1$. (2) can be proved to hold for $n = 1$ in a similar way.

Suppose that (1) and (2) hold for any $n < k$. Let $n = k > 1$. If A is an inconsistency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ and $inconsistency(A, D) \in SS(\pi)$, then there are some T and some $C = (\neg A_m \leftarrow \textsf{true})$ such that $cpp\_1(A, T) \in SS(\pi)$, $invalid\_impl(T, C) \in SS(\pi)$ and $insufficiency(A_m, D) \in SS(\pi)$. C is invalid in $\mathcal{M}$ by lemma 7.3.2 and hence A_m is satisfiable in $\mathcal{M}$. Therefore, A_m is an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ since $\mathcal{P}$ finitely fails on A_m. The sum of the number of calls to *inconsistency*/2 and the number of the calls to *insufficiency*/2 is $k-1$ in the refutation of $\pi \cup \{\leftarrow insufficiency(A_m, D)\}$. By the induction hypothesis, D is either an inconsistent clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$ or an incompletely covered atom of $\mathcal{P}$ w.r.t. $\mathcal{M}$. So, (1) holds for $n = k$. (2) can be proved to hold for $n = k$ in a similar way.

We now prove (3). Since $operational(A, D) \in SS(\pi)$ and $D \neq no\_diagnosis$, there are an active execution tree T and a one-level tree S such that $aet(A, T) \in SS(\pi)$ and $segment(T, S) \in SS(\pi)$ and $diverge(S, D) \in SS(\pi)$. T is the active execution tree for $\mathcal{P}$ and A. By lemma 7.3.7.(1),

S is a one-level tree whose root $< p, \sigma >$ is admissible, whose righmost leaf $< q, \theta >$ is inadmissible and whose other leaves are literals on which $\mathcal{P}$ succeeds. Hence, the theorem follows from lemma 7.3.8.(1) and definition 2.3.6. ∎

PROOF OF THEOREM 7.3.10 (p. 154): We first prove (1) and (2). Let R be a fair computation rule. Let k be the rank of the successful SLDNF-derivation of $\mathcal{P} \cup \{\leftarrow A\}$ via R in case (1), and k be the rank of the finitely failed SLDNF-derivation of $\mathcal{P} \cup \{\leftarrow A\}$ via R in case (2) [74]. The proof is done by induction on k using lemmas 7.3.3 and 7.3.6.

Let $k = 0$. Let T be the proof tree for $\mathcal{P}$ and A that corresponds to the successful SLDNF-derivation of $\mathcal{P} \cup \{\leftarrow A\}$ via R. T is an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and A by definition 7.1.2. $\mathit{cpp\_1}(A, T) \in SS(\pi)$. By lemma 7.3.3, there is a node L' of T such that $C = RI(T, L')$ and $\mathit{invalid\_impl}(A, C) \in SS(\pi)$. L' is positive since $k = 0$. $\mathit{inconsistency}(A, C) \in SS(\pi)$. So, (1) holds for $k = 0$. (2) can be proved to be hold for $k = 0$ in a similar way.

Suppose that (1) and (2) hold for any $k' < n$. Let $k = n$. Let T be the proof tree for $\mathcal{P}$ and A that corresponds to the successful SLDNF-derivation of $\mathcal{P} \cup \{\leftarrow A\}$ via R. T is an $\mathcal{M}$-*cpp* for $\mathcal{P}$ and $\mathcal{M}$ by definition 7.1.2. $\mathit{cpp\_1}(A, T) \in SS(\pi)$. By lemma 7.3.3, there is a node L' of T such that $C = RI(T, L')$ and $\mathit{invalid\_impl}(A, C) \in SS(\pi)$. If L' is positive then $\mathit{inconsistency}(A, C) \in SS(\pi)$. Otherwise, $L' = \neg A_m$ and A_m is an insufficiency symptom of $\mathcal{P}$ w.r.t. $\mathcal{M}$. The rank k' of the finitely failed SLDNF-derivation of $\mathcal{P} \cup \{\leftarrow A_m\}$ is less than n. By the induction hypothesis, there is a D such that D is either an inconsistent clause instance of $\mathcal{P}$ w.r.t. $\mathcal{M}$ or an incompletely covered atom of $\mathcal{P}$ w.r.t. $\mathcal{M}$ and $\mathit{insufficiency}(A_m, D) \in SS(\pi)$. $\mathit{inconsistency}(A, D) \in SS(\pi)$. So, (1) holds for $k = n$. (2) can be proved to hold for $k = n$ in a similar way.

We now prove (3). Let T be the active execution tree for $\mathcal{P}$ and A. The lowest active node is inadmissible. Hence the theorem follows from lemmas 7.3.7.(2) and 7.3.8.(2). ∎

Bibliography

[1] A. Adam and J.-P. Laurent. LAURA, a system to debug student programs. *Artificial Intelligence*, 15(1,2):75–122, 1980.

[2] K.R. Apt. Logic programming. In J.V. Leeuwen, editor, *Handbook of Theoretical Computer Science: (Volume B) Formal Models and Semantics*, pages 493–574. Elsevier Science Publishers, Amsterdam, Oxford, 1990.

[3] H. Azzoune. Type inference in Prolog. In E. Lusk and R. Overbeek, editors, *Proceedings of the Ninth International Conference on Automated Deduction*, pages 258–277, Argonne, Illinois, USA, May 23-26 1988. Springer-Verlag, Berlin, New York, 1988. Springer-Verlag Lecture Notes in Computer Science 310.

[4] R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.

[5] A. Barr, M. Beard, and R.C. Atkinson. The computer as a tutorial laboratory: the Stanford BIP project. *International Journal of Man Machine Studies*, 8(5):567–596, 1976.

[6] L. Bouge, N. Choquet, L. Fribourg, and M.C. Gaudel. Test sets generating from algebraic specifications using Logic Programming. *Journal of System and Software*, 6(4):343–360, 1986.

[7] R. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[8] M. Bruynooghe. Adding redundancy to obtain more reliable and more readable Prolog programs. In *Proceedings of the First International Logic Programming Conference*, pages 129–133, Marseille, France, 1982.

[9] M. Bruynooghe. A practical framework for the abstract interpretation of logic progams. *Journal of Logic Programming*, 10(2):91–124, 1991.

[10] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: towards the global optimisation of Prolog programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 192–204, San Francisco, August 31- September 4 1987. The IEEE Computer Society Press, Washington, 1987.

[11] L. Byrd. Understanding the control flow of Prolog programs. In S. Tärnlund, editor, *Proceedings of the logic programming workshop*, pages 127-138, Debrecen, Hungary, 1980.

[12] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM computing surveys*, 17(4):471–522, 1985.

[13] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A generic abstract interpretation algorithm and its complexity analysis. In Furukawa [47], pages 64–78.

[14] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, New York, 1984.

[15] M. Codish, D. Dams, and Yardeni E. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In Furukawa [47], pages 79–93.

[16] M. Codish, D. Dams, G. Filé, and M. Bruynooghe. Freeness analysis for logic programs - and correctness. In D.S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 117–131, Budapest, Hungary, June 21-24 1993. The MIT Press, Cambridge Massachusetts, 1993.

[17] M. Codish, M.G. de la Banda, M. Bruynooghe, and M. Hermenegildo. Goal dependent vs. goal independent analysis of logic programs. In *Logic Programming and Automated Reasoning, 5th International Conference*, Lecture Notes in Artificial Intelligence 822, pages 304–319, Kiev, Ukraine, July 16-22 1994. Springer-Verlag, Berlin, Heidelberg, 1994.

[18] A. Cortesi and G. Filé. Abstract interpretation of logic programs: an abstract domain for groundness, sharing, freeness and compoundness analysis. *SIGPLAN Notices*, 26(9):52–61, 1991.

[19] P. Cousot and R. Cousot. Abstract interpretation: a unified framework for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the fourth annual ACM symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. The ACM Press, New York, 1977.

[20] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the sixth annual ACM symposium on Principles of programming languages*, pages 269–282, San Antonio, Texas, 1979. The ACM Press, New York, 1979.

[21] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(1, 2, 3 and 4):103–179, 1992.

[22] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. Laboratoire d'Informatique, URA 1327 du CNRS, Département de Mathématiques et d'Informatique, Ecole Normale Supérieure, June 1992.

[23] P.W. Dart and J. Zobel. A regular type language for logic programs. In Frank Pfenning, editor, *Types in Logic Programming.* The MIT Press, Cambridge, Massachusetts, 1992.

[24] S. K. Debray. Functional computations in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):451–481, 1989.

[25] S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.

[26] S. K. Debray and Nai-Wei Lin. Automatic complexity analysis of logic programs. In Furukawa [47], pages 599–613.

[27] S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–230, September 1988.

[28] *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, Linköping, Sweden, May 3-5 1993. Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden, 1993.

[29] P. Deransart. Proofs of declarative properties of logic programs. In J.Diaz and F.Orejas, editors, *Proceedings of International Joint Conference on TAPSOFT'89*, pages 207–226, Barcelona, Spain, March 1989. Springer-Verlag, Berlin, New York, 1989.

[30] N. Dershowitz and Y.-J. Lee. Deductive Debugging. In *Proceedings of 1987 Symposium of Logic Programming*, pages 298–306, San Francisco, August 31- September 4 1987. The IEEE Computer Society Press, Washington, 1987.

[31] N. Dershowitz and Y.J. Lee. Logic Debugging. *Journal of Symbolic Computation*, 15:745–773, 1993.

[32] W. Drabent, S. Nadjm-Tehrani, and J. Małuszynski. The use of assertions in algorithmic debugging. In ICOT, editor, *The Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 573–581, Tokyo, November 1988. ICOT, Tokyo, 1988.

[33] W. Drabent, S. Nadjm-Tehrani, and J. Małuszynski. Algorithmic debugging with assertions. In Harvey Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 502–521. The MIT Press, Cambridge, Massachusetts, 1989.

[34] M. Ducassé. Opium$^+$, a meta-debugger for Prolog. In Y. Kodratoff, editor, *Proceedings of the eighth ECAI*, pages 272–277, Münich, August 1-5 1988. Pitman, London, 1988.

[35] M. Ducassé. Analysis of failing Prolog executions. ICLP91 Preconference Workshop on Logic Programming Environments, Paris, June 1991. University of Orléans Technical Report LIFO N 91-61, 1991.

[36] M. Ducassé and A. M. Emde. Opium: A debugging environment for Prolog development and debugging research. *Software Engineering Notes*, 16(1):54–59, 1991.

[37] A. Edman and S.-Å. Tärnlund. Mechanization of an oracle in a debugging system. In *IJCAI83*, volume 2, pages 553–555, Karlsruhe, West Germany, August 1983. William Kaufmann Publishers, Inc., Los Altos, 1983.

[38] M. Eisenstadt. A powerful Prolog trace package. In T. O'Shea, editor, *Proceedings of the Sixth ECAI*, September 1984. North-Holland, Amsterdam, 1984.

[39] M. Eisenstadt. Retrospective zooming: A knowledge based tracing and debugging methodology for logic programming. In *IJCAI85*, pages 717–719, Los Angeles, 1985. Morgan Kaufmann Publishers, Inc., San Mateo, 1985.

[40] M. Eisenstadt and M. Brayshaw. The transparent Prolog machine (TPM): An execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):277–342, 1988.

[41] A.-M. Emde and M. Ducassé. Automated debugging of non-terminating Prolog programs. In S. Bourgault and M. Dincbas, editors, *Actes du Séminaire de Programmation en logique de Trigastél*, pages 89–103, CNET, Lannion, France, May 1990.

[42] M. Falaschi, G. Levi, and C. Palamidessi. Declarative modelling of the operational behavior of logic programs. *Theoretical Computer Science*, 69(3):289–318, 1989.

[43] G. Ferrand. Error diagnosis in logic programming, an adaptation of E.Y. Shapiro's method. *Journal of Logic Programming*, 4(3):177–198, 1987.

[44] G. Ferrand. The notions of symptom and error in declarative diagnosis of logic programs. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* [28], pages 33–49.

[45] P. Fritzson, T. Gyimothy, M. Kamkar, and N. Shahmehri. Generalized algorithmic debugging and testing. *SIGPLAN Notices*, 26(6):317–326, 1991.

[46] T.W. Früehwirth. Using meta-interpreters for polymorphic type checking. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-programming in Logic*, pages 339–351, Leuven, Belgium, April 4-6 1990. Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, B-3030 Leuven, Belgium, 1990.

[47] K. Furukawa, editor. *Proceedings of the Eighth International Conference on Logic Programming*, Paris, June 24-28, 1991. The MIT Press, Cambridge, Massachusetts, 1991.

[48] I.P. Goldstein. Summary of MYCROFT. *Artificial Intelligence*, 6(3):249–288, 1975.

[49] J.D. Gould. Psychological evidence on how people debug computer programs. *International Journal of Man Machine Studies*, 7(2):151–182, 1975.

[50] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *The Seventh Annual ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 17-19 1990. The ACM Press, New York, 1990.

[51] N. Heintze and J. Jaffar. Semantic types for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[52] K. Horiuchi and T. Kanamori. Polymorphic type inference in Prolog by abstract interpretation. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Proceedings of the Sixth Conference on Logic Programming*, pages 195–214, Tokyo, June 1987. Springer-Verlag New York, 1988.

[53] W.E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, 1976.

[54] A.M.J. Hutching, D.L. Bowen, L. Byrd, P.W.H. Chung, F.C.N. Pereira, L.M. Pereira, R.Rae, and D.H.D. Warren. Edinburgh Prolog (the new implementation) user's manual. AI Applications Institute, University of Edinburgh, 8 October 1986.

[55] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 154–165, Cleveland, Ohio, USA, 1989.

[56] D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *Journal of Logic Programming*, 13(1, 2, 3 and 4):291–314, 1992.

[57] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(1, 2, 3 and 4):205–258, 1992.

[58] W.L. Johnson. Understanding and debugging novice programs. *Artificial Intelligence*, 42(1):51–97, 1990.

[59] W.L. Johnson and E. Soloway. Proust: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, 11(3):267–275, 1985.

[60] N.D. Jones and H. Søndergaard. A semantics-based framework for abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 123–142. Ellis Horwood Limited, Chichester, 1987.

[61] T. Kanamori. Abstract interpretation based on Alexander templates. *Journal of Logic Programming*, 15(1 & 2):31–54, 1993.

[62] T. Kanamori and K. Horiuchi. Type inference in Prolog and its application. In *IJCAI85*, pages 704–707, Los Angeles, 1985. Morgan Kaufmann Publishers, Inc., San Mateo, 1985.

[63] T. Kanamori and T. Kawamura. Analyzing success patterns of logic programs by abstract hybrid interpretation. ICOT Technical Report TR-279, 1987.

[64] T. Kanamori and T. Kawamura. Abstract interpretation based on OLDT resolution. ICOT Technical Report TR-619, 1990.

[65] T. Kanamori and T. Kawamura. Abstract interpretation based on OLDT resolution. *Journal of Logic Programming*, 15(1 & 2):1–30, 1993.

[66] T. Kanamori, T. Kawamura, M. Maeji, and K.Horiuchi. Logical program diagnosis from specifications. ICOT Technical Report TR-447, March 1989.

[67] R. S. Kemp and G. A. Ringwood. An algebraic framework for abstract interpretation of definite programs. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 516–530, Austin, 1990. The MIT Press, Cambridge, Massachusetts, 1990.

[68] A. King. A synergistic analysis for sharing and groundness which traces linearity. Draft, 1994.

[69] R. A. Kowalski and K. A. Bowen, editors. *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seatle, August 15-19 1988. The MIT Press, Cambridge, Massachusetts, London, England, 1988.

[70] Y.J. Lee and N. Dershowitz. Debugging logic programs using specifications. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* [28], pages 64–78.

[71] G. Levi and M. Martelli, editors. *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon, 1989. The MIT Press, Cambridge, Massachusetts, 1989.

[72] Y. Lichtenstein and E. Shapiro. Abstract algorithm debugging. In R.A. Kowalski and K.A. Bowen [69], pages 512–531.

[73] J.W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

[74] J.W. Lloyd. *Foundations of Logic Programming, Second Edition.* Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo, 1987.

[75] C.-K. Looi. Analysing novices' programs in a Prolog intelligent teaching system. In Y. Kodratoff, editor, *Proceedings of the eighth ECAI*, pages 314–319, Münich, August 1-5 1988. Pitman, London, 1988.

[76] L. Lu. A generic declarative diagnoser for normal logic programs. In *Logic Programming and Automated Reasoning, 5th International Conference*, Lecture Notes in Artificial Intelligence 822, pages 290–304, Kiev, Ukraine, July 16-22 1994. Springer-Verlag, Berlin, Heidelberg, 1994.

[77] L. Lu and P. Greenfield. Abstract fixpoint semantics and abstract procedural semantics of definite logic programs. In *Proceedings of IEEE Computer Society 1992 International Conference on Computer Languages*, pages 147–154, Oakland, California, USA, April 20-23 1992. The IEEE Computer Society Press, Los Alamitos, California, 1992.

[78] L. Lu and P. Greenfield. An algorithm for finite approximations of definite programs and its implementation in Prolog. In Krysia Broda, editor, *4th UK Conference on Logic Programming*, Workshops in Computing, pages 62–78, London, March 30-April 1 1992. Springer-Verlag, London, 1992.

[79] L. Lu and P. Greenfield. Logic program testing based on abstract interpretataion. In *Formal Methods in Programming and Their Applications, International Conference*, Lecture Notes in Computer Science 735, pages 170–180, Novosibirsk, Russia, June 28-July 3 1993. Springer-Verlag, Berlin, Heidelberg, 1993.

[80] F.J. Lukey. Understanding and debugging programs. *International Journal of Man Machine Studies*, 12(1):189–202, 1980.

[81] M. Maeji and T. Kanamori. Top-down zooming diagnosis of logic programs. ICOT Technical Report TR-290, August 1987.

[82] K. Mannila and E. Ukkonen. Flow analysis of Prolog programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 205–214, San Francisco, August 31-September 4, 1987. The IEEE Computer Society Press, Washington, 1987.

[83] A. Mariën, G. Janssens, A. Mulkers, and M. Bruynooghe. The impact of abstract interpretation: An experiment in code generation. In Levi and Martelli [71], pages 33–47.

[84] B. Marre. Toward automatic test data set selection using algebraic specifications and Logic Programming. In K. Furukawa [47], pages 202–219.

[85] K. Marriott. Frameworks for abstract interpretation. *Acta Informatica*, 30(2):103–129, 1993.

[86] K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In R.A. Kowalski and K.A. Bowen [69], pages 733–748.

[87] K. Marriott and H. Søndergaard. Bottom-up dataflow analysis of normal logic programs. *Journal of Logic Programming*, 13(1, 2, 3 and 4):181–204, 1992.

[88] C. Mellish. Some global optimisations for a Prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1985.

[89] C. Mellish. Abstract interpretation of Prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 181–198. Ellis Horwood Limited, Chichester, 1987.

[90] P. Mishra. Towards a theory of types in Prolog. In *Proceedings of the IEEE international Symposium on Logic Programming*, pages 289–298, Atlantic City, 1984. The IEEE Computer Society Press, Washington, 1984.

[91] L.J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, 1990.

[92] W.R. Murray. Heuristic and formal methods in automatic program debugging. In *IJCAI85*, pages 15–19, Los Angeles, 1985. Morgan Kaufmann Publishers, Inc., San Mateo, 1985.

[93] W.R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems.* Morgan Kaufmann Publishers, Inc., San Mateo, 1988.

[94] K. Muthukumar and M. Hermenegildo. Determination of variable dependence information at compile-time through abstract interpretation. Technical Report ACA-ST-232-89, Microelectronics and Computer Technology Corporation, March 1989.

[95] K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In Furukawa [47], pages 49–63.

[96] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(1, 2, 3 and 4):315–347, 1992.

[97] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.

[98] S. Nadjm-Tehrani. Debugging Prolog programs declaratively. In M. Brugnooghe, editor, *Proceedings of the Second Workshop on Meta-Programming in Logic*, pages 137–155, Leuven, Belgium, April 4-6 1990. Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, B-3030 Leuven, Belgium, 1990.

[99] L. Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10(3):255–285, 1992.

[100] L. Naish, P.W. Dart, and J. Zobel. The NU-Prolog Debugging Environment. In G. Levi and M. Martelli [71], pages 521–536.

[101] U. Nilsson. Towards a framework for the abstract interpretation of logic programs. In P. Deransart, B. Lorho, and J. Małuszynski, editors, *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming*, pages 68–82, Orléans, France, May 1988. Springer-Verlag, Berlin, New York, 1989.

[102] U. Nilsson. Systematic semantic approximations of logic programs. In *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming*, pages 293–306, Berlin, 1990. Springer-Verlag, Berlin, New York, 1990.

[103] U. Nilsson. Abstract interpretation: A kind of magic. In J. Małuszynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming: Proceedings of the Third International Symposium PLILP'91*, pages 299–309, Passau, Germany, August 1991. Springer, Berlin, New York, 1991.

[104] R. A. O'Keefe. Finite fixed-point problems. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic programming*, volume 2, pages 729–743, Melbourne, May 1987. The MIT Press, Cambridge, Massachusetts, London, England, 1987.

[105] L.M. Pereira. Rational debugging in logic programming. In E. Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 203–210, London, 1986. Springer-Verlag, Berlin, New York, 1986.

[106] L.M. Pereira and M. Calejo. A framework for Prolog debugging. In R.A. Kowalski and K.A. Bowen [69], pages 481–495.

[107] D.A. Plaisted. An efficient bug location algorithm. In *Proceedings of the Second International Conference on Logic Programming*, pages 151–157, Upssala, July 1984. The MIT Press, Cambridge, Massachusetts, 1984.

[108] D. Plummer. Coda: An extended debugger for Prolog. In R.A. Kowalski and K.A. Bowen [69], pages 496–511.

[109] A.D. Polimeni and H.J. Straight. *Foundations of Discrete Mathematics.* Brooks and Cole, Pacific Grove, California, 1985.

[110] J.A. Purchase, M.G. Menelaou, and R.L. Winder. A critical survey of high level language debugging. RN/89/65, Department of Computer science, University College London, August 1989.

[111] T. Rajan. The design of animated tracing tools for novice programmers. Technical report no. 16, Human Cognition Research Laboratory, The Open University, 1988.

[112] Raghu Ramakrishnan. Magic: A spellbinding approach to logic programs. In Kowalski and Bowen [69], pages 140–159.

[113] G.R. Ruth. Intelligent program analysis. *Artificial Intelligence*, 7(1):65–85, 1976.

[114] D. Sahlin. Determinacy analysis for full Prolog. SIGPLAN NOTICES 26(9):23–30, 1991.

[115] T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34(1):227–240, 1984.

[116] D.A. Schmidt. *Denotational Semantics: a methodology for language development.* Allyn and Bacon, Inc., Boston, Massachusetts, London, 1986.

[117] D. De Schreye and M. Bruynooghe. An application of abstract interpretation in source level program transformation. In P. Deransart, B. Lorho, and J. Małuszynski, editors, *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming*, pages 35–57, Orléans, France, 1988. Springer-Verlag, Berlin, New York, 1989.

[118] R.L. Sedlmeyer and W.B. Thompson. Knowledge-based fault localization in debugging. *Software Engineering Notes*, 8(4):25–31, 1983.

[119] H. Seki. On the power of Alexander templates. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 150–159, Philadelphia, Pennsylvania, 1989. The ACM Press, New York, 1989.

[120] R.E. Seviora. Knowledge-based program debugging systems. *IEEE Software*, 4(3):21–32, 1987.

[121] E. Shapiro. Algorithmic program diagnosis. In *ACM Conference Record of the Ninth annual ACM Symposium on Principles of Programming Languages*, pages 299–308, Albuquerque, New Mexico, Jan. 25-27 1982. The ACM Press, New York, 1982.

[122] E. Shapiro. *Algorithmic Debugging*. The MIT Press, Cambridge, Massachusetts, 1983.

[123] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984.

[124] H. Søndergaard. An application of abstract interpretation of logic programs: occur check problem. In *Proceedings of the European symposium on programming*, Lecture Notes in Computer Science 213. Springer-Verlag, 1986.

[125] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, London, 1986.

[126] R. Sundararajan and J.S. Conery. An abstract interpretation scheme for groundness, freeness, and sharing analysis of logic programs. In R. Shyamasundar, editor, *Proceedings of 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 203–216, New Delhi, India, December 18-20, 1992. Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo, Hong Kong, Barcelona, Budapest, 1992.

[127] H. Takahashi and E. Shibayama. Preset - a debugging environment for Prolog. In *Proceedings of Fourth Conference on Logic Programming*, pages 90–99, Tokyo, 1985. Springer-Verlag, Berlin, New York, 1986.

[128] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proceedings of the Third International Conference on Logic Programming*, pages 84–98, London, 1986. Springer-Verlag, Berlin, New York, 1986.

[129] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5(1-4):285–308, 1955.

[130] A. Taylor. Removal of dereferencing and trailing in Prolog compilation. In Levi and Martelli [71], pages 48–60.

[131] J. Tiuryn. Type inference problems: A survey. In B. Roven, editor, *Proceedings of the Fifteenth International Symposium on Mathematical Foundations of Computer Science*, pages 105–120, Banska Bystrica,Czechoslovakia, 1990. Springer-Verlag, New York, 1990.

[132] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Artificial Intelligence*, 23(10):733–742, 1976.

[133] K. Verschaetse and D. De Schreye. Deriving termination proofs for logic programs, using abstract procedures. In Furukawa [47], pages 301–315.

[134] I. Vessey. Expertise in debugging computer programs: an analysis of the content of verbal protocols. *IEEE Transactions on Systems, Man and Cybernetics*, 16(5):621–637, 1986.

[135] A. Waern. An implementation technique for the abstract interpretation of Prolog. In Kowalski and Bowen [69], pages 700–710.

[136] D. H. D. Warren. Implementing Prolog - compiling predicate logic programs. Research Report 39 and 40, Department of Artificial Intelligence, The University of Edinburgh, 1977.

[137] D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.

[138] R. Warren, M. Hermenegildo, and S. K. Debray. On the practicality of global flow analysis of logic programs. In Kowalski and Bowen [69], pages 684–699.

[139] R.C. Waters. The programmer's apprentice: A session with KBemacs. *IEEE Transactions on Software Engineering*, 11(11):1296–1320, 1985.

[140] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[141] H. Wertz. Stereotyped program debugging: an aid for novice programmers. *International Journal of Man Machine Studies*, 16(4):379–392, 1982.

[142] J. Xu and D.S. Warren. A type inference system for Prolog. In R.A. Kowalski and K.A. Bowen [69], pages 604–619.

[143] S.Y. Yan. Declarative testing in logic programming. In *Proceedings of the Third Australian Software Engineering Conference*, pages 423–435, Canberra, 1987. The Australian Computer Society, Canberra, 1987.

[144] S.Y. Yan. Foundations of declarative debugging in arbitrary logic programming. *International Journal of Man Machine Studies*, 32(2):215–232, 1990.

[145] E. Yardeni and E. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–153, 1991.

[146] J. Zobel. Derivation of polymorphic types for Prolog programs. In J.-L. Lassez, editor, *Logic Programming: Proceedings of the Fourth International Conference*, pages 817–838 volume 2, pages 729–743, Melbourne, May 1987. The MIT Press, Cambridge, Massachusetts, London, England, 1987.

Index