Type Domains for Abstract Interpretation: A Critical Study

PER MILDNER

Uppsala University
Information Technology
Computing Science Department

Thesis for the Degree of Doctor of Philosophy

UPPSALA 1999
Type Domains for Abstract Interpretation: A Critical Study

Per Mildner

A Dissertation submitted in partial fulfilment of the requirements for the Degree of Doctor of Philosophy at Computing Science Department, Information Technology, Uppsala University.
(Dissertation for the Degree of Doctor of Philosophy in Computing Science presented at Uppsala University in 1999)

Abstract


Programming languages with dynamic typing such as Prolog do not require that the programmer declares the types of data or procedures. This flexibility, however, comes at a price. The lack of declarations makes it hard for a compiler to produce fast code since data must be assumed to be of any type.

Abstract interpretation is a technique to automatically infer conservative approximations of program properties such as the types of program variables and procedure arguments. The properties of interest are described using an abstract domain. The abstract domains for type analysis investigated in this thesis are based on deterministic term grammars but differ in the representation of the term grammars and in the widening used, that is, the method used to ensure that the analysis terminates.

One proposed representation of term grammars, type graphs, are shown to be exponentially larger than the term grammars they represent not only in the worst case but also in practice. One previously proposed widening for type graphs is shown to be impractically expensive and also no more precise than other more efficient methods. A novel implementation technique is proposed for another type graph widening that appears in the literature. The precision using this implementation technique is shown to be better than for the other investigated widenings. The performance, however, suffers from the problems inherent in the type graph representation unless some crucial modifications to the analyzer are made.

To overcome the problems with the type graph representation a more compact representation of term grammars is proposed. A number of widenings using this representation are investigated and shown not to suffer from problems with huge term grammar representations. The precision obtained using these widenings are as good or better than the precision obtained using the original widening proposed for type graphs.

Finally, two sets of benchmarks that have been used in the literature to investigate analysis methods are considered. It is shown that these benchmarks will give unrealistically good absolute precision and one set of benchmarks, the GAIA benchmarks, is shown to have further deficiencies that make it completely unsuitable for benchmarking purposes.

*Per Mildner, Computing Science Department, Information Technology, Uppsala University, Box 311, SE–751 05 Uppsala, Sweden.*

© Per Mildner 1999

ISSN 0283–359X
Printed by Nina Tryckeri HB, Uppsala 1999
To Ina, Arvid and Hedvig
ACKNOWLEDGMENTS

Thanks to Jonas Barklund, my advisor during the first years. A special thanks for letting me pursue interests somewhat outside the original plan . . .

Thanks to Håkan Millroth, not only for taking over after Jonas, but also for encouragement and support over the years.

Last, but not least, in the line of advisors, Roland Bol. Thanks for providing the necessary cattle prod. Indeed, how hard can it be, when the table of contents is completely you just have to fill in the blanks.

A special thanks also to Sven-Olof Nyström for commenting on numerous drafts of this manuscript.

Finally, thanks to all the other people that made a long life at the department fun. Especially Thomas Lindgren for lots of interesting discussion from instruction scheduling to abstract interpretation and other things that make life interesting. Johan Bevenyr too and the other guys involved in the various prolog projects throughout the years.

In later years, Erik Johansson, Christer Jonsson and Greger Ottoson too contributed to the good atmosphere.

This thesis would not have been possible without the support from my wife, Ina.

Finally a big thanks to those that I may have forgotten.
CONTENTS

1 INTRODUCTION 3
  1.1 Types in Logic Programming ................................. 5

2 ABSTRACT INTERPRETATION 7
  2.1 Example: A Simple Abstract Domain ......................... 7
    2.1.1 Conditional Expressions ............................... 10
    2.1.2 Loops and Fixpoints ................................. 11
    2.1.3 Infinite Chains and Widening .......................... 14
    2.1.4 Summary of Abstract Domain Operations ................. 19
  2.2 Abstract Interpretation of Prolog ......................... 20
    2.2.1 Preliminaries ...................................... 21
    2.2.2 Analysis ........................................... 22
  2.3 Abstract Substitutions .................................... 24
    2.3.1 Variable Aliasing ................................... 26
    2.3.2 Type Component .................................... 27
    2.3.3 Same Value Component ................................ 28
    2.3.4 Abstract Substitution Operations ..................... 33
    2.3.5 Abstract Unification ................................ 38

3 TERM GRAMMARS 40
  3.1 Regular Term Grammars ..................................... 41
  3.2 Operations on Term Grammars ............................... 46
    3.2.1 Union ............................................. 47
    3.2.2 Deterministic Union ................................ 47
    3.2.3 Containment ....................................... 49
    3.2.4 Intersection ...................................... 50
  3.3 Representing Term Grammars ............................... 51
    3.3.1 Preliminaries ...................................... 52
    3.3.2 Grammar Graphs ..................................... 53
    3.3.3 Type Graphs ....................................... 56
    3.3.4 Regular Unary Logic Programs ....................... 60
5.4.6 Summary and Conclusions ........................................ 145  
5.5 Summary and Conclusions ........................................ 147  

6 Methods using Minimized Grammar Graphs ........................................ 150  
6.1 General Issues ...................................................................... 151  
6.1.1 Domain Operations for Minimized Grammar Graphs .................. 151  
6.1.2 Adapting Type Graph Widenings ....................................... 152  
6.2 $\sqsubseteq$-Minimization .................................................. 153  
6.2.1 $\sqsubseteq$-Minimization Is Not a Widening ......................... 156  
6.2.2 Performance .......................................................... 157  
6.2.3 Precision .............................................................. 159  
6.2.4 A Method by Saglam & Gallagher ..................................... 159  
6.3 Type Jungle ........................................................................ 161  
6.4 Janssens' Widening Revisited .............................................. 162  
6.4.1 Using Grammar Graphs as Input ....................................... 162  
6.4.2 $\sqsubseteq$-Minimization .................................................. 163  
6.5 Spanning Tree Widening .................................................... 163  
6.5.1 Definition ............................................................... 165  
6.5.2 Variations of the Spanning Tree Widening .......................... 166  
6.5.3 Performance ............................................................ 166  
6.5.4 Precision .............................................................. 170  
6.5.5 Relation to Gallagher & de Waal ................................. 170  
6.5.6 Summary and Conclusion .............................................. 172  
6.6 Topological Clash Revisited .............................................. 173  
6.6.1 Direct Construction of Topological Clash Widening ............. 173  
6.6.2 Discussion .............................................................. 180  

7 Evaluation ........................................................................... 182  
7.1 Functors Widening ........................................................... 183  
7.1.1 Comparison with Type Graph domain ............................... 183  
7.1.2 Impact of Input-Data on Precision ................................... 183  
7.1.3 An Unintuitive Result .................................................... 186  
7.2 Type-Jungle Widening ...................................................... 187  
7.3 Janssens' Widening ........................................................... 189  
7.3.1 Janssens' widening with $\sqsubseteq$-Minimization .................... 192  
7.4 Topological Clash Widening .............................................. 194  
7.4.1 Original Topological Clash Widening ............................... 194  
7.4.2 Direct Construction ..................................................... 199  
7.5 Spanning Tree Widening .................................................... 202  
7.6 $\sqsubseteq$-Minimization Widening .......................................... 205  
7.7 Precision Comparison ...................................................... 206  

8 Conclusions ........................................................................... 208  

A Precision on Larger Benchmarks ........................................... 214
A.1 Overview of Tables ................................. 214

B Correctness of the Topological Clash Widening 223
B.1 Stationarity .................................. 223

Bibliography ................................. 229

Index ........................................ 235
During the writing of this thesis I repeatedly got the following email

Increase your personal prestige and money earning power through an advanced university degree.

Eminent, non-accredited universities will award you a degree for only $200.

Degree granted based on your present knowledge and experience. No further effort necessary on your part.

Just a short phone call is all that is required for a BA, MA, MBA, or PhD diploma in the field of your choice.

For details, call XXX-XXX-XXXX

There were days (and nights) when this offer was very tempting.
This thesis investigates practical aspects of type-analysis for Prolog. Type analysis is the problem of determining, at compile time, the type of the procedure arguments that may occur at runtime. This information will enable a compiler to generate better code.

For languages with explicit type declarations and static typing, such as C, ML and Mercury, the type information provided by the programmer gives the compiler a lot of information about the run-time values. For languages with dynamic typing, such as Lisp, Erlang, and Prolog, compilers have typically had to assume that any kind of value can occur anywhere, thus making it harder to provide efficient implementations of these languages.

However, even if any kind of value is allowed everywhere, programs are typically written with some specific kind of data in mind. For languages with dynamic typing it is possible to infer, automatically or semiautomatically, type-like information. Many such analysis methods have been proposed, although fewer have been implemented and few have been put to large-scale use.

Almost any conceivable compiler optimization for Prolog can benefit from type information. Simple uses of such information include simplified unification and control. More advanced uses include compile-time garbage collection and determinacy detection.

There are also other application areas where the results from type analysis can be useful. During verification and debugging it can provide information to a programmer that are not immediately apparent from the program text. The use of type analysis for debugging is not the focus of this thesis but the analyzer have already proven useful in this respect. The analyzer uncovered problems in several popular benchmark programs.

Abstract interpretation [10] is a general methodology for performing various kinds of program analyses. An abstract interpreter performs the program analysis by simulating the execution of the analyzed program. The simula-
tion uses descriptions of the values that would occur at runtime. Elements from an abstract domain are used to represent descriptions. The abstract interpreter can be parameterized by an abstract domain and associated operations.

The methods investigated in this thesis are formulated as type analysis of Prolog programs but some of the methods may be applicable also in other contexts. In particular it should be possible to use the abstract domains also for type analysis for pure, dynamically typed, logic or functional languages, with little or no modification.

The abstract domains for type analysis investigated in this thesis are all based on deterministic term grammars but differ in the representation of the term grammars and in the widening used, that is, the method used to ensure that the analysis terminates.

The characteristics of the abstract domains and their widenings will be studied using a simple analyzer based on abstract interpretation. The analyzer uses a fixpoint iteration to obtain a conservative approximation of the procedure arguments. An interesting and often overlooked issue is the interactions between the analyzer, the abstract domain, and the widening. For some of the abstract domains and widenings considered in this thesis common optimizations to the analyzer may in fact be counterproductive since they can adversely affect the costs involving the abstract domain.

There are two, often opposing, characteristics of an abstract domain. The first is the efficiency of the resulting analysis. Efficiency is a prerequisite if the analyzer should ever be made a part of a compiler or other development tool. If efficiency cannot be guaranteed in general then the analysis must at least be efficient for important classes of programs. Of particular importance for a type analysis is how efficient it is when inferring the types of recursive data structures manipulated by an analyzed program. Some of the methods proposed in the literature will be shown to be infeasibly expensive when inferring recursive types.

The second important characteristic of an abstract domain and the accompanying widening is the quality of the analysis results. The different methods are therefore implemented and the precision and performance of the obtained analyses compared using a number of realistic benchmark programs of substantial size. In addition to a number of new widenings this comparison includes the performance of two methods from the literature that have not previously been evaluated on realistic benchmarks.

This introduction concludes with a brief discussion of other notions of types in logic programming. The rest of this dissertation is organized as follows: Chapter 2 gives an introduction to abstract interpretation in general and as it relates to type analysis of Prolog. Chapter 3 describes term grammars
and the operations needed to use term grammars as an abstract domain. Chapter 3 concludes with a description of several concrete representations of term grammars and their operations.

Chapter 4 contains all the machinery needed to evaluate the abstract domains. This includes the abstract interpreter and various iteration strategies that affect analysis efficiency. It also contains a section on measuring performance and precision. The chapter concludes with a presentation of the benchmarks that will be used, and perhaps more interestingly, the reasons some popular benchmarks will not be used.

Chapter 5 describes methods that use type graphs to represent term grammars. The main part of this chapter describe the implementation of two widenings on type graphs that have been proposed in the literature [32, 30]. In this chapter synthetic benchmarks are used to highlight problems inherent in this representation of term grammars and in the widenings.

Chapter 6 describes methods that use minimized grammar graphs, a particularly compact representation of term grammars devised to overcome the problems with type graphs. This representation is often exponentially smaller than type graphs when representing the same term grammar. This chapter describes widenings that operate directly on grammar graphs and also methods that use grammar graphs to improve the efficiency of the type graph methods.

Chapter 7 evaluates precision and performance of the various methods on larger, realistic benchmarks.

Chapter 8 contains a summary of the dissertation and my contributions and a discussion of possible future work.

1.1 TYPES IN LOGIC PROGRAMMING

The notion of type considered in this thesis are conservative approximations of the terms that can occur as arguments to predicates (that is, procedures) at runtime. Such information makes it possible for a compiler to generate more efficient code. The intent is to be able to analyze existing Prolog program, without the need to add type declarations or similar constructs to the language. The types are descriptive that is, they describe the terms that can occur at runtime.

This is opposed to prescriptive types that are used to provide, often through programmer supplied type declarations, information about the allowable predicate arguments or results. It is possible to extend Prolog or other logic programming languages with type declarations and methods for verifying that a program complies with these declarations can be devised [57, 42, 9].
However, while static type checking as enabled by these techniques has many merits, the resulting language is not Prolog.

Another notion of types, sometimes called semantic types [27], are concerned with approximating the (declarative) meaning of a logic program. The meaning of a predicate $p$ is typically regarded as the set of (ground) Herbrand terms $\overline{t}$ for which $p\overline{t}$ may succeed, that is, an approximation of the success set of $p$. A typical approach is to use a bottom up analysis to obtain a fixpoint using an abstract version of the immediate consequence operator $T_P$ [38]. Another approach is based on set constraints where a constraint system is constructed that relates the sets of possible values of the program variables. This constraint system is then solved to obtain an approximation of the possible values of the program variables and typically also the possible success sets of the predicates [40, 57].

However, success sets does not provide information about the arguments with which predicates are called. An important special case, however, is when the success set can be shown to be empty. In this case the predicate is known to fail or be non-terminating and, provided the predicate has no side effects, any calls to the predicate can be removed by the compiler [54]. It should be noted that it is possible use a bottom up analysis to simulate a top down analysis by transforming the program using query answer or magic sets transformation and then analyze the transformed program to infer information about the original program, including information about calls [17].
Abstract interpretation [10] is a methodology to construct program analysis algorithms. Starting with a concrete semantics, or execution model, an abstract interpretation is a way to simulate the concrete behavior of a program using an abstract semantics that captures only the properties of interest.

The abstract interpretation should be *sound*, that is, it must describe all possible behaviors of the analyzed program. However, it does not have to, and in general cannot, precisely predict the behavior of the analyzed program, but rather a *conservative* or approximate result will suffice. This means that the analysis will in general not only predict everything that could possibly happen in the analyzed program but also some things that will in fact never happen. One of the challenges is therefore to design abstract semantics that gives as precise analysis results as possible while still ensuring that the analysis is sound and reasonably efficient. The properties of interest are typically uncomputable in general so the first efficiency requirement is that the analysis actually terminates for all inputs.

### 2.1 Example: A Simple Abstract Domain

A classical example of an abstract interpretation is the rule of signs, that is, that in some cases the sign of an expression can be known with knowledge only of the signs of the involved integers. I will use this domain and a simple imperative language to introduce, by example, the necessary concepts of abstract interpretation.

Consider a concrete expression (or “program”) \((x \times x) + (y \times y)\). With the operations + and \(\times\) interpreted as ordinary addition and multiplication respectively, the value of the expression is 25 if \(x = 3\) and \(y = -4\). Letting \(Pos\) stand for any positive integer and \(Neg\) for any negative integer we can reinterpret addition and multiplication according to the rule of signs:
The appropriate descriptor for 3 is \textit{Pos} and for \(-4\) it is \textit{Neg}. The abstract interpretation of the above expression would then yield \((\textit{Pos} \times \textit{Pos}) \oplus (\textit{Neg} \times \textit{Neg})\), and using the abstract versions of \(+\) and \(\times\) we can conclude that the result is \textit{Pos}, correctly inferring the fact that \(25\), the result of the original concrete expression, is in fact positive.

To ensure that the results from the analysis is valid, that is, sound, the abstract version, such as \(\hat{\oplus}\), of a concrete operation \((\oplus)\) must be sound. This means that if the result of the concrete operation applied to some concrete arguments \(x, y\), say, is \(z\), then the result \(\hat{\oplus}\) of applying the abstract operation on abstract versions \((\hat{x}, \hat{y})\) of the arguments must be a description of the corresponding concrete value \(z\). For \(\hat{\oplus}\) this means that it would be unsound to define \(\textit{Pos} \hat{\oplus} \textit{Neg} = \textit{Pos}\). Even though this would be correct in some cases such as \(5 + (-2) = 3\) (since \(5\) and \(3\) are \textit{Pos}-itive and \(-2\) is \textit{Neg}-ative), it is not correct in general since, for example, \(5 + (-8) = -3\) which correspond to \(\textit{Pos} \hat{\oplus} \textit{Neg} = \textit{Neg}\).

What should be done with a case such as \(x + y\) with \(x\) positive and \(y\) negative? In this case the definition of abstract addition gives no information and the sign of the result depends on the relative magnitude of \(x\) and \(y\), a property not captured by our abstract domain of signs. To handle such cases when nothing is known about the concrete values the abstract domain is always extended with an element \(\top\) (called "top") denoting all values, in this case all integers. This gives the following additional rules,

<table>
<thead>
<tr>
<th>(\top) (\hat{\oplus}) (\top) (\widehat{\textit{Pos}}) (\textit{Pos}) (\textit{Neg}) (\textit{Pos})</th>
<th>(\top) (\hat{\times}) (\top) (\textit{Pos}) (\textit{Neg}) (\textit{Neg})</th>
<th>(\top) (\hat{\times}) (\textit{Pos}) (\textit{Neg}) (\textit{Neg})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\top) (\hat{\oplus}) (\top) (\top) (\top) (\top)</td>
<td>(\top) (\hat{\times}) (\top) (\top) (\top)</td>
<td>(\top) (\hat{\times}) (\textit{Neg}) (\top) (\top)</td>
</tr>
</tbody>
</table>

With the addition of a \(\top\) we suddenly have a choice when selecting a descriptor for a concrete value such as 3, either \textit{Pos} since 3 is positive, or \(\top\) since 3 clearly is among all integers. On the other hand \textit{Pos} conveys more information about a value than \(\top\), that is, \textit{Pos} is a more precise description
than \( \top \) and we would like to use the most precise abstract value whenever possible.

The concretization function \( \gamma(a) \) denotes the set of concrete values described by the abstract descriptor \( a \), in particular the concretization of the special “least precise” abstract value \( \top \) is the whole concrete domain. For the example domain above we would have \( \gamma(\text{Pos}) \) the set of positive integers and \( \gamma(\top) \) the whole concrete domain, that is, the set of all integers.

By using the concretization function we can make the notion of “more precise” more precise; intuitively an abstract descriptor \( a_1 \) is more precise than \( a_2 \) if and only if \( \gamma(a_1) \subseteq \gamma(a_2) \), written using the containment operator as \( a_1 \subseteq a_2 \), however, soundness of the analysis only requires the weaker condition \( a_1 \subseteq a_2 \implies \gamma(a_1) \subseteq \gamma(a_2) \). This containment or precision order is only a partial relation, since not all descriptors are comparable, for example, neither one of \( \text{Pos} \) and \( \text{Neg} \) is more precise than the other. In general \( \subseteq \) is not even a partial relation but only a pre-order as there may be two distinct abstract values \( a_1, a_2 \) such that \( a_1 \subseteq a_2 \land a_2 \subseteq a_1 \). For this reason it makes sense to compare abstract descriptors not by equality but by equivalence (\( \equiv \)), where \( a_1 \equiv a_2 \iff a_1 \subseteq a_2 \land a_2 \subseteq a_1 \). For all descriptors \( a, a \sqsubseteq a \) and \( a \sqsubseteq \top \). If \( a \sqsubseteq b \) but \( b \not\sqsubseteq a \) then \( a \) is said to be strictly more precise than \( b \), written \( a \sqsubset b \). Note that \( \subseteq \) is a partial order on the equivalence classes induced by \( \equiv \).

Continuing the example, what about the case \( (x+y) \) when \( y \) is 0? Since 0 is neither positive not negative we have no choice but to use \( \top \) to describe it, and as a consequence lose all information about the value of the expression.

Another possibility is to redefine \( \text{Pos} \) to mean non-negative integer and \( \text{Neg} \) to mean non-positive integer and to keep the above rules for abstract addition and multiplication. Unfortunately we now have a non-trivial choice when selecting an abstract value to describe 0. We can choose either \( \text{Pos} \) or \( \text{Neg} \) and since neither is more precise than the other we are not helped by \( \subseteq \). Instead the appropriate choice depends on the context since, when \( y \) is 0 for \( (x+y) \), if \( x \) is positive then approximating \( y \) with \( \text{Pos} \) would give \( (\text{Pos} + \text{Pos}) = \text{Pos} \) whereas approximating \( y \) with \( \text{Neg} \) would give the less precise \( (\text{Pos} + \text{Neg}) = \top \), with \( x \) negative the reverse would hold. To avoid these kinds of choices it is preferable to define the abstract domain such that there is always a unique best choice. For rule of signs a better solution is then to keep the original definitions of \( \text{Pos} \) and \( \text{Neg} \) and to introduce a new abstract value \( \text{Zero} \) denoting the integer 0 together with additional rules such as \( \text{Zero} \cdot \text{Pos} = \text{Pos} \) and \( \text{Zero} \cdot \top = \text{Zero} \).

The abstraction function \( \alpha \) gives the (usually unique and most precise) description of a concrete value or a set of concrete values. It is closely linked to the concretization function \( \gamma \). Intuitively we would like to have that the composition of the abstraction and the concretization function (or
Abstract Interpretation

vice versa is the identity function, that is, \( \alpha(\gamma(a)) = a \) and \( \gamma(\alpha(S)) = S \) for any set \( S \) of concrete values and any abstract descriptor \( a \). In practice this is too strong and for soundness \( S \subseteq \gamma(\alpha(S)) \) is sufficient.

With \( \alpha \) and \( \gamma \) we can now make precise what constitutes a sound approximation of a concrete operation, such as \( \times \). An abstract version \( \hat{f} \) of some concrete operation \( f \) must be **locally consistent**, \( f(S) \subseteq \gamma(\hat{f}(\alpha(S))) \), that is, all outcomes of the concrete operation must be included in the concretization of the outcome of the abstract operation.

### 2.1.1 Conditional Expressions

Consider now a more complex example involving a conditional expression where the value depends on some test whose outcome is not known.

\[
\text{if } (\text{test}) \text{ then } (x + y) \text{ else } (x \times y)
\]

Suppose we know that \( x \) is described by the abstract value \( Pos \) and \( y \) by \( Zero \). The abstract interpretation of the first conditional branch would yield \((Pos \not< Zero)\), i.e., \( Pos \), whereas the second conditional branch would yield \((Pos \not> Zero)\), i.e., \( Zero \). This situation is very common since, in general, static analysis will not have enough information to decide the outcome of tests. To be on the safe side the analysis will have to assume that either branch can be taken, that is, it need to somehow express the fact that the result is either \( Pos \) or \( Zero \). To this end an upper bound operation \( \sqcup \) is defined that computes a safe approximation of two abstract values, that is for two abstract values \( a_1, a_2 \) we should have \( a_1 \sqsubseteq a_1 \sqcup a_2 \) as well as \( a_2 \sqsubseteq a_1 \sqcup a_2 \), thus, we should have \( \gamma(a_1) \sqcup \gamma(a_2) \subseteq \gamma(a_1 \sqcup a_2) \). To preserve as much precision as possible \( \sqcup \) is often defined to give the most precise upper bound, or the least upper bound (LUB). If the two arguments are comparable by \( \sqsubseteq \) then the larger of the two is their least upper bound.

For the abstract domain in the example only the top element, \( \top \), is less precise than both \( Pos \) and \( Zero \). The top element is always an upper bound but depending on the domain and the abstract values there may be other more precise choices. We can further extend our domain of signs to include the abstract values \( NonNeg \) and \( NonPos \) describing the non-negative and non-positive integers respectively. The least upper bound for the conditional expression above would then be \( NonNeg \).

Now consider the following conditional expression and what can be said about \( x \) in the first conditional branch \( expr_1 \) if analysis has determined that \( x \) is \( NonNeg \) and \( y \) is \( NonPos \).

\[
\text{if } x = y \text{ then } expr_1 \text{ else } expr_2
\]
In this case the analysis cannot know if the equality test will be true or false, however, the analysis can retain some information from the test to be used in the analysis of the branches since it knows that if the first branch is taken then the test was true. In particular in the example we can conclude that within \( \text{expr}_1 \) the value of \( x \), by being equal to \( y \), must have a value that is described by both \textit{NonNeg} and \textit{NonPos}. The only concrete value for which this is true is 0 and the most precise approximation of the value of \( x \) within \( \text{expr}_1 \) is thus \textit{Zero}.

To obtain a safe approximation of the concrete values that satisfies two abstract descriptions we define an abstract intersection operator \( \cap \), intuitively it is the abstract counterpart of intersection of two sets of concrete values. The safeness of \( \cap \) require that \( \gamma(a_1) \cap \gamma(a_2) \subseteq \gamma(a_1 \cap a_2) \). Similarly to \( \sqcap \) the most precise \( a \) satisfying this condition is usually intended and if the two arguments are comparable then the least of them is the most precise such value. For some domains \( \cap \) coincide with the greatest lower bound \( \text{glb} \), defined so that \( a = \text{glb}(a_1, a_2) \) is the unique least precise \( a \) such that \( a \subseteq a_1 \) and \( a \subseteq a_2 \).

As a final example consider the above example when \( x \) and \( y \) are instead described by \textit{Pos} and \textit{Neg} respectively. What would then be a suitable approximation for \( x \) within \( \text{expr}_1 \), that is what should be used for \( \text{Pos} \cap \text{Neg} \)? The positive and the negative integers have no values in common so an appropriate abstract value would be one that as precisely as possible describes the empty set of values. In concrete terms it means that there is no concrete value possible for \( x \) within \( \text{expr}_1 \) and that the analysis would be able to conclude that \( \text{expr}_1 \) can never be reached, and thus that the equality test never would come out true.

Typically all abstract domains are extended with such a \textit{bottom} element \( \perp \) defined such that \( \perp \subseteq a \) for any abstract domain element \( a \), and \( \gamma(\perp) = \emptyset \).

In concrete terms \( \perp \) means that the execution cannot reach this state.

### 2.1.2 Loops and Fixpoints

1 \( x := 0 \)
2 do
3 \( \langle \text{name1} \rangle \)
4 \( x := x \times 2 \)
5 while \( \langle \text{name2} \rangle \)
6 \( \langle \text{name3} \rangle \)

Figure 2.1: A simple loop

Iteration constructs such as loops and recursion in the language make the situation much more interesting. Consider the (concrete) value of the variable \( x \) in the program fragment of Figure 2.1. Clearly \( x \) is 0 immediately
after the assignment at line 1, a fact easily captured by the abstract domain of signs introduced above. Following the execution forward into the loop we can see that \( x \) will still be 0 when we reach \( stmt_1 \) the first time (we assume that the value of \( x \) may be used but not changed in \( stmt_1 \)). At line 4 \( x \) is incremented giving \( x \) the value 2 when the loop test is performed the first time. Suppose the loop test is true so that the loop should iterate once again, the execution now jumps back to line 2 with \( x \) having the value 2, then \( stmt_1 \) is evaluated again and so on. Eventually the loop test becomes false, the loop is exited and \( stmt_2 \) is reached, where \( x \) presumably is used.

The first observation we can make is that at a particular point in the program \( x \) may have more than one concrete value. For example, at line 3 the variable \( x \) could take on any non-negative even integer (for now we ignore issues such as maximum range of the integer representation used). Thus, in general, sets (or \textit{collections}) of concrete values, a \textit{collecting semantics}, is a better way to describe the concrete execution.

The second observation we can make is that with looping constructs in the language the concrete semantics would naturally be described as a (smallest) fixpoint of some function that describes how the program transitions from one state to another.

How does these issues affect the abstract interpretation? First of all the abstract descriptions too would have to be descriptions of sets of concrete values, in our example they already do and this was necessary already to handle conditional execution. Secondly the abstract execution (abstract semantics) would typically also be defined as a fixpoint when iteration constructs are present in the language. Furthermore, the concrete semantics may not be computable, whereas the abstract semantics, would have to be (efficiently) computable, and the abstract values finitely representable, in order to give a terminating analysis.

<table>
<thead>
<tr>
<th>( l )</th>
<th>( x' )</th>
<th>Code</th>
<th>( x_l )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>( x := 0 )</td>
<td>( x_1 \uparrow \alpha(0) )</td>
</tr>
<tr>
<td>2</td>
<td>( x^2 \uparrow x_1, x^2 \uparrow x_5 )</td>
<td>( do )</td>
<td>( x_2 \uparrow x^2 )</td>
</tr>
<tr>
<td>3</td>
<td>( x^3 \uparrow x_2 )</td>
<td>( stmt_1 )</td>
<td>( x_3 \uparrow x^3 )</td>
</tr>
<tr>
<td>4</td>
<td>( x^4 \uparrow x_3 )</td>
<td>( x := x+2 )</td>
<td>( x_4 \uparrow x^4+\alpha(2) )</td>
</tr>
<tr>
<td>5</td>
<td>( x^5 \uparrow x_4 )</td>
<td>( \text{while} \ test )</td>
<td>( x_5 \uparrow x^5 )</td>
</tr>
<tr>
<td>6</td>
<td>( x^6 \uparrow x_5 )</td>
<td>( stmt_2 )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.2: Loop with semantic equations for beginning \((x')\) and end \((x_l)\) of line \( l \)

In Figure 2.2 the loop from Figure 2.1 is repeated together with equations that describe the abstract value of \( x \) before \((x')\) and after \((x_l)\) each line \( l \) and the relations between these values. As can be seen there are circular dependencies through the loop, so for example \( x_2 \) depends on both \( x_1 \) and 
functions $x_5$ and the latter, in turn, depends indirectly (via $x_4$, $x^4_2$, $x^3$) on $x_2$. Of particular interest are the equations for $x^2_2$ corresponding to a point where the value of $x$ can come from two different directions, and the equation for $x_4$ that says that $x_4$ should safely approximate the result of the (abstract) addition.

There are several possible ways to obtain a, preferably most precise, solution to this kind of equations (or a safe approximation of a solution). Only algorithms based on fixpoint iteration will be considered here.

A typical fixpoint iteration algorithm would start with an unsafe approximation (such as assuming all variables can be described by $\perp$, or equivalently that no parts of the program will be executed), then update the information corresponding to an equation $x \supseteq \text{rhs}$, as new approximations become available for the variables used in the right-hand side ($\text{rhs}$), by computing an upper bound of the previous approximation and the new value of the right-hand side. So for example, a new approximation for $x_5$ would affect the right-hand sides of the equations $x^3 \supseteq x_2$ and $x^2 \supseteq x_5$, so the fixpoint iteration algorithm would perform $x^3 \leftarrow x^3 \cup x_5$ and $x^2 \leftarrow x^2 \cup x_5$. This in turn may cause a change to $x^6$ and $x^5$ that must be propagated. Eventually there will be no more changes and the solution thus obtained is a safe approximation of a most precise solution.

An alternative way to obtain a (approximation of a) fixpoint is to simply repeatedly recompute and incorporate all right-hand sides in some order until no changes occur. While simple this method has the disadvantage that it may perform a lot of unnecessary work. For this reason I will consider methods that try to recompute only the equations that could be affected.

Going back to the value of $x$ in the example in Figure 2.2. The initial approximation would then be $x^1_1 = \perp$ for all lines $l$. Starting at line 1 the analysis would perform (the abstract version of) the assignment, concluding that the most precise description of $x_1$ is $\text{Zero}$. Now the input to the equation $x^2 \supseteq x_1$ changed ($x_1$ used to be $\perp$) and $x^2$ is set to $x^2 \cup x_1 = \perp \cup \text{Zero} = \text{Zero}$. This change to $x^2$ propagates through line 2, which does not modify $x$ so $x_2$ and $x_3$ becomes $\text{Zero}$ as well, then $x_3 = \text{Zero}$ is used for the first analysis of $\text{stmt}_1$, after which $x^2$ and $x_4$ too changes from $\perp$ to $\text{Zero}$. Now the abstract version of the incrementation is reached and the right hand side of the assignment, abstract addition, is performed using the value of $x$ at that point ($x_4$) and an approximation of the constant 2, that is $\text{Zero} + 2 = \text{Zero} + \text{Pos}$, yielding $\text{Pos}$. Abstractly performing the assignment finally sets $x_4$ to $x_4 \cup \text{Pos} = \perp \cup \text{Pos} = \text{Pos}$ which makes it necessary to update $x^5$ and then $x_5$.

At line 5 the algorithm now have a choice since the analysis must propagate the new approximation of $x_5$ to both $x^2$ and $x^6$. Assuming the analysis
proceeds to line 6 the analysis of $stmt_2$ would use the approximation $Pos$ as the value of $x^0$.

Analysis now continues by re-visiting line 2 and in particular the equation $x^2 \equiv x_5$. In this case we already have $x^2 = Zero$ obtained the first time line 2 was visited from line 1 so the new approximation for $x^2$ becomes $x^2 \sqcup x_5 = Zero \sqcup Pos = NonNeg$. This change propagates via $x_2$, $x^3$ into $stmt_1$ which is reanalyzed with $NonNeg$ for $x^2$. Line 4 is reached again, evaluating the right hand side of the assignment $NonNeg + \alpha(2) = NonNeg + Pos = Pos$ and finally performing the assignment concluding that the new abstract value of $x_4$ is $Pos$, that is, the abstract value of $x_4$ did not change, all parts of the program (only line 5) that could depend on the value of $x_4$ have already been analyzed with this information available.

The analysis terminates and concludes among other things that $x$ at line 5 is positive (an over-approximation since $x$ can only be even). This is also the most precise result possible with this domain. Unfortunately, for other domains, the most precise result is not always obtainable, in particular the most precise sound approximation will not be obtained for the domains used later on for type analysis of Prolog.

Why does the analysis terminate? The analysis continues as long as there are changes that need to be propagated. In the example the only changes are to the approximation of a variable and this can only happen a finite number of times; approximations starts at $\bot$ and every time the approximation for some variable changes it is to a strictly larger (that is less precise) approximation and since, for the domain of signs, there is only a finite number of abstract values in any sequence of increasing approximations, the increase will eventually stop at $\top$ if not before. If any such chain of increasing (or ascending, towards the “top” $\top$) approximations is finite the domain has the ascending chain condition, in fact the sign domain has the even stronger property of finite height.

### 2.1.3 Infinite Chains and Widening

Unfortunately some useful abstract domains do not have finite height, and some have finite but very large height.

Note that infinite domain size in itself need not be a problem. If we extend the domain of signs not only with the abstract descriptor $Zero$ with concretization $\{0\}$ but also with infinitely many elements $One$, $Two$, $Three$, ..., standing for $\{1\}$, $\{2\}$, $\{3\}$, ..., then the domain would be infinite but the maximum length of any (strictly) increasing chain would still be five, an example being $\bot \sqsubseteq One \sqsubseteq Pos \sqsubseteq NonNeg \sqsubseteq \top$. 
\[
\begin{align*}
\bot \cup b &= b \\
a \cup \bot &= a \\
[l, h] \cup [l', h'] &= [\min(l, l'), \max(h, h')] \\
a \subseteq b &\iff a = \bot \\
&\lor \\
&\quad a = [l, h] \land b = [l', h'] \land l' \leq l \land h \leq h' \\
\bot \hat{\cup} b &= \bot \\
a \hat{\cup} \bot &= \bot \\
[l, h] \hat{\cup} [l', h'] &= [l + l', h + h'] \\
\text{where } x + \infty &= \infty + x = \infty \\
\text{and } x + -\infty &= -\infty + x = -\infty
\end{align*}
\]

Figure 2.3: Interval Domain Operations.

If there are infinite chains in the domain, however, then an approximation can be updated infinitely many times to a strictly larger abstract value and the analysis might not terminate.

Consider an alternative domain to describe sets of integers, a domain of intervals \([l, h]\) such that \(l \leq h, l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{\infty\}\), where an abstract value \([l, h]\) describes the set \(\{j \mid l \leq j, j \leq h, j \in \mathbb{Z}\}\). This domain is not only infinite but there are also infinite ascending chains such as \([3, 5] \subseteq [3, 6] \subseteq [3, 7] \ldots\). In Figure 2.3 some of the usual domain operations as well as abstract addition is defined.

The domain of intervals is strictly more precise than the domain of signs, for example \(\text{Zero}, \text{Neg}, \text{Pos}, \top\) corresponds to the elements \([0, 0]\), \([-\infty, -1]\), \([1, \infty]\), and \([-\infty, \infty]\) (= \(\mathbb{Z}\)) respectively whereas an interval such as \([-3, 2]\) cannot be described except by \(\top\) in the domain of signs. We will use the names from the original sign domain, e.g., \(\text{Pos}\) instead of \([1, \infty]\), when convenient.

What would happen if the domain of intervals were used to analyze the program of Figure 2.2? After line 1 \(x_1\) would be abstracted as \([0, 0]\), that is \(\text{Zero}\), as before. Analysis proceeds with line 2, 3 and 4 as before with \(x_2 \ldots x_4\) all becoming \([0, 0]\). The first difference with the original domain of signs occurs when performing the abstract counterpart of incrementation. Assuming the abstract version of addition is as precise as possible given this more precise domain, after the assignment statement the value of \(x_4\) becomes \(x_4 \hat{\cup} (x_4 + \alpha(2)) = \bot \cup ([0, 0] \hat{\cup} [2, 2]) = [2, 2]\). This value now propagates through line 5 and updates \(x_5\) from \(\bot\) to \([2, 2]\). As before all paths where this value of \(x\) could reach must be followed since they were
analyzed under the, now incorrect, assumption that the \( \bot \) is a conservative approximation of the value of \( x_5 \). Assuming the analyzer reanalyzes line 2, \( x^2 \) should become \( x^2 \sqcup x_5 \), that is \([0,0] \sqcup [2,2] = [0,2] \). Propagating this through the loop will eventually reach line 4 setting \( x_4 \) to \( x_4 \sqcup (x^4 + \alpha(2)) = [0,0] \sqcup ([0,2] \int [2,2]) = [0,0] \sqcup [2,4] = [0,4] \). This will go on forever using successively larger intervals as abstract descriptors.

As can be seen from this example the problem is caused by the circular dependencies introduced by the loop. The increasing chain is created through a series of applications of \( \sqcup \) at line 2, \((\bot \sqcup x_1) \sqcup x_5 \sqcup x_5 \ldots \), however, nothing prevents us from using a less precise upper bound in these situations, an upper bound imprecise enough to ensure that any such chain have finite length.

Such a widening \((\sqcup)\) would receive two arguments, the first representing the old information that should be updated and the second representing the new information that should be incorporated with the old. Using a widening the update of \( x^2 \) at line 2 of the example would become \( x^2 \leftarrow x^2 \sqcup x_1 \) and \( x^2 \leftarrow x^2 \sqcup x_5 \) and the latter will ensure that the updates caused by the loop can not go on forever. The properties of a widening are formally defined in Definition 2.1.3.

Designing a suitable widening that ensures efficiency and soundness of the analysis and preserves as much relevant information as possible is a non-trivial task, we will see later that it can have a tremendous impact on the efficiency and precision of the analysis.

For the interval domain the classical widening is defined such that it over-approximates growth in either direction by positive or negative infinity respectively.

\[\begin{align*}
  a \sqcup b &= b & \text{if } a = \bot \\
  a \sqcup b &= a & \text{if } b \sqsubseteq a \\
  [l, h] \sqcup [l', h'] &= [l'^{\mu}, h'^{\mu}] & \text{otherwise}
\end{align*}\]

where

\[\begin{align*}
  l^{\mu} &= \begin{cases} 
  -\infty & \text{if } l' < l \\
  l & \text{otherwise}
  \end{cases} \\
  h^{\mu} &= \begin{cases} 
  \infty & \text{if } h' > h \\
  h & \text{otherwise}
  \end{cases}
\]
 Returning to our example, \( x^2 \) would become \([0, 0]\) the first time as before but the second time analysis reaches line 2 it would perform \( x^2 \leftarrow [0, 0] \uplus [2, 2] = [0, \infty] \) a value that will not change further during analysis. In this example the final result is the same as for our original domain of signs, but had line 1 of the program been \( x := -1 \) then analysis with intervals would yield \( x^3 = [1, \infty] = \text{Pos} \) as final result whereas analysis with the domain of signs would give \( x^3 = \top \).

A particularly simple way to obtain a widening is to map the domain onto a, typically finite, sub-domain with the ascending chain condition using a restriction operator \( R \). \( R \) should be conservative so \( a \subseteq R(a) \) must hold.

One possibility for the interval domain is to map an interval \([l, h]\) to one of the intervals corresponding to the domain of signs. Later, we will see several examples of widenings defined in this manner.

**Definition 2.1.2 (Restriction Widening)**

Let \( R : \mathcal{D} \rightarrow \mathcal{D} \) be a function such that the sub-domain \( \mathcal{D} \subseteq \mathcal{D} \) has no infinite ascending chains. A widening can the be defined as:

\[
a \uplus b = R(a \sqcup b)
\]

The analysis can benefit from a more precise domain even with a rather coarse widening since widening need not be performed except at cycles in the program. In the example widening was used at the beginning of the loop but it could be applied anywhere along a cyclic dependency. Widening could also be used everywhere, presumably leading to a faster, perhaps less precise, analysis.

There are two important properties a widening should have. First, the widening is used to combine an old approximation with new information in a sound way and so should be an upper bound operation. Second, it must ensure that the approximation updated in this way will reach a final value in a finite number of steps. This is formalized in Definition 2.1.3.
Definition 2.1.3 (Widening)

soundness \( a_1 \sqsubseteq a_1 \sqcup a_2 \land a_2 \sqsubseteq a_1 \sqcup a_2 \)

stationarity For any increasing chain \( a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \ldots \) the chain \( b_0 = a_0, b_1 = b_0 \sqcup a_1, \ldots, b_{i+1} = b_i \sqcup a_{i+1}, \ldots \) is not strictly increasing for \( \sqsubseteq \), that is, it should be a stationary sequence.

Several variations on this definition is possible, the widening could be defined so that it has access to all previous approximations in the sequence; it can be defined so that it has access to the length of the sequence, for example it could limit the resource usage by using a less precise approximation after a fixed number of steps.

The soundness property ensures that using a widening instead of the ordinary upper bound, as described above, will give a sound analysis.

The second property, stationarity, ensures that regardless of the particulars of the analysis an abstract value cannot be updated more than a finite number of times\(^1\). A proof of this particular termination aspect of an analyzer can thus be made considering properties of the widening and the domain only. Such a proof is then robust in the sense that changes to the analyzer framework will not affect this aspect of termination.

To ensure termination it is, however, only necessary to consider sequences of abstract values that could occur during analysis. Thus there may exist a sequence \( a_i, \ldots \) that would lead to an infinitely increasing sequence \( b_i, \ldots \) but if it can be shown that no such sequence could occur during analysis with a particular analyzer then the stationarity property will nevertheless hold for this analyzer. A problem with this relaxed requirement is that it can make it harder to obtain a proof of stationarity and the proof would be brittle, in the sense that stationarity must be re-established as soon as the analyzer framework is changed, or some parameter of it is modified. In particular it makes it hard to use the widening in other frameworks.

There are several, more or less equivalent, ways to define the widening properties. All the widenings considered in this thesis allows the second argument \( a_i \) to be an arbitrary (not necessarily increasing) sequence of abstract values.

\(^1\)If the abstract domain has the ascending chain condition then \( \sqcup \) is a widening by Definition 2.1.3.
2.1. Example: A Simple Abstract Domain

Definition 2.1.4

soundness \( a_1 \subseteq a_1 \uparrow a_2 \land a_2 \subseteq a_1 \uparrow a_2 \)

stationarity For any sequence \( a'_0, a'_1, a'_2, \ldots \) the chain \( b_0 = a'_0, b_1 = b_0 \uparrow a'_1, b_{i+1} = b_i \uparrow a'_{i+1}, \ldots \) is not strictly increasing for \( \subseteq \), that is, it should be a stationary sequence.

Given a widening \( \uparrow \) as in Definition 2.1.3 it is easy to obtain a widening \( \uparrow' \) satisfying Definition 2.1.4 simply by defining \( b \uparrow' a \) as \( b \uparrow (b \sqcup a) \).

The increasing sequence \( \cdots \subseteq a_i \subseteq a_{i+1} \subseteq \cdots \) required by Definition 2.1.3, can thus be obtained from an arbitrary sequence \( a'_0, \ldots, a'_{i-1}, \ldots \) by defining \( a_0 = a'_0 \) and \( a_{i+1} = b_i \sqcup a'_{i+1} \). In fact most widenings used in this thesis are formulated in this manner and for this reason the analyzer framework explicitly performs the upper bound between the old description \( b_i \) and the new information that should be added \( a'_{i+1} \).

Some widenings can be formulated to satisfy Definition 2.1.4 directly without the need to perform the upper bound \( b_i \sqcup a'_{i+1} \), either in the framework or as an initial step of the widening. This may be more efficient if upper bound is an expensive operation, as is the case for the grammar domains used in this thesis.

2.1.4 Summary of Abstract Domain Operations

The following list summarizes the usual domain operations and their required properties

- An abstract domain \( D \) of descriptions \( a \).
- A concretization function \( \gamma(a) \) that gives the set of concrete values corresponding to an abstract description \( a \).
- A containment relation \( a \subseteq b \), also called a precision relation, that satisfies \( a \subseteq b \implies \gamma(a) \subseteq \gamma(b) \). If \( a \sqsubseteq b \) or \( b \sqsubseteq a \) then \( a \) and \( b \) are said to be comparable.
- A least precise descriptor \( \top \) such that \( \forall a \ a \sqsubseteq \top \).
- A most precise descriptor \( \bot \) such that \( \forall a \bot \sqsubseteq a \) and \( \gamma(\bot) = \emptyset \).
- An upper bound operation \( a \sqcup b \) that abstracts union of concrete values. It is required that \( c = a \sqcup b \implies a \sqsubseteq c \land b \sqsubseteq c \) and also that \( \gamma(a) \sqcup \gamma(b) \subseteq \gamma(a \sqcup b) \)
• If the abstract domain have infinite increasing chains then a widening $a \uparrow b$ is required as discussed in the previous section. The widening should be an upper bound and also satisfy either of Definition 2.1.3 or Definition 2.1.4.

• An abstraction of intersection $a \cap b$ such that $c = a \cap b \implies \gamma(a) \cap \gamma(b) \subseteq \gamma(a \cap b)$.

2.2 ABSTRACT INTERPRETATION OF PROLOG

The execution model of Prolog differs significantly from the kind of imperative languages used in the previous section. Since the abstract interpreter simulates the concrete execution an abstract interpreter for Prolog will also be quite different from an abstract interpreter for an imperative programming language.

The following are the main characteristics that affects the analysis of Prolog:

1. Logical variables and unification. The variables in Prolog are terms of their own and can be unified with other variables or terms. If two variables have been unified then subsequent unification of either variable with some other term will also affect the other variable. This variable aliasing is troublesome to model since dependencies introduced by unification will have effect not only on the terms syntactically involved in a unification but also on any variable that are aliased to some (sub-)term involved in the unification.

2. Prolog control. Each predicate, corresponding to a procedure, consists of a number of clauses. During (concrete) execution these clauses will be tried in order starting from the first, until some clause succeeds. Within each clause the goals, corresponding to procedure calls, will be executed in sequence from left to right. The analyzer typically cannot know which clauses will actually be tried and so makes the simplifying assumption that any of the clauses could be used. This assumption gives the analyzer great freedom in choosing when a clause should be analyzed and makes it relatively easy to avoid analyzing clauses of a predicate for which the analysis results cannot have changed.

3. Single assignment. Prolog is a pure language, in the sense that once a variable have been unified with, or bound to, a non-variable term, the value of the variable will not change except that variables in the term may become bound through unification. This simplifies the analysis considerably compared to analysis of non-pure languages. It also enhances the precision when analyzing procedure calls since the arguments to a call cannot have changed when the call returns except that variables may have become bound.
4. Recursion. Prolog does not contain any explicit looping construct, but instead uses recursion. An immediate consequence of this is that analysis only within a procedure is insufficient to obtain interesting results. Therefore a global inter-procedural analysis is needed.

All of the concepts introduced in the previous section will be needed for analysis of Prolog. Instead of an abstract version of assignment the main workhorse is an abstract version of unification.

In the following only the concepts needed to understand the type analysis are presented. See [11] for a full account of abstract interpretation in a logic programming context.

2.2.1 Preliminaries

To simplify the presentation all programs are assumed to be in a simplified form. It is easy to transform an ordinary (Prolog) program into this normal form. The analyzer used in this thesis uses a less drastic normal form and treats disjunctions, if-then-else, all-solutions constructs and other builtin control structures directly. These details have little impact on the presentation and will, for the most part, be ignored.

A program in normal form consists of definite Horn clauses

\[ p(X_1, \ldots, X_n) : - l_1, \ldots, l_k. \]

A clause consists of two parts, a head \( p(X_1, \ldots, X_n) \) where \( p \) is an \( n \)-ary predicate symbol and \( X_i \), the argument variables, are distinct program variables, and a body \( l_1, \ldots, l_k \) of literals. The clauses with the predicate symbol \( p \) in the head make up the definition of the predicate, or procedure, \( p \). Note that the argument variables for all clauses of a particular predicate will be the same.

Variables occurring syntactically in the program are called program variables. The \( m \) program variables of a clause are assumed to come from the set \( \{ X_1, \ldots, X_n, X_{n+1}, \ldots, X_m \} \) where \( X_1, \ldots, X_n \) are the argument variables occurring in the clause head. Other variable names will be used for clarity.

Each literal \( l_i \) in the body is of one of the following forms.

1. \( q(Y_1, \ldots, Y_m) \) a goal, \( q \) an \( m \)-ary predicate symbol and all \( Y_i \) distinct program variables.

\(^2\)The only problem is the cut builtin, however treating cut as a predicate that always succeeds will give a sound, but less precise, analysis.
2. $Y_0 = f(Y_1,\ldots,Y_m)$ unification, $f$ an $m$-ary function symbol and all $Y_i$ distinct program variables.

3. $X = Y$ unification, $X$ and $Y$ distinct program variables.

As an example we will use the well known predicate to concatenate lists.

$$\text{append}([\ ], Ys, Ys).$$

$$\text{append}([X|Xs], Ys, [X|Zs]) :- \text{append}(Xs, Ys, Zs).$$

A normal form of this predicate is

$$\text{append}(X_1, X_2, X_3) :- X_1 = [\ ], X_2 = X_3.$$

$$\text{append}(X_1, X_2, X_3) :-$$

$$X_1 = [X_4|X_5], X_3 = [X_4|X_5], \text{append}(X_5, X_2, X_0).$$

Rewriting a program into normal form will not change the meaning of the program, however, it can affect the precision and efficiency of the analysis since more variables are introduced. For the abstract domains considered here, precision will not be affected and the cost of an extra variable is largely independent of the number of variables\(^3\).

A substitution $\theta$ is a function from a set of variables, the domain of the substitution, to terms. A substitution with domain $\{x_1,\ldots,x_n\}$ is written $\{x_1 \leftarrow t_1,\ldots,x_n \leftarrow t_n\}$ where $t_i$ is the value, or binding, of $x_i$ in $\theta$. A substitution with domain $V$ is said to be a substitution over the variables in $V$. An alternative and often convenient formulation is to consider a substitution $\{x_1 \leftarrow t_1,\ldots,x_n \leftarrow t_n\}$ as a $n$-ary tuple $\vec{t} = \langle t_1,\ldots,t_n \rangle$ of terms.

Substitutions are written with postfix notation so $t\theta$ is the term $t$ where variables in the domain of $\theta$ occurring in $t$ are replaced with their values.

**2.2.2 Analysis**

This section provides a brief description of the top-down analyzer used in this thesis. A more detailed description appear in Section 4.1.

Define the set $\text{CALLS}[p]$ to be the set of calls $p(\vec{t})$ to the predicate $p$ that occur during execution of a program. Similarly define $\text{SUCC}[p]$ as the set of $p(\vec{t})$ such that $\vec{t}$ are the arguments when $p$ succeeds (that is, a call to $p$ returns without failing).

\(^3\)In contrast, for some domains the cost is exponential in the number of variable in a clause [36].
Given an approximation \( \overline{\text{INIT}} \) of the set of possible initial goals \( q(\overline{T}) \in \text{INIT} \) the analyzer will compute an approximation of \( \text{CALLS}[p] \) and \( \text{SUCC}[p] \) for every predicate \( p \) in the program.

An approximation (or abstraction) of a set of possible calls \( p(\overline{T}) \) will be called a call pattern. The call pattern used to approximate \( \text{CALLS}[p] \), will be denoted \( \text{CP}[p] \). Similarly a success pattern is an abstraction of the way a predicate may succeed. The success pattern approximating \( \text{SUCC}[p] \) will be denoted \( \text{SP}[p] \).

The call and success patterns will be regarded as abstractions of sets of substitutions over the argument variables \( \{x_1, \ldots, x_n\} \).

**Definition 2.2.1 (Soundness)** Given a call pattern \( \overline{\text{INIT}} \) describing a set of possible initial goals \( q(\overline{T}) \in \gamma(\text{INIT}) \). The analysis result \( \text{CP} \) and \( \text{SP} \) is sound if and only if

1. if the execution of an initial goal \( q(\overline{T}) \) would lead to a call \( p(\overline{T}) \) then \( p(\overline{T}) \in \gamma(\text{CP}[p]) \)
2. if the execution of an initial goal \( q(\overline{T}) \) would lead the a call to \( p \) that succeeds with \( p(\overline{T}) \) then \( p(\overline{T}) \in \gamma(\text{SP}[p]) \)

\( \square \)

The analyzer will simulate the ordinary top-down SLD based execution starting from the abstract description of the initial goal.

The analyzer collects information about the program variables after the head of each clause and after each literal in a body. These program points will be written using \( \bullet \) with subscript or superscript, a clause with \( k \) literals in the body will have program point \( \bullet_0 \) before the first literal and \( \bullet_k \) last in the body.

The values of the program variables \( \{x_1, \ldots, x_m\} \) at a particular program point in a clause can be described by a substitution with domain \( \{x_1, \ldots, x_m\} \). So for each program point \( \bullet_i \) in a clause there will be a set \( \Theta_i \) of possible substitutions, all with the same domain \( \{x_1, \ldots, x_m\} \).

Abstractly performing a call to a predicate \( p \) starts with a call pattern describing the call, or equivalently an abstract substitution \( \beta_{\text{call}} \) over the argument variables \( \{x_1, \ldots, x_n\} \).

The analyzer will first extend the domain of \( \beta_{\text{call}} \) to include all the variables in the clause. This corresponds to the fact that the variables not in the clause head are unbound at program point \( \bullet_0 \).
The simulated execution then proceeds through the clause body simulating the goals in the body. Eventually the end of the clause is reached with an abstract substitution $\beta_k$. The variables \{$x_{n+1}, \ldots, x_m$\} are now removed from the domain of $\beta_k$ to give an abstract substitution describing the possible values of the arguments of $p$ when the call succeeds.

A call to a goal $q(y_1, \ldots, y_r)$ in the body of a clause with the current abstract substitution $\beta$ is performed in two steps. First a new abstract substitution $\delta$ is obtained by removing all variables except \{$y_1, \ldots, y_r$\} from the domain of $\beta$. Then the variables in $\delta$ are renamed so that $y_i$ instead becomes $x_i$. The abstract substitution $\delta$ now has domain \{$x_1, \ldots, x_r$\} and can be used as a call pattern for analyzing $q$ as outlined above.

Each time a new call pattern appear, that is one that is not already approximated by CP[$p$] the analyzer use a widening to update CP[$p$]. Each time a new success pattern is obtained from a clause SP[$p$] is updated similarly. Updates will happen only to strictly less precise call (success) patterns and since widenings are used this can only happen finitely many times.

A fixpoint has been reached when all clauses of $p$ have been analyzed with the call pattern CP[$p$] and when all clauses that call $p$ have been analyzed after the last time SP[$p$] was updated.

To ensure that information is propagated to all parts of the analyzed program a worklist will be used to record the clauses that need to be analyzed. When the worklist is empty a fixpoint has been reached and CP[$p$] and SP[$p$] will be sound approximations of the possible calls to and returns from $p$ respectively.

The next section describe the abstract substitutions used in the analyzer. The analyzer is described in Section 4.1.

### 2.3 Abstract Substitutions

A mentioned above the sets of substitutions of interest are infinite in general and instead abstract substitutions will be used to safely approximate these sets.

The abstract substitutions used in this thesis consists of two components, a type component (Section 2.3.2) that describes the possible terms each variable can be bound to, and a same value component (Section 2.3.3) that contains information about (sub-)terms of variables that are known to be equal.

The abstract substitutions and their operations will be used in an analyzer framework described in detail in Section 4.1. The analyzer framework require the following:
1. Abstract substitutions $\beta$ that abstracts sets of substitutions $\theta$, that is $\gamma(\beta)$ is a set of substitutions $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ or equivalently $n$-ary tuples of terms $\overline{t} = \langle t_1, \ldots, t_n \rangle$.

2. A containment operation $\beta \subseteq \delta$. Soundness require that $\beta \subseteq \delta \implies \gamma(\beta) \subseteq \gamma(\delta)$.

   During analysis abstract substitutions will be used to describe sets $\Theta$ of the possible arguments $\overline{t}$ to a predicate $p$. A set $\Theta'$ of possible arguments is a sound approximation of $\Theta$ if it contains all the argument-tuples in $\Theta$, that is, $\Theta \subseteq \Theta'$.

   The analyzer on the other hand will not manipulate (the in general infinite) sets of substitutions and so cannot check the soundness criteria directly. Instead the analyzer will use $\beta \subseteq \delta$ to determine that $\delta$ is a sound approximation of $\beta$. The soundness condition $\beta \subseteq \delta \implies \gamma(\beta) \subseteq \gamma(\delta)$ is a sufficient condition to guarantee that when the analyzer decides a sound approximation has been reached, using the test $\beta \subseteq \delta$, then the set of concrete substitutions described by $\delta$ is indeed a sound approximation of the concrete substitutions described by $\beta$.

   Note that the reversed condition is desirable but may not hold, that is, $\gamma(\beta) \subseteq \gamma(\delta) \implies \beta \subseteq \delta$ is not a necessary condition. However, if this condition does not hold then even though a sound approximation has been reached ($\gamma(\beta) \subseteq \gamma(\delta)$) this will not be detected by the analyzer if $\beta \not\subseteq \delta$. The effect of this is that the fixpoint iteration in the analyzer will needlessly continue until $\subseteq$ can be used to detect that a sound approximation has been reached.

3. A least precise element $\top$ such that $\forall \beta \beta \subseteq \top$.

4. A most precise element $\bot$ such that $\gamma(\bot) = \emptyset$ and $\forall \beta \bot \subseteq \beta$. Operationally $\gamma(\bot) = \emptyset$ corresponds to an impossible state, for example, a call that cannot happen.

5. An upper bound operation $\beta \cup \delta$. This is the abstract counterpart of union of two sets of substitutions.

   Soundness of the analysis require that this is a sound approximation of union of sets of substitutions. It is required that $\beta \subseteq \beta \cup \delta$ and $\delta \subseteq \beta \cup \delta$. Note that (by 1.) this implies $\gamma(\beta) \cup \gamma(\delta) \subseteq \gamma(\beta \cup \delta)$, that is, $\cup$ is a sound approximation of union.

6. A widening $\beta \triangledown \delta$. The widening is used to ensure that the analysis terminates. It should be an upper bound as in 2. above and in addition satisfy one of definitions 2.1.3 or 2.1.4 (Page 19).
7. In addition the analyzer require domain specific operations to handle builtins of which the most important is the abstract version of unification. These details are described with the analyzer framework in Section 4.1.

The abstract substitutions will be substitution closed (Section 2.3.1) and unification will correspond to an overapproximation of intersection, $\beta\delta$.

2.3.1 Variable Aliasing

Unification can make two (or more) variables aliased to each other. Subsequent binding of one of the variables will affect all the aliased variables. An analyzer must account for this non-local effect of unification to guarantee that the result of the analysis is sound.

In particular if two variables may be aliased then the analysis must ensure that when (abstractly) unifying one of the variables than the abstract value of the other variable must reflect that it may have been unified as well. Another equivalent way of viewing may alias information is as the complement of, and a way to represent, information that two variables are definitely not aliased.

Such may alias information can be added to the abstract substitution, as was done by Janssens [32]. However, maintaining this information is expensive and complicated. One of the reasons that this information is expensive to maintain is a that a variable may be aliased to quite a lot of other variables and that a conservative approximation of such aliasing would say that the variable may be aliased to even more variables. Since information becomes less precise as the analysis proceeds this means that may alias information becomes more and more expensive to maintain.

Information about aliasing that is known to hold, definite alias information, is less expensive to maintain. One reason is that a variable seldom is definitely aliased to more than a few other variables. Furthermore, conservative approximation of definite aliasing amounts to losing the information that two variables are definitely the same. Thus as the analysis proceeds and information becomes less precise the definite aliasing information will become less expensive to maintain.

The two kinds of alias information also differ in how complicated they are to maintain within abstract operations. If definite alias information is complicated to maintain for some operation then it is always safe to just throw away the definite alias information. May alias information on the other hand must not be lost as that could give an unsound analysis.

Definite alias information can enhance precision and a variant of definite aliasing information are used in the abstract substitutions in the analyzer.
Unfortunately this information is not enough to guarantee sound handling of unification.

If no may alias information is present then soundness of the analysis requires that the analyzer treats every unification as potentially affecting every variable. If the abstract domain is substitution closed [15] ( Definition 2.3.1 ) then the abstract interpreter can safely ignore may alias information.

**Definition 2.3.1 (Substitution Closed)** A set $S$ of terms is substitution closed if $\forall t, t' \in S \implies t \theta = t' \theta \in S$, that is, if $t'$ is a term obtained by binding variables in some term $t \in S$ then $t'$ too is in $S$. An abstract domain $D$, such as $D_{\text{type}}$, is substitution closed if $\gamma(a)$ is substitution closed for any $a \in D$.

If the analysis uses a substitution closed abstract domain then it does not have to track variable aliasing accurately since any instantiations would already be accounted for by the abstract substitution. An unfortunate corollary is that the abstract domain cannot represent the information that some (sub-)term is definitely a free variable\(^4\).

### 2.3.2 Type Component

A set $\Theta = \{ \theta_1, \ldots, \theta_i, \ldots \}$ of concrete substitutions over $n$ variables $\{ x_1, \ldots, x_n \}$, such that $\theta_i = \{ x_1 \leftarrow t^i_1, \ldots, x_n \leftarrow t^i_n \}$, will be approximated by an abstract substitution with type component $\{ x_1 \leftarrow T_{x_1}, \ldots, x_n \leftarrow T_{x_n} \}$ where $\forall i t^i_j \in \gamma(T_{x_j})$, and each type $T_{x_j} \in D_{\text{type}}$ is an abstraction of a set of terms. For $x$ a variable in the domain of an abstract domain the type of $x$, written $T_x$, will be used to refer to its type in the type component.

The type domain $D_{\text{type}}$ should come equipped with the usual domain operations ( Section 2.1.4 ). It should be substitution closed so that the analysis can ignore aliasing between variables.

When convenient the type component of an abstract substitution $\{ x_1 \leftarrow T_{x_1}, \ldots, x_n \leftarrow T_{x_n} \}, T_{x_i} \in D_{\text{type}}$ will be treated as a tuple of types where $T_i$, that is, element $i$ of the tuple, is the type of variable $x_i$.

The domain operations are defined using the corresponding operations for $D_{\text{type}}$. Note that if any $T_i$ is $\bot$ then the concretization for the whole type component will be empty and it will be equivalent to, and usually written as, the type component $\bot$. Note that $\gamma(\bot) = \{ \} \neq \gamma(\bot) = \emptyset$.

\(^4\)But see Section 4.3.2.1.
Definition 2.3.2 (Type Component Operations) Here a type component written $\bot$ should be taken to mean any type component with empty concretization. Except for the definition of $\gamma$ a type component written as $\langle \ldots \rangle$ is assumed to have non-empty concretization, that is, none of the element types is $\bot$. $T^n$ stands for any n-ary type-component (including $\bot$).

$$
\begin{align*}
\top &= \langle T_1, \ldots, T_n \rangle \text{ such that } T_i = \top \\
\gamma(\bot) &= \emptyset \\
\gamma(\langle T_1, \ldots, T_n \rangle) &= \{ \langle t_1, \ldots, t_n \rangle \mid t_i \in \gamma(T_i) \}
\end{align*}
$$

$\bot \subseteq T^n$

$\langle T_1, \ldots, T_n \rangle \not\subseteq \bot$

$\langle T_1, \ldots, T_n \rangle \subseteq \langle T'_1, \ldots, T'_n \rangle \iff \forall 1 \leq i \leq n T_i \subseteq T'_i$

$T^n = T'^n \iff T^n \subseteq T'^n \land T'^n \subseteq T^n$

$\bot \sqcup T^n = T^n \sqcup \bot = T^n$

$\langle T_1, \ldots, T_n \rangle \sqcup \langle T'_1, \ldots, T'_n \rangle = \langle T_1 \sqcup T'_1, \ldots, T_n \sqcup T'_n \rangle$

$\bot \sqcap T^n = T^n \sqcap \bot = \bot$

$\langle T_1, \ldots, T_n \rangle \sqcap \langle T'_1, \ldots, T'_n \rangle = \langle T_1 \sqcap T'_1, \ldots, T_n \sqcap T'_n \rangle$

$\bot \triangledown \langle T_1, \ldots, T_n \rangle = \langle \bot \triangledown T_1, \ldots, \bot \triangledown T_n \rangle$

$\langle T_1, \ldots, T_n \rangle \triangledown \langle T'_1, \ldots, T'_n \rangle = \langle T_1 \triangledown T'_1, \ldots, T_n \triangledown T'_n \rangle$

When an operation is applied to type components of two abstract substitutions then the substitutions will always describe bindings for the same set of variables. Concretely this means that the two tuples making up the type components will have the same length, as required by Definition 2.3.2.

2.3.3 Same Value Component

Substitution closedness makes it unnecessary to keep track of aliasing but comes at a price. Since a variable could be instantiated to any term an substitution closed set $S$ containing a variable would have to include all possible terms. For an instantiated closed domain this means that (sets containing) variables would have to be abstracted as $\top$. An abstract substitution corresponding to an unbound variable $\{X \leftarrow X\}$ would be abstracted as $\{X \leftarrow \top\}$ essentially losing all information about the value of $X$. This sometimes leads to unacceptable imprecision. Fortunately a large part of
this loss of precision can be alleviated by keeping track of (some) subterms that are known to have the same value in any concrete substitution.

As an example, consider execution of a clause and the substitutions $\theta_i$ over the variables $\{X,Y\}$ at the three program points $\bullet_i$.

$$\text{foo} :- \bullet_0 \ X = Y, \ \bullet_1 \ Y = g, \ \bullet_2 \ q(X).$$

At the beginning of the clause $X$ and $Y$ are (unaliased) free variables, so $\theta_0 = \{X \leftarrow X,Y \leftarrow Y\}$. With a substitution closed domain of types both $X$ and $Y$ would be abstracted as $\tau$, that is $\alpha(\theta_0) = \{X \leftarrow \tau,Y \leftarrow \tau\}$. After the first unification the concrete substitution would be $\theta_1 = \{X \leftarrow Y,Y \leftarrow Y\}$, corresponding to the fact that $X$ and $Y$ now are the same (free) variable. However, since both $X$ and $Y$ are still variables, the substitution closed abstract domain would still be $\{X \leftarrow \tau,Y \leftarrow \tau\}$.

Now consider the effect of the second unification, where $X$ is bound to $g$. $\theta_2$ would become $\{X \leftarrow g,Y \leftarrow g\}$, and then $q$ would be called with $g$ as argument.

Abstractly performing the unification $Y = g$ given the substitution $\{X \leftarrow \tau,Y \leftarrow \tau\}$ would result in $\{X \leftarrow \tau,Y \leftarrow \alpha(g)\}$ and the analysis would still know nothing about the value of $X$. Finally the analyzer would have to conclude that $q$ could be called with any of the possible values of $X$, that is $\gamma(\tau)$, the set of all terms.

The problem is made worse by the fact that putting a program into normal form will introduce variable dependencies like those in the example.

To work around this problem the abstract substitution domain will be extended with a same value component, $D_{same}$ that keeps track of some of the pairs of subterms that would be identical in any concrete substitution. The same value component that will be used in this section is almost identical to the one used by Janssens [31].

With same value information added to the example above $\{X \leftarrow \tau,Y \leftarrow \tau\}$ would instead become a pair $(\{X \leftarrow \tau,Y \leftarrow \tau\},\alpha(\{X = Y\}))$ where the same value component $\alpha(\{X = Y\})$, means that, even though the concrete values are unknown, $X$ and $Y$ will be identical in any concrete substitution. Note that the concrete constraint $X = Y$ that $X$ and $Y$ have the same value, would be represented by an element $\alpha(X = Y)$ from the abstract domain of same value constraints. When performing the abstract version of the unification $Y = g$ the analyzer would infer not only that $Y$ becomes bound to $g$ but that $X$ does too since $X$ is known to have the same value as $Y$. Finally the analyzer would infer that $q$ only could be called with $g$.

A same value component will sometimes increase precision even if no variables are present. Consider the abstract substitution $\alpha(\Theta)$ of a set of
concrete substitutions $\Theta = \{\theta_1, \theta_2\}$ with $\theta_1 = \{X \leftarrow g, Y \leftarrow g\}$ and $\theta_2 = \{X \leftarrow h, Y \leftarrow h\}$, $\alpha(\Theta)$ would be $\{X \leftarrow \alpha(\{g, h\}), Y \leftarrow \alpha(\{g, h\})\}$ so $\gamma(\alpha(\Theta))$ would be $\{\theta_1, \theta_2, \theta_3, \theta_4\}$ where $\theta_3 = \{X \leftarrow g, Y \leftarrow h\}$, $\theta_4 = \{X \leftarrow h, Y \leftarrow g\}$ are not in the original set $\Theta$ of concrete substitutions. The reason is that the type-component discards any dependencies between the bindings of two variables. Adding a same value constraint $\{X = Y\}$, obtaining $\beta = (\{X \leftarrow \alpha(\{g, h\}), Y \leftarrow \{g, h\}\}, \alpha(\{X = Y\}))$ would allow the concretization function for the abstract substitutions to conclude that $\theta_3, \theta_4$ are not in $\gamma(\beta)$.

2.3.3.1 Selectors To keep track of same value information not only between the values of variables in an abstract substitutions, but also between subterms of such values, we need a way to reference (or select) a subterm of such a value.

Define $t/s$, the subterm of a concrete term $t$ referenced by a selector $s$ inductively as follows. The empty selector $\epsilon$ refers to the term $t$, that is, $t/\epsilon = t$. If $t/s = t'$, $t'$ is a compound term $f(t_1, \ldots, t_i, \ldots, t_n)$ (where $f$ is an $n$-ary function symbol) then $t/s.(f,i) = t_i, 1 \leq i \leq n$. In general for two selectors $s, p$, if $t/s = t'$ and if $t'/p$ exists then $t/s.p = t'/p$. The initial $\epsilon$ of a non-empty selector will often be omitted, so $\epsilon.p$ will be written simply as $p$.

We say a selector $p$ extends $p'$ if $\exists s, p'.s = p$. We say that two selectors overlap if either extends the other.

Given a set $S$ of terms, a selector $s$ is determinate in $S$, if for all $t \in S$ the term $t/s$ exists, otherwise it is non-determinate. The selector $\epsilon$ is always determinate.

For a type $T$, $s$ is determinate in $T$ if (and only if) $s$ is determinate in $\gamma(T)$, $\gamma(T)$ is the set of abstract terms in $T$. If $s$ is determinate in $T$ then $T/s$ selects the type corresponding to the subterms selected by $s$, that is $T/s = \alpha(\{t/s | t \in \gamma(T)\})$.

For convenience we will sometimes write out the “determinate” parts of a type. For example, if $\gamma(T) = \{f(a,g(b)), f(a,g(h)), f(c,g(b)), f(c,g(h))\}$ then the selectors determinate in $T$ are $\{\epsilon, (f,1), (f,2), (f,2).g, (1, g,1)\}$ and the type can be written $f(T', g(T''))$ where $\gamma(T') = \{a, c\}$ and $\gamma(T'') = \{b, h\}$. A type written $T = f(\tau)$ would correspond to the set of, possibly nonground, terms constructed from the (unary) function symbol $f$. To further abuse notation we will use sets of such determinate parts to indicate disjunctive types, so for example $\{X \leftarrow \{g(T'), h(T'')\}\}$ means that the type $T$ of $X$ is such that $\gamma(T) = \{g(t'), h(t'') | t' \in \gamma(T') \cup \{h(t'') | t'' \in \gamma(T'')\}\}$.

Define subterm replacement $t'' = t/s \leftarrow t'$, for two terms $t, t'$ and a selector $s$ determinate in $t$, as the term identical to $t$ except that the subterm $t''/s$ is $t'$, thus, $f(c,g(h(d)))/(f,2).g, (1, b) = f(c, g(b))$. 


For a selector $s$ determinate in $T$ we require that the type domain can provide operations to select and replace a type corresponding to a determinate selector. A consequence of this is that if the type domain provides for determinate non-empty selectors then it must be able to represent accurately the structure of terms to an arbitrary (finite) depth.

The operation replace($T, s, T'$) ($s$ determinate in $T$), written $T\lceil s \leftarrow T'$, should return a new type $T''$ such that $\gamma(T'')$ is the same as $\gamma(T)$ except that subterms corresponding to $s$ now come from $T'$. That is $\gamma(T\lceil s \leftarrow T') = \{t\lceil s \leftarrow t' \mid t \in \gamma(T), t' \in \gamma(T')\}$. This operation should not introduce any imprecision so all $p$ that are determinate in $T$ and does not overlap with $s$ should be determinate in $T\lceil s \leftarrow T'$ and for all $s'$ that are determinate in $T'$ the selector $s.s'$ should be determinate in $T\lceil s \leftarrow T'$.

2.3.3.2 Abstract Same Value Constraints An abstract same value constraint is a pair $\{x_i/s_i, x_j/s_j\}$ such that $s_i, s_j$ are selectors. Intuitively the meaning of one such constraint is that for any concrete substitution $\{\ldots, x_i \leftarrow t_i, \ldots, x_j \leftarrow t_j, \ldots\}$ the subterms $t_i/s_i$ and $t_j/s_j$ exists and are identical\(^5\). It is possible for the two variables to be the same but then $s_i$ and $s_j$ cannot overlap (as that would either imply a circular term or a redundant constraint). In what follows abstract same value constraints will be called just same value constraints.

A same value constraint $C_1 = \{x/s_x, y/s_y\}$ is more restrictive than $C_2 = \{x/p_x, y/p_y\}$ ($C_2$ is less restrictive than $C_1$) if there exist an $s$ such that $p_x = s_x.s$ and $p_y = s_y.s$. If additionally $s \neq \epsilon$ then $C_1$ is strictly more restrictive than $C_2$ ($C_2$ is strictly less restrictive than $C_1$).

A same value constraint $\{x/s_x, y/s_y\}$ is determinate if $s_x$ is determinate in $T_x$ and $s_y$ is determinate in $T_y$. For determinate same value constraints (and determinate selectors in general), the function symbols in the selectors are redundant. The same value constraints will normally be determinate, if not they will be made determinate as part of a normalization procedure, described below.

**Definition 2.3.3 (Same Value Component)** The same value component of an abstract substitution is a set $\text{SV AL}$ of same value constraints determinate in the type component of the abstract substitution.

There are many equivalent sets of same value constraints. One reason is that equality is transitive, thus the set $\{\{X/s_x, Y/s_y\}, \{Y/s_y, p, Z/s_z\}\}$ conveys the same information as $\{\{X/s_x, Y/s_y\}, \{Y/s_y, p, Z/s_z\}, \{X/s_x, p, Z/s_z\}\}$.\(^5\)

\(^{5}\)Identical in the sense of the Prolog builtin \texttt{==/2}. 
Another reason is that constraints can be redundant, either since a more restrictive constraint is also present or because all corresponding arguments of two terms are pairwise equal, in which case the terms are equal. As an example of the latter, if $X/s_x$ and $Y/s_y$ both correspond to terms constructed using a binary function symbol $f$, then the constraints $\{X/s_x,(f,1),Y/s_y,(f,1)\}, \{X/s_x,(f,2),Y/s_y,(f,2)\}$ are equivalent to the single constraint $\{X/s_x,Y/s_y\}$.

In Section 2.3.4.1 a normalization algorithm for abstract substitutions are presented that among other things ensures that the same value constraints in an abstract substitution have a canonical representation.

2.3.3.3 Same Value Component Operations If $C_1$ is more restrictive than $C_2$, then the set of concrete substitutions for which $C_1$ holds is contained in the set for which $C_2$ holds. This is the same as saying that $C_1$ is more precise than (or contained in) $C_2$.

Definition 2.3.4 (Same Value Containment)

$$\{X/s_x,Y/s_y\} \subseteq \{X/p_x,Y/p_y\} \iff \exists s \ p_x = s_x \wedge p_y = s_y \wedge$$

This carries over to sets of same value constraint, so that a set $SVAL_1$ is more precise than $SVAL_2$, if every constraint in $SVAL_2$ is less precise than some same value constraint in $SVAL_1$. A consequence of this is that the top element for sets of same value constraints is the empty set.

Definition 2.3.5 (Containment)

$$SVAL_1 \subseteq SVAL_2 \iff \forall C_2 \in SVAL_2 \exists C_1 \in SVAL_1 : C_1 \subseteq C_2$$

From the definition of $\subseteq$ and the requirement $a, b \subseteq a \sqcup b$ follows that a suitable upper bound of two sets of same value constraints can be defined as the constraints from either that have a more restrictive constraint in the other.

The two same value components are assumed to be part of two normalized abstract substitutions $\beta, \delta$. This means that the same value components are determinate in their corresponding type components. The result of the upper bound is not necessarily determinate with respect to the upper bound $\text{TYPE}(\beta) \sqcup \text{TYPE}(\delta)$, this will be handled in the upper bound of abstract substitutions (Section 2.3.4.5).
Definition 2.3.6 (Upper Bound)

\[
SVAL_1 \sqcup SVAL_2 = \{ C_1 | C_1 \in SVAL_1, \exists C_2 \in SVAL_2 : C_2 \subseteq C_1 \} \cup \{ C_2 | C_2 \in SVAL_2, \exists C_1 \in SVAL_1 : C_1 \subseteq C_2 \}
\]

The concrete substitutions that satisfy both the constraints in \( SVAL_1 \) and \( SVAL_2 \), that is, the concrete substitutions in \( \gamma(SVAL_1) \cap \gamma(SVAL_2) \), are simply those that satisfy the union of these two sets of constraints.

Definition 2.3.7 (Intersection)

\[
SVAL_1 \hat{\cap} SVAL_2 = SVAL_1 \sqcup SVAL_2
\]

The result of same value intersection will typically not have transitivity explicit. These issues will be handled in the intersection for abstract substitutions (Section 2.3.5).

2.3.4 Abstract Substitution Operations

The final form of the abstract substitutions is either \( \bot \) or a pair \( \beta = (\{ x_1 \leftarrow T_x, \ldots, x_n \leftarrow T_x \}, SVAL) \), with accessors \( \text{TYPE}(\beta) \) and \( \text{SVAL}(\beta) \), respectively.

It is convenient to have a normal form of the abstract substitutions since there are many equivalent abstract substitutions.

2.3.4.1 Normalization

There are several factors that can make two abstract substitutions equivalent but unequal. One reason is that the same value component have many equivalent variants.

Another reason is that the type component can contain types that are unnecessarily general, for example, if there is a same value constraint \( \{ X/s_x, Y/s_y \} \) and if \( T_x/s_x \neq T_y/s_y \) in the type component then the type component can be strengthen to \( \{ X/T_x[s_x \leftarrow T], Y/T_y[s_y \leftarrow T] \} \), where \( T = T_x/s_x \hat{\cap} T_y/s_y \), without affecting the meaning of the abstract substitution.

Furthermore, sometimes the types are precise enough to convey some information about equal sub-terms. For example, in a type component \( \{ X/f(T), Y/g(T) \} \) with \( T \) a type describing a single concrete term, the same value constraint
\{X/(f, 1), Y/(g, 1)\} is already implied and need not be present in the same value component.

**Definition 2.3.8 (Implied Constraints)** The set of same value constraints implicit in the type-component

\[
\text{implied}(\{x_1 \leftarrow T_1, \ldots, x_n \leftarrow T_n\}) = \{\{x_i/s_i, x_j/s_j\} \mid \exists t \gamma(T_i/s_i) = \gamma(T_j/s_j) = \{t\}\}
\]

The selectors \(s\) for which condition \(\exists t \gamma(T/s) = \{t\}\) is true will be obtained directly using a type-domain operation (Section 3.4.1). There will be no need to refer to \(\gamma(T/s)\).

\[\square\]

To obtain a canonical representation of the abstract substitution a normalization procedure (Algorithm 2.3.9) is used that removes redundancy while ensuring that transitive same value constraints are explicit. Applying Rule R1 to R5 of Algorithm 2.3.9 exhaustively will eventually achieve this, without affecting the concretization of the abstract substitution. See Janssens’ thesis [31] for proofs and other details.
Algorithm 2.3.9 (Normalize Abstract Substitution) \text{normalize}(\beta): 

All same value constraints should already be determinate in $\beta$

Initially $\beta_n$ becomes $\beta$

Exhaustively apply Rule R1-R5 and when none of these apply, apply R6

\[ R1 \quad \text{[transitivity]} \]
if \( \{x/s_x, y/s_y\}, \{y/s_y, s_z/s_z\} \in \text{SVAL}(\beta_n) \)
and \( T_x/s_x \equiv T_y/s_y \) and \( T_y/s_y \equiv T_z/s_z \)
and \( \exists \{x/p_x, z/p_z\} \in \text{SVAL}(\beta_n): \exists p: p_x.p = s_x.p, p_z.p = s_z.p \)
then add \( \{x/s_x, s_z/s_z\} \) to \( \text{SVAL}(\beta_n) \)

\[ R2 \quad \text{[all args]} \]
if \( \exists f: 1 \leq i \leq \text{arity}(f): \{x/s_x, f(i), y/s_y, (f(i))\} \in \text{SVAL}(\beta_n) \)
then add \( \{x/s_x, y/s_y\} \) to \( \text{SVAL}(\beta_n) \)

\[ R3 \quad \text{[keep restrictive]} \]
if \( \{x/s_x, y/s_y\}, \{x/s_x, p, y/s_y, p\} \in \text{SVAL}(\beta_n), p \neq \epsilon \)
then remove the latter from \( \text{SVAL}(\beta_n) \)

\[ R4 \quad \text{[occur check]} \]
if \( \{x/s_x, x/s_x, s\} \in \text{SVAL}(\beta_n), s \neq \epsilon \)
then $\beta_n$ becomes $\bot$

\[ R5 \quad \text{[strengthen]} \]
if \( \{x/s_x, y/s_y\} \in \text{SVAL}(\beta_n) \)
and \( (\text{if } x = y \text{ then } s_x \text{ and } s_y \text{ does not overlap}) \)
and \( T_x/s_x \neq T_y/s_y \)
then
\[
T = T_x/s_x \bigcap T_y/s_y \\
\text{if } T = \bot \text{ then } \beta_n \text{ becomes } \bot \\
\text{replace } x/T_x, y/T_y \text{ in } \text{TYPE}(\beta_n) \text{ with } x/T_x[s_x \leftarrow T], y/T_y[s_y \leftarrow T] 
\]

\[ R6 \quad \text{[implied by type]} \]
Finally remove all same value constraints implicit in the type, i.e.,
Set \( \text{SVAL}(\beta_n) \) to \( \text{SVAL}(\beta_n) \setminus \text{implied}(\text{TYPE}(\beta)) \).

\[ \square \]

Rule R1 makes transitive constraints explicit unless already implied by a more restrictive constraint, the type compatibility test is necessary to ensure that the new constraint is determinate. Rule R2 introduces a more restrictive constraint when all corresponding argument positions have the same value, this makes the constraints on the arguments redundant. Rule R3 removes these and other redundant constraints for which a more restrictive constraint is already present. Rule R4 detects same value constraints that would imply a circular term. Rule R5 ensures that identical subterms are described by equivalent types.
As a final pass Rule R6 removes all same value constraints that are already implied by the type component. These same value constraints, implied by the type component (Definition 2.3.8), are then added back temporarily in the containment and upper bound operations.

Janssens did not handle specially the constraints implied by the type component. The reason I chose to do so is that this can lead to better precision and to fewer iterations to find a fixpoint during analysis, see Section 2.3.4.3.

Rule R1 to R5 are from Janssens’ thesis where it was suggested that the normalization can be implemented by first checking for explicit circularities (R4), and if none are present apply R5 exhaustively. After this, if $\beta_n \neq \bot$, by the transitivity of $\equiv$, the tests for type compatibility can be ignored and only R1-R4 need be applied. Rule R6 was not used by Janssens but it can be applied as a final pass to remove any implied constraints.

2.3.4.2 Concretization The concretization for an abstract substitution is the same as for the type component except that it only includes term tuples for which the same value constraints hold.

**Definition 2.3.10 (Abstract Substitution Concretization)**

\[
\begin{align*}
\gamma(\bot) &= \emptyset \\
\gamma((\text{TYPE}, \text{SVAL})) &= \\
&\{\{x_i \leftarrow t_1, \ldots, x_n \leftarrow t_n\} \in \gamma(\text{TYPE}) \mid \exists(x_i/s_i, x_j/s_j) \in \text{SVAL} \implies t_i/s_i = t_j/s_j\} \\
\end{align*}
\]

2.3.4.3 Containment Soundness of the analysis require that containment satisfies $\beta \subseteq \delta \implies \gamma(\beta) \subseteq \gamma(\delta)$, so that the analyzer can use the test $\beta \subseteq \delta$ to determine that a sound approximation ($\gamma(\delta)$) of the set of possible substitutions ($\gamma(\beta)$) has been reached.

The first part of the containment operation for abstract substitutions is to determine containment between the type components. By the requirement on the type component operation $\text{TYPE}(\beta) \subseteq \text{TYPE}(\delta) \implies \gamma(\text{TYPE}(\beta)) \subseteq \gamma(\text{TYPE}(\delta))$, which would be enough if the same value components are empty.

However the concretization for abstract substitutions (Definition 2.3.10) will only retain those elements from $\gamma(\text{TYPE}(\delta))$ that have subterms identical as specified by SVAL(\delta). For this reason the containment operation must ensure that any substitution $\theta$ removed from $\gamma(\text{TYPE}(\delta))$ (by not having subterms identical as specified by SVAL(\delta)) will also not be present in $\gamma(\beta)$. 
This is ensured by requiring that any subterm identical in the substitutions in \( \gamma(\delta) \) are also identical in \( \gamma(\beta) \). This in turn will be the case if the requirement is either imposed by the same value component of \( \beta \) or if it is already implied by the type component of \( \beta \).

**Definition 2.3.11 (Containment)**

\[
\beta \sqsubseteq \delta \iff \\
\beta = \bot \\
\lor \\
\delta \neq \bot, \\
\text{TYPE}(\beta) \sqsubseteq \text{TYPE}(\delta) \\
(\text{SVAL}(\beta) \cup \text{implied}(\text{TYPE}(\beta))) \sqsubseteq \text{SVAL}(\delta)
\]

Intuitively, adding the implied constraints to \( \text{SVAL}(\beta) \) will make it more likely that there will exist a more precise constraint in the left-hand side for each same value constraint in \( \text{SVAL}(\delta) \), so it will make containment for abstract substitutions come true more often while still being sound.

Adding implied constraints also to the right-hand side could never affect the outcome of the containment operation in this case since the fact that \( \text{TYPE}(\beta) \sqsubseteq \text{TYPE}(\delta) \) means that for each implied constraint in \( \text{TYPE}(\delta) \) there will exists a more restrictive constraint implied by \( \text{TYPE}(\beta) \).

2.3.4.4 Top  From the definition of containment it can be seen that the top element of the domain of abstract substitutions is \( \{x_1 \leftarrow \top, \ldots, x_n \leftarrow \top\} \).

2.3.4.5 Upper Bound  Upper bound of two abstract substitutions is used by the analyzer when combining a new call pattern with an existing one for some predicate and also when combining the result from a single clause with an existing success pattern.
Definition 2.3.12 (Upper Bound)

\[
\begin{align*}
\beta_1 \sqcup \bot &= \beta_1 \\
\bot \sqcup \beta_2 &= \beta_2 \\
\beta_1 \sqcup \beta_2 &= \delta, \text{ where} \\
\text{TYPE}(\delta) &= \text{TYPE}(\beta_1) \sqcup \text{TYPE}(\beta_2) \\
\text{SVAL}_{\beta_1} &= \text{SVAL}(\beta_1) \cup \text{implied}(\text{TYPE}(\beta_1)) \\
\text{SVAL}_{\beta_2} &= \text{SVAL}(\beta_2) \cup \text{implied}(\text{TYPE}(\beta_2)) \\
\text{SVAL} &= \text{SVAL}_{\beta_1} \cup \text{SVAL}_{\beta_2} \\
\text{SVAL}(\delta) &= \\
\{ C \in \text{SVAL} \mid C \text{ determine in } \text{TYPE}(\delta) \text{ and } T_x^\delta/s_x \equiv T_y^\delta/s_y \} \\
\setminus \text{implied}(\text{TYPE}(\delta))
\end{align*}
\]

Unless otherwise noted, the test \( T_x^\delta/s_x \equiv T_y^\delta/s_y \) will be true for the upper bound operations on types considered in this thesis. The test can thus be ignored. It was not needed for the upper bound on type graphs used in Janssens’ thesis and is only necessary when the upper bound is imprecise, for example when a widening operation is combined into the upper bound operation, as in Section 5.3.3.3.

2.3.4.6 Widening  Widening is just a particularly imprecise upper bound and it is defined exactly as Definition 2.3.12 above. The only difference is that widening should be done for the type component so

\[
\text{TYPE}(\delta) = \text{TYPE}(\beta_1) \sqcup \text{TYPE}(\beta_2)
\]

instead becomes

\[
\text{TYPE}(\delta) = \text{TYPE}(\beta_1) \lor \text{TYPE}(\beta_2)
\]

2.3.5 Abstract Unification

Intersection is used as the abstract counterpart to unification. This is possible since intersection on the abstract substitutions, and the type component in particular, is a conservative approximation of the intersection of the corresponding sets of concrete substitutions. Together with the fact that the domain is substitution closed this will give a conservative approximation of all possible substitutions obtainable by unifying a substitution from one set with a substitution from the other.

As an example consider the case when an abstract substitution is of the form \( \beta_x = (\{ X/T_x \}, \text{SVAL}) \), that is, it can be seen as an abstraction of the (substitution closed) set of terms \( S_x = \{ X\theta \mid \theta \in \gamma(\beta_x) \} \) to which \( X \) could
be bound, and another set of terms $S_y$ based on $\beta_y$ defined similarly. Now
unifying a term $t_x \in S_x$ with a term $t_y \in S_y$ amounts to finding a (most gen-
eral) unifier, that is, a substitution $\theta_{mgu}$ such that $t_x \theta_{mgu} = t_y \theta_{mgu}$. If such
a unifier exists then $t_x \theta_{mgu} \in S_x$ and $t_y \theta_{mgu} \in S_y$ since $S_x, S_y$ are substitution
closed. Thus the (substitution closed) set of terms formed by unifying a term from $S_x$ with a term $S_y$ is simply $S_x \cap S_y = \gamma(\beta_x) \cap \gamma(\beta_y)$ and from the requirement $\gamma(a) \cap \gamma(b) \subseteq \gamma(a \cap b)$ follows that unification is safely modeled by $\beta_x \cap \beta_y$. Also note that a concrete substitution $\{x_1/t_1, \ldots, x_n/t_n\}$ can be seen as a single $n$-ary term $\langle t_1, \ldots, t_n \rangle$ so the example is equally valid for any two abstract substitutions (over the same domain of variables $\{x_1, \ldots, x_n\}$).

Remains to show that $\gamma(\beta) \cap \gamma(\delta) \subseteq \gamma(\beta \cap \delta)$ really holds for the definition of intersection (Definition 2.3.13). If the same value component is empty (that is $\top$) then it follows immediately by the assumption that it does hold for the type component. Since equality of subterms is unaffected by instantiation and thus unification it follows that the same value constraint does not further restrict the set of unifiable terms provided the same value constraints hold before the unification. Thus enforcing the same value constraints will just remove unnecessary concrete substitutions.

**Definition 2.3.13 (Intersection)**

\[
\begin{align*}
\beta_1 \cap \bot & = \bot \\
\bot \cap \beta_2 & = \bot \\
\beta_1 \cap \beta_2 & = \text{normalize}(\text{TYPE}(\beta_1) \cap \text{TYPE}(\beta_2), \text{SVAL}(\beta_1) \cap \text{SVAL}(\beta_2))
\end{align*}
\]

One of the uses of abstract unification in the analyzer is to model the return from a call back into the body of a calling clause. The analyzer exploits the fact that the arguments passed into a call will be unifiable with the arguments returned from a call\(^6\). As an example consider the call to \texttt{p} in the clause \texttt{foo :- e_{1} p(X) e_{2}, q(X).} Abstract interpretation of this clause would start with an abstract substitution $\langle \{X \leftarrow \top\}, \emptyset \rangle$ at \texttt{e_{1}} since $X$ is a free variable. Assuming the single clause $\texttt{p(Y) :- Y=c.}$. Abstract interpretation of the call to \texttt{p} would result in the success pattern of $p$ would be $\langle \{Y \leftarrow c\}, \emptyset \rangle$ and by abstractly unifying $X$ with $Y$ the abstract substitution at \texttt{e_{1}} would become $\langle \{X \leftarrow c\}, \emptyset \rangle$, that is, the bindings performed by \texttt{p} are passed back into the body of the caller \texttt{foo}.

---

\(^6\)While this may seem to be trivial fact it is not true in the presence of for example \texttt{setarg/3} or similar operations, as found in some Prolog implementations.
Term Grammars

Term grammars will be used to describe the possible values of variables. A term grammar denotes a set of, not necessarily ground, terms.

This chapter describes how term grammars and various representations of these are used as the underlying type domain $D_{\text{type}}$ of the abstract substitutions in Section 2.3.

The first section defines term grammars and the operations necessary for use as an abstract domain. Section 3.3 describes particular ways to represent such grammars together with details of any modifications that are needed to the grammar operations to fit a particular representation.

The two representations of term grammars that will be used are type graphs and grammar graphs, directed graphs where nodes correspond to non-terminals and right-hand sides and the arcs correspond to the arrows of the grammar productions.

Type graphs were introduced by Janssens & Bruynooghe [32] and subsequently used by Hentenryck et al. [30]. Type graphs impose a tree-like shape on the graph representing the grammar which often make it necessary to duplicate nodes corresponding to non-terminals and right-hand sides. The shape of type graphs can make a type graph exponentially larger than the term grammar it represents in the worst case, and as we will see also in practice.

Grammar graphs are my generalization of type graphs to remove the requirement that the graphs have a tree-like shape. A grammar graph will therefore have approximately the same size as the term grammar it represents.

The last section of this chapter discuss the operations needed to use (a representation of) term grammars as the types in abstract substitutions.
3.1 REGULAR TERM GRAMMARS

Regular Term Grammars is the basis for all the type domains in this thesis. The type domains will use a restricted form, the deterministic term grammars. Equivalent formalism include top-down deterministic tree automata [8], deterministic root-to-frontier finite tree recognizers [24], and regular unary logic programs\(^1\) [57].

The type domain is used to provide the abstract substitutions with an abstraction of a set of terms corresponding to the possible values of a variable. The following definition defines *tuple distributivity* [40, 57] a property that will be satisfied by the set of terms defined by the grammar based type domains in this chapter. Tuple distributivity means that dependencies between argument positions of a term are lost.

**Definition 3.1.1 (Tuple Distributive Closure)** Let a path in a term \(t\) be \(st'\) where \(s\) is a selector in \(t\) (Section 2.3.3.1) and \(t' = t/s\) is a constant or a variable. That is, a path is a maximal length selector of \(t\) adorned with the “leaf” sub-term \(t'\) it selects. The set of all paths of \(t\) is \(\text{paths}(t)\) and for a set \(S\) of terms \(\text{paths}(S) = \bigcup_{t \in S} \text{paths}(t)\).

The tuple distributive closure of a set \(S\) is

\[
\{ t \mid t \text{ is a term and } \text{paths}(t) \subseteq \text{paths}(S) \}
\]

A set \(S\) of terms is tuple distributive if it is the same as its tuple distributive closure.

\[\square\]

As an example, for the set \(S = \{ f(a,g(b,c)), f(d,g(e,g)) \}\) the paths are \(\{(f,1).a, (f,2).(g,1).b, (f,2).(g,2).c, (f,1).d, (f,2).(g,1).e, (f,2).(g,2).g\}\). The tuple distributive closure of \(S\) is \(\{ f(a,g(b,c)), f(d,g(e,g)), f(d,g(b,c)), f(a,g(e,c)), f(d,g(b,g)), f(a,g(e,c)) \}\), since for example \(\text{paths}(f(d,g(b,g))) = \{(f,1).d, (f,2).(g,1).b, (f,2).(g,2).g\} \subseteq \text{paths}(S)\).

A *regular term grammar*, or grammar for short, describes a set of finite terms constructed from a finite alphabet \(F\) of ranked *function symbols* or *functors*. The rank or *arity* of a function symbol \(f\) is written \(\text{arity}(f)\), the set of function symbols of arity \(i\) is denoted \(F_i\), the zero arity symbols \(F_0\) are called constants.

A grammar \(G\) consists of a finite set \(R\) of productions \(N \rightarrow \text{rhs}\) where \(N \in \mathcal{N}\) is a non-terminal and the right hand side is either a non-terminal or \(f(N_1, \ldots, N_n)\) constructed from an \(n\)-ary function symbol and \(n\) non-

\(^1\)See also Section 3.3.4.
terminals. The concretization (or denotation) of a non-terminal, \( N \), \( \gamma(N) \), is the set of terms derivable from its production, that is,

\[
\gamma(N) = \bigcup_{(N \rightarrow \text{rhs}) \in \mathcal{R}} \gamma(\text{rhs})
\]

\[
\gamma(f(N_1, \ldots, N_n)) = \{ f(t_1, \ldots, t_n) \mid t_i \in \gamma(N_i) \}
\]

\[\square\]

The concretization of a grammar is the concretization of a distinguished non-terminal, the start symbol \( S \in \mathcal{N} \), of the grammar. We will often regard a non-terminal as a grammar and use the name of the start-symbol interchangeably with the name of the grammar.

As an example, the non-terminals in the following grammar defines a list \( L \) of constants \( a \) and \( b \), and a binary tree \( T \) with such lists as key.

\[
L \rightarrow [ ]
\]

\[
L \rightarrow .(E,L)
\]

\[
E \rightarrow a
\]

\[
E \rightarrow b
\]

\[
T \rightarrow \text{empty}
\]

\[
T \rightarrow t(L,T,T)
\]

The concretization of the type for the list elements \( E \) is \( \gamma(E) = \gamma(a) \cup \gamma(b) = \{a\} \cup \{b\} = \{a, b\} \). The concretization of \( L \) is \( \gamma(L) = \gamma([ ]) \cup \gamma(.(E,L)) \), that is, \( \{[ ]\} \cup \{(a, []), (b, []), (a, (a, [])), (b, (a, [])), \ldots \} \), an infinite set of arbitrarily long lists.

When we need to be explicit a grammar will be written as a four-tuple \( G = (S, \mathcal{N}, \mathcal{F}, \mathcal{R}) \). For the most part the maintenance of these sets will be left implicit.

The following two definitions are of particular importance. They define a restricted normal form of the grammars that will be used to represent types.

**Definition 3.1.3 (Normal Form)** A grammar is in normal form if none of the right-hand sides are non-terminals.

\[\square\]

It is always possible to rewrite a grammar into normal form, without affecting its concretization [8].
**Definition 3.1.4 (Deterministic Grammar)** A grammar is deterministic if it is normal and for each non-terminal $N$ the function symbols are all distinct in the right-hand sides of the productions for $N$.

Deterministic grammars are tuple distributive (Definition 3.1.1) and thus strictly less expressive than non-deterministic ones.

Consider the normal non-deterministic grammar $T$ denoting the set \{ $f(a, b), f(c, d)$ \},

\[
\begin{align*}
T & \rightarrow f(A, B) \\
T & \rightarrow f(C, D) \\
A & \rightarrow a \\
B & \rightarrow b \\
C & \rightarrow c \\
D & \rightarrow d
\end{align*}
\]

A deterministic grammar $T'$ with a concretization that includes $\gamma(T)$ would also have to include \{ $f(c, b), f(a, d)$ \}, that is,

\[
\begin{align*}
T' & \rightarrow f(AC, BD) \\
AC & \rightarrow a \\
AC & \rightarrow c \\
BD & \rightarrow b \\
BD & \rightarrow d
\end{align*}
\]

To ease the presentation non-terminals with a single production will often be “inlined” and multiple right-hand sides combined so the grammar $T$ would be written $T \rightarrow f(a, b) | f(c, d)$.

As presented above the concretization of a grammar corresponds to a Herbrand universe of ground terms, built from a finite alphabet $F$. To be able to describe terms containing numbers and variables we introduce two distinguished symbols \texttt{num} and \texttt{any} not in $F$.

The concretization of \texttt{num} is the set of all numbers. It is easy to replace \texttt{num} with more specific symbols with non-overlapping concretization such as integers and floats. While important for the precision of the analysis these details have little impact on the presentation and will, for the most part, be ignored.

The concretization of \texttt{any} is the set of all terms including variables. An alternative would be to use \texttt{free} denoting the set of all free variables. The obvious disadvantage with using \texttt{any} is that a grammar is unable to express that a term is a variable. The reason to nevertheless use \texttt{any} is that
the abstract substitutions in Section 2.3 require the type component to be substitution closed. It is possible to have a special symbol for free variables if more powerful abstract substitutions are used, this would correspond to Janssens’ integrated types [32].

If there is a production \( N \rightarrow \text{any} \) then no other productions for \( N \) will contribute to the concretization of \( N \). We will assume that such unnecessary productions are removed.

The special symbol \( \text{any} \) is also useful to represent unknown terms obtained from outside the analyzed program, either through builtins such as \text{read} and \text{name} or because the analyzer does not have access to the whole program.

To further simplify the use of grammars as type domains a grammar (or non-terminal) with an empty concretization is written using the bottom symbol (\( \perp \)).

It is convenient remove certain redundancy from the grammars, in particular non-terminals that cannot contribute to the concretization of the start symbol of the grammar. Definition 3.1.5 formalizes this.

**Definition 3.1.5 (Reduced)** A non-terminal is productive if its concretization is non-empty. A non-terminal is reachable if it can be reached via zero or more “\( \rightarrow \)” from the start symbol of the grammar. A grammar is reduced if all non-terminals are reachable and productive or if the grammar is \( \perp \).

Algorithm 3.1.6 will produce an equivalent reduced grammar from a normal deterministic grammar. It is based on a similar algorithm described in [8]. Note that reduction does not affect the concretization of a grammar since it just removes productions that could not contribute to the concretization of (the start symbol of) the grammar.

The grammar operations that will be used by the analyzer will always receive reduced normal deterministic grammars as input and will produce normal deterministic grammars as result directly. There will therefore not be a need for algorithms to normalize a grammar or to make a grammar deterministic. Furthermore, removal of unreachable non-terminals will be left implicit and most operations will only produce productive non-terminals. The exception is intersection that will have to explicitly perform something similar to Algorithm 3.1.6 as a final step to ensure that the resulting grammar is reduced (it will already be normal and deterministic by construction).

The size of a grammar \( G \), denoted \(|G|\) is defined to be the number of non-terminals in the grammar. The size of a non-terminal \( N \) will be taken to
Regular Term Grammars

mean the number of non-terminals reachable from $N$. Since unreachable non-terminals are always assumed to be removed implicitly the size of a grammar $G$ is the same as the size $|S|$ of its start-symbol $S$.

A grammar is minimal if it contains the fewest possible non-terminals for a given concretization of its start symbol. For the grammars used in this thesis, that is, normal deterministic grammars, the minimal form is unique up to renaming of non-terminals.

To see that the minimal form is unique consider two equivalent normal deterministic grammars that are minimized (and thus reduced), $G_a, G_b$ with start symbols $S_a, S_b$ respectively. The grammars are equivalent if and only if the start symbols are equivalent, that is $\gamma(S_a) = \gamma(S_b)$. num can be considered as just a constant since its denotation does not overlap with that of any other function symbol. If there is a production $N \rightarrow \textbf{any}$ then that is the only production for that non-terminal and two such productions are thus unique up to renaming of non-terminals. Since the grammars are deterministic and normal there will be exactly one production $N \rightarrow f(\ldots, N_i, \ldots)$ for each top-level function symbol $f$ of terms in $\gamma(N)$. Two equivalent non-terminals must therefore have the same number of productions and the right-hand sides of the productions for a particular function symbol $f$ must be equivalent. Two right-hand sides $f(\ldots, N_i, \ldots)$ $f(\ldots, M_i, \ldots)$ are equivalent if and only if $\forall i \gamma(N_i) = \gamma(M_i)$ (by Definition 3.1.2). This establishes equivalence classes of non-terminals, starting with $\{S_a, S_b\}$, the equivalence classes will contain at least one non-terminal from each of the grammars and in fact will contain exactly one from each since if more than one non-terminal from the same grammar were present in an equivalent class then this implies that one of them could replace the other, contradicting the assumption that the grammars are minimal. Since the equivalence classes are disjoint a non-terminal can be uniquely identified with the equivalence class to which it belongs thus providing a systematic way to rename the non-terminals in the two grammars so that the two grammars becomes identical.

Grammar operations will typically not result in minimal grammars unless an explicit minimization is performed. Note that minimality implies that the grammar is reduced. Minimization can be done in a similar way to finite automata minimizations [8, 55]. The method used in the analyzer when minimizing (the representations of) grammars is a naive version that, roughly, just compares each pair of non-terminals $T, T'$ for equality and if equal replaces all occurrences of $T$ with $T'$.\[\]


Algorithm 3.1.6 (Reduce)
Define the sequence $PR_i$ for a grammar $(S, N, F, R)$ as

$$
PR_0 = \emptyset \\
PR_{i+1} = \{ N \in N | \exists (N \rightarrow \text{rhs}) \in R \text{ every non-terminal in rhs is in } PR_i \} \cup PR_i
$$

The set of productive non-terminals $PR$ of a grammar $G$ is $PR_n$ for $G$, where $n$ is the smallest $n$ such that $PR_n = PR_{n+1}$.

Define the sequence $RE_i$ for a grammar $(S, N, F, R)$ as

$$
RE_0 = \{ S \} \\
RE_{i+1} = \{ N \in N | \exists (N' \rightarrow f(\ldots, N, \ldots)) \in R : N' \in RE_i \} \cup RE_i
$$

The set of reachable non-terminals $RE$ of a grammar $G$ is $RE_n$, where $n$ is the smallest $n$ such that $RE_n = RE_{n+1}$.

To obtain the equivalent reduced grammar from a grammar $G = (S, N, F, R)$ first compute the productive non-terminals $PR$ of $G$, let

$$
G' = \begin{cases} \\
\bot & \text{if } S \notin PR \\
(S, PR, F, R') & \text{otherwise} \\
\end{cases}
$$

where $R'$ is the subset of $R$ containing only productive non-terminals

then, unless $G' = \bot$, compute the reachable non-terminals $RE$ of $G' = (S, N', F, R')$. Finally return

$$
G'' = \begin{cases} \\
\bot & \text{if } G' = \bot \\
(S, RE, F, R'') & \text{otherwise} \\
\end{cases}
$$

where $R''$ is the subset of $R'$ containing only reachable non-terminals

\[ \square \]

3.2 OPERATIONS ON TERM GRAMMARS

The abstract substitution operations in Section 2.3 are formulated in terms of corresponding operations on the type domain. The operations necessary are the usual domain operations (Section 2.1.4) upper bound $\sqcup$, containment $\subseteq$, and abstract intersection $\hat{\cap}$ and also the operations necessary to interact with the same value component of the abstract substitution, that is, type selection and replacement using determinate selectors (Section 2.3.3.1).

The most critical operation, however, is the widening that is used. A widening is necessary since a domain based on term grammars have infa-
finite ascending chains. Most of this thesis is concerned with widenings on (representations of) term grammars and how they affect the efficiency and precision of the resulting analyses.

For a non-terminal in a normal deterministic grammar there is a one-to-one correspondence between the productions and the “top-level” of their right-hand sides on one hand and the concretization of the non-terminal on the other. Given a set of terms $S$ the task of constructing a non-terminal $N$ with the smallest concretization that includes $S$ is completely determined by the top-level function symbols (the principal labels) of the terms in $S$. This makes it easy to construct the algorithm for upper bound, containment and intersection and it gives these algorithms reasonable asymptotic complexity. It also makes it straightforward to show that the upper bound is a least upper bound, and that containment and intersection are precise, in the sense that $(N \subseteq N') \iff \gamma(N) \subseteq \gamma(N')$ and $\gamma(N \cap N') = \gamma(N) \cap \gamma(N')$.

### 3.2.1 Union

The least upper bound, or union of two, not necessarily normal nor deterministic, grammars $G_1 = (S_1, N_1, F_1, R_1)$ and $G_2 = (S_2, N_2, F_2, R_2)$ is simply, $G_1 \cup G_2 = (S, N_1 \cup N_2, F, R_1 \cup R_2 \cup \{S \rightarrow S_1|S_2\})$, assuming the non-terminals in $G_1$ and $G_2$ are renamed apart.

A normalization step can then be applied to remove the two productions \{ $S \rightarrow S_1|S_2$ \}.

Unfortunately the resulting grammar is not necessarily deterministic, even if $G_1$ and $G_2$ are, so a further determinization step would be needed to produce the deterministic union $G_1 \sqcup G_2$. Since the argument grammars will be deterministic it is possible to instead construct a deterministic upper bound directly, as in the next section. For this reason the union will not be used and no determinization algorithm is needed.

### 3.2.2 Deterministic Union

If $G_1$ and $G_2$ are deterministic and normal (as will always be the case in the analyzer) then a deterministic union, the least upper bound for deterministic grammars, can be computed directly. The algorithm for deterministic union (Algorithm 3.2.1) and the algorithms that follows operate on two non-terminals from the same (normal deterministic) grammar. It is easy to reformulate in terms of different grammars if needed.

---

2Renaming apart is not needed if it is ensured that two non-terminals have the same name only if they have the same concretization. This will be the case if upper bound is performed on two non-terminals from the same grammar.

3In general this is not minimal and not unique.
\[
\begin{align*}
T_1 & \rightarrow c(T_2) \\
T_1 & \rightarrow d & T_4 & \rightarrow c(T_5) \\
T_2 & \rightarrow c(T_3) & T_4 & \rightarrow g \\
T_2 & \rightarrow e & T_5 & \rightarrow c(T_4) \\
T_3 & \rightarrow c(T_1) & T_5 & \rightarrow h \\
T_3 & \rightarrow f & & \\
\end{align*}
\]

Figure 3.1: Example grammars. The upper bound of $T_1$ and $T_4$ will have the worst case number of non-terminals, that is $|T_1| \times |T_4| = 3 \times 2$.

**Algorithm 3.2.1 (Deterministic Union)**

*Deterministic union of two non-terminals in a normal deterministic grammar $(N_1 \cup N_2)$.*

\[
\begin{align*}
\text{Seen} & \ := \ \emptyset \quad \text{— used as a map from } N \times N \text{ to new non-terminals} \\
\text{return } \text{upp}(N_1, N_2) \\
\end{align*}
\[
\begin{align*}
\text{upp}(N, N') & = \\
\quad & \text{if } N = \text{any or } N' = \text{any then return any} \\
\quad & \text{if } N = N' \text{ then return } N \\
\quad & \text{if } \exists M \langle N, N', M \rangle \in \text{Seen} \text{ then return } M \\
\quad & \text{create a new non-terminal } M \\
\quad & \text{Seen} := \text{Seen} \cup \{ \langle N, N', M \rangle \} \\
\quad & \text{for each } f \text{ with productions} \\
\quad & \quad N \rightarrow f(A_1, \ldots, A_n), \\
\quad & \quad N' \rightarrow f(B_1, \ldots, B_n) \\
\quad & \quad \text{add a production } M \rightarrow f(\text{upp}(A_1, B_1), \ldots, \text{upp}(A_n, B_n)) \\
\quad & \text{for the remaining } f \\
\quad & \quad \text{such that there is only one right-hand side } f(M_1, \ldots, M_n) \\
\quad & \quad \text{add a production } M \rightarrow f(M_1, \ldots, M_n) \\
\quad & \text{return } M \\
\end{align*}
\]

The resulting grammar is deterministic and the maximum number of non-terminals created is $|N_1| \times |N_2|$. The global variable $\text{Seen}$ is used as a memoization table to remember encountered non-terminals and so ensure termination\(^4\). The same technique is used throughout.

\(^4\)Since $N_1$ and $N_2$ can overlap it might be an advantage to let the entries in $\text{Seen}$ be $\langle \{N, N'\}, M \rangle$ so that symmetric pairs are detected as well. This optimization applies to the other algorithms as well.
\begin{align*}
T_0 & \rightarrow c(T_7) \\
T_0 & \rightarrow d \\
T_0 & \rightarrow e \\
T_7 & \rightarrow c(T_6) & T_{T_6 \cup T_8} & \rightarrow c(T_{T_6 \cup T_8}) \\
T_7 & \rightarrow e & T_{T_6 \cup T_8} & \rightarrow d \\
T_7 & \rightarrow f & T_{T_6 \cup T_8} & \rightarrow e \\
T_7 & \rightarrow f & T_{T_6 \cup T_8} & \rightarrow f
\end{align*}

Figure 3.2: Example grammars. $T_6 \cup T_8$ is not minimal even though none of the non-terminals reachable from $T_6$ and $T_8$ are comparable.

As an example of the worst case behavior, consider $T_1 \cup T_4$ with grammars as in Figure 3.1. The resulting grammar will be minimal but it will still have one non-terminal for each pair $T_i, T_j, i \in \{1, 2, 3\}$, $j \in \{4, 5\}$.

The result of deterministic union may not always be minimal, even if the two involved grammars are. Consider $T_6 \cup T_8$, Figure 3.2, where none of the involved non-terminals are comparable, still the result would be the non-minimal grammar $T_{T_6 \cup T_8}$.

The fact that upper bound, in the worst case, causes quadratic growth and generally produces non-minimal grammars suggests that it might be advantageous to minimize the grammar after performing upper bound.

3.2.3 Containment

Containment $N \subseteq N'$ of two non-terminals correspond exactly to the containment relation on their concretizations, that is $N \subseteq N' \iff \gamma(N) \subseteq \gamma(N')$. In particular it satisfies the requirement $N \subseteq N' \Rightarrow \gamma(N) \subseteq \gamma(N)$ as required for use with the abstract substitutions of Section 2.3.

\footnote{See \cite{39} for a proof of this and a detailed complexity analysis for an imperative version of this algorithm.}
Algorithm 3.2.2 determines containment for normal deterministic grammars. An algorithm for equivalence (≡) can be defined directly in a similar way.

Containment essentially performs the same work as the deterministic upper bound and so has a worst case time-complexity quadratic in the number of non-terminals.

Containment is perhaps the most speed-critical component in the analyzer. It is used to detect if a fixpoint has been reached. It can also be used to detect the common case of comparable arguments to (deterministic) union and to intersection.

**Algorithm 3.2.2 (Containment)**

Containment between two non-terminals in a normal deterministic grammar \( (N_1 \subseteq N_2) \).

\[\text{Seen} \text{ is a global variable}\
\text{Seen} := \emptyset \quad \text{a subset of } N \times N\
\text{return } \text{leq}(N_1, N_2)\]

\[
\text{leq}(N, N') =
\text{if } N' \rightarrow \text{any then return true}
\text{if } N = N' \text{ then return true}
\text{if } (N, N') \in \text{Seen} \text{ then return true}
\text{Seen} := \text{Seen} \cup \{(N, N')\}
\text{for each production } N \rightarrow f(M_1, \ldots, M_n) \text{ do}
\text{if there is a production } N' \rightarrow f(M'_1, \ldots, M'_n) \text{ then}
\text{if } \text{leq}(M_i, M'_i) \text{ is true for all } i \in [1, n] \text{ then}
\text{return true}
\text{else}
\text{return false}
\text{else}
\text{return false}\]

\[\square\]

### 3.2.4 Intersection

As described in Section 2.3.5, intersection is used to model unification. Intersection too can result in a grammar quadratic in the size of the input. Unlike the case with upper bound, this property of intersection has not been a problem in practice.

The need to reduce the grammar as a final step of intersection (Algorithm 3.2.3) can be seen when computing \( T_0 \cap T_N \), Figure 3.2. The result of the call \( \text{intersect}(T_0, T_N) \) would be the non-productive and thus non-reduced grammar \( T' \rightarrow c(T') \). \( T' \rightarrow c(T) \). As an optimization the case when
a non-terminal would get no productions should be detected immediately and ⊥ returned.

Algorithm 3.2.3 (Intersection)
Intersection of two non-terminals in a normal deterministic grammar $(N_1 \cap N_2)$.

- $\text{Seen}$ is a global variable
$\text{Seen} := \emptyset$ — used as a map from $N \times N$ to new non-terminals
return \text{reduce(intersect}(N_1, N_2))

\begin{align*}
\text{intersect}(N, N') = \\
& \quad \text{if } N = \text{any then return } N' \\
& \quad \text{if } N' = \text{any then return } N \\
& \quad \text{if } N = N' \text{ then return } N \\
& \quad \text{if } \exists M (N, N', M) \in \text{Seen} \text{ then return } M \\
& \quad \text{create a new non-terminal } M \\
& \quad \text{Seen} := \text{Seen} \cup \{(N, N', M)\}
\end{align*}

for each $f$ with productions
\begin{align*}
N & \rightarrow f(A_1, \ldots, A_n), \\
N' & \rightarrow f(B_1, \ldots, B_n)
\end{align*}

add a production $M \rightarrow f(\text{intersect}(A_1, B_1), \ldots, \text{intersect}(A_n, B_n))$

return $M$

3.3 REPRESENTING TERM GRAMMARS

The main part of this section describes grammar graphs, a concrete representation of a term grammar as a graph where nodes correspond to non-terminals and right-hand-sides.

Type graphs are a restricted form of grammar graphs. The theoretical foundation and a prototype implementation for using type graphs in abstract interpretation of Prolog, was described in Janssens’ thesis [31] and summarized in an article by Janssens and Bruynooghe [32]. In their work they consider two kinds of type graphs. Both what they call rigid types, with identical expressibility as the grammars described here, and integrated types, that in addition allows a special terminal for unbound variables\(^6\). Integrated types are not instantiation-closed so, while more precise, this requires that the analyzer keep track of possible aliasing. Many of the results presented in this thesis would be applicable to integrated types as well.

A final section describes how grammars can be represented as logic programs on a particular form, a representation that will not be used in this thesis but have been used by other researchers to formulate widenings that can be

\^6Thus, in effect, integrating modes into the type.
formulated for grammars. See sections 6.5.5 and 6.2.4 for the widenings in question.

3.3.1 Preliminaries

The following definitions will be used later.

A (directed) graph $G$ is a pair $(N,A)$ of nodes $n \in N$ and (directed) arcs $a \in A$, $A$ a subset of $N \times N$. We often abuse notation and use $G$ to denote the set of nodes $N$.

A rooted graph is a graph with a distinguished node called the root. We often abuse notation and use $G$ to denote the root of the rooted graph $G$ and also use a node $n \in G$ to denote the subgraph, with root $n$, reachable from $n$.

If there is an arc $(n,n') \in A$ the node $n$ is a predecessor of $n'$ and $n'$ is a successor of $n$. For an arc $a = (n,n') \in A$ $n$ is the source and $n'$ the destination or target of the arc, furthermore, $a$ is an outgoing arc of $n$ and an incoming arc of $n'$.

A path is a sequence $n_0, \ldots, n_k$ of nodes such that, if $k > 1$, $(n_i, n_{i+1}) \in A$. A simple, or acyclic, path is a path with all $n_i$ distinct, otherwise the path is cyclic. If there is a path $(n_i, \ldots, n_j)$ then $n_j$ is reachable from $n_i$.

A directed acyclic graph (or DAG), is a graph with no cyclic paths. A (rooted) tree is a DAG satisfying

1. There is a distinguished node, the root, with no incoming arcs.

2. All nodes except the root have exactly one incoming arc.

3. There is a path from the root to each node.

A successor in a tree is called a child, a predecessor in a tree is called a parent. The ancestor $(\text{anc}(n))$ and descendant $(\text{desc}(n))$ relations are the transitive and reflexive closures of the parent and child relations respectively. A proper ancestor $(\text{anc}^+(n))$ of a node $n$ is an ancestor distinct from $n$, similarly for proper descendant. The depth of node in a tree is the number of arcs on the path from the root to the node. The depth of a tree is the depth of its deepest node.

A spanning tree of a graph $(N,A)$ is a tree $(N,A')$ where $A' \subseteq A$. A spanning tree of a rooted graph should have the same root as the graph. In general the spanning tree is not unique, if it is then it will sometimes be called simply the tree of the graph. In the context of some (usually
3.3. Representing Term Grammars

unique) spanning tree of a graph the relations child, parent, (proper) ancestor, (proper) descendant and depth are all defined for the graph as the corresponding relations on the spanning tree.

We will in fact use a slight variation of graphs where the nodes and the arcs are labeled. The label of a node $n$ is denoted $lb(n)$ and for a set $S$ of nodes we define $\{S_{lb} = \{n \in S \mid lb(n) = l\}\}$. When the letter $f$ is used to denote a label then it is implicit that the label is a function symbol (as opposed to integers or the special labels any and or introduced in the next section).

If there are $k$ outgoing arcs from $n$ then they are denoted $(n, i, n_i)$ each with a distinct label $i \in [1, k]$ and the $n_i$ not necessarily distinct. Sometimes function symbols $f$ will be used as arc labels instead.

For an arc $(n, l, m)$ we say $m$ is the $l$:th argument of $n$ or, in general, $m$ is argument $l$ of $n$. Argument $l$ of $n$, denoted $n \downarrow l$ is sometimes treated as a modifiable attribute of the node so setting $n \downarrow l$ to $m'$ means that any arc $(n, l, m)$ is replaced with an arc $(n, l, m')$. When convenient the label $l$ will be omitted. The function $args(n)$ returns the arguments of $n$.

We also use $x \downarrow i$ to denote the $i$:th (one-based) argument of a tuple $x$ and generalize to sets $S$ of nodes or tuples so that $S \downarrow l = \{x \downarrow l \mid x \in S\}$.

3.3.2 Grammar Graphs

A term grammar can be viewed as a rooted graph where nodes correspond to non-terminals and right-hand sides of productions and where arcs correspond to the arrow ($\rightarrow$) of productions or references from right-hand sides to non-terminals. A distinguished node, the root, corresponds to the start-symbol of the grammar. A grammar represented as a graph of nodes, as described in detail below, will be referred to as a grammar graph.

The grammar graphs will be used to represent normal deterministic grammars as defined in the previous section and all operations are easily derived from the corresponding operations on normal deterministic grammars.

In addition to the (syntactic) requirement that a grammar graph is normal and deterministic certain redundancy will be removed. The grammar concepts productive, reachable and reduced (Definition 3.1.5) can easily be adapted to grammar graphs. We will generally assume that the grammar graphs are reduced. In addition the grammar graphs will be compact, defined below, corresponding to replacing non-terminals that only have a single production with the right hand side of that production.

Nodes are labeled, the function $lb(m)$ gives the label of any node $m$. The function $arity(m)$ is defined for every node $m$ and is the same as the number of outgoing arcs of the node. A simple node is a node with zero arity, this
includes constants, and any special nodes such as \texttt{num} used to denote the set of all numbers.

Each non-terminal \( N \) is represented as an \textit{or-node} \( m \) with a labeled arc \((m, f, m_f)\) to each functor node \( m_f \) representing a right-hand side of a production \( N \rightarrow f(\ldots) \). It is sometimes convenient to treat the successors of an or-node as ordered according to some total order on the function symbols and to use integers instead of function symbols as labels for the arcs from an or-node. The \textit{label} of an or-node \( m \) is \( \text{lb}(m) = \text{or} \) where \text{or} is a special symbol not among the function symbols.

If there is a production \( N \rightarrow \text{any} \) then it is the only production for \( N \) and there would be a single arc to an any-node.

A \textit{functor-node} corresponds to a right-hand side \( f(N_1, \ldots, N_n) \). A functor-node \( m \) has \( n \) arcs \((m, i, m_i)\) to the (not necessarily distinct) nodes \( m_i \) representing the non-terminals \( N_i \). For such a functor-node \( m \) the label \( \text{lb}(m) \) is \( f \).

An \textit{any-node} corresponds to a right-hand side \text{any}. It is often convenient to assume that there is at most one node representing \text{any} for each grammar and to refer to it under the name \text{any}. The label of an any-node \( m \) is \( \text{lb}(m) = \text{any} \) where \text{any} is a special symbol not among the function symbols.

The concretization (or denotation) of a node is formulated equivalently to the case for grammars (Definition 3.1.2)

\textbf{Definition 3.3.1 (Concretization of Grammar Graph Node)} The concretization of a grammar graph node \( n \).  

\[
\gamma(n) = \bigcup_{1 \leq i \leq \text{arity}(n)} \gamma(n \downarrow i) \quad \text{if} \ \text{lb}(n) = \text{or}
\]

\[
\gamma(n) = \{f(t_1, \ldots, t_{\text{arity}(n)}) \mid t_i \in \gamma(n \downarrow i)\} \quad \text{if} \ \text{lb}(n) = f
\]

\[
\gamma(\text{any}) = \text{the set of all terms}
\]

\[
\gamma(\text{num}) = \text{the set of all numerical terms}
\]

\[
\square
\]

The \textit{principal nodes} of a node is the set of functor-nodes corresponding to the top-level functors in the concretization of the node. For an or-node it is the arguments of the node and for a functor-node it is the singleton set containing the node itself. The function is overloaded on sets of nodes in the obvious manner.
Definition 3.3.2 (Principal Nodes)

\[
\text{prnd}(n) = \begin{cases} 
\text{args}(n) & \text{if } n \text{ is an or-node.} \\
\{n\} & \text{if } n \text{ is a functor-node.} \\
\emptyset & \text{if } n \text{ is an any-node.}
\end{cases}
\]

\[
\text{prnd}(S) = \bigcup_{n \in S} \text{prnd}(n)
\]

The principal labels of a node or a set of nodes are the labels, that is, function symbols, of the principal nodes.

Definition 3.3.3 (Principal Labels)

\[
\text{prlb}(x) = \{ f | f = \text{lb}(m), m \in \text{prnd}(x) \}
\]

With this representation an or-node \( m \) with a single outgoing arc \((m, m')\) is redundant and all references to \( m \) can be replaced with a reference to \( m' \). This means that any remaining or-node will have at least two successors and that they will all be functor-nodes. If a normal deterministic grammar graph is also reduced then it is compact.

Algorithm 3.3.4 will build a compact grammar graph from a reduced normal deterministic grammar graph. Note that this does not affect the concretization of the grammar graph, it only makes the representation more compact. In what follows it is assumed that all grammar graphs are compact unless otherwise noted.

\(^7\)For convenience \texttt{num} will be regarded as a special function symbol, it too can appear as a child of an or-node.
Algorithm 3.3.4 (Compact)

Define

\[
\text{use}(n) = \begin{cases} 
  m & \text{if } n \text{ is an or-node and } (n,m) \text{ is the only outgoing arc from } n \\ 
  n & \text{otherwise} 
\end{cases}
\]

Let \( r \) be the root of the grammar graph. Designate \text{use}(r) as the new root. Replace each arc \((n, l, m)\) with an arc \((n, l, \text{use}(m))\). This may make some or-nodes unreachable.

Note that in a compact grammar graph any cycle will involve at least one or-node (with at least two successors) and at least one function node. The reason is that a cycle involving only function nodes would make all the function nodes unproductive and since the grammar graphs are normal a successor of an or-node cannot be an or-node.

Several or-nodes are allowed to refer to the same functor-node, in effect sharing right-hand sides of the corresponding productions. A consequence of this is that for each compact (and thus also reduced deterministic and normal) grammar graph exists a unique minimized grammar graph such that no two nodes are equivalent, that is not two nodes have the same concretization.

A good algorithm for minimizing compact grammar graphs could use techniques similar to how automata are minimized [8]. The implementation used in the analyzer uses a very naive algorithm that compares each pair of nodes \( n, n' \) for equality and replace arcs to \( n \) with arcs to \( n' \), see Section 6.1.1 for a discussion.

3.3.3 Type Graphs

Type graphs were introduced by Janssens and Bruynooghe as a formalism for representing sets of terms [32]. Type graphs have a shape that is tree-like in a manner formally defined below. The tree shape makes it easy to visualize and reason about this representation of term grammars. The disadvantage is that enforcing the tree shape can make a type graph (exponentially) larger than the corresponding minimal grammar graph.
Definition 3.3.5 Type Graph
A type graph is a compact grammar graph with a unique (rooted) spanning tree with the same root as the grammar graph.

This means that the arcs of a type graph belong to either of two classes; the forward arcs making up the spanning tree and backward arcs, each backward arc going from a node to an ancestor (in the spanning tree) of that node. A forward path is a path consisting of only forward arcs.

The uniqueness of the spanning tree makes relations such as ancestor, parent and child well defined for type graphs. This is often convenient and is utilized by the type graph specific widenings in Chapter 5.

A consequence of the tree-like shape of a type graph is that different parts of the type graph cannot share nodes except where the node is a common ancestor reached via backward arcs. This implies that a minimized type graph may have to contain several nodes with the same concretization in order to maintain the tree-like shape, in the worst case a minimal type graph can be exponentially larger than the corresponding minimal grammar graph. We will see later that this is a real problem in practice.

3.3.3.1 Operations on Type Graphs  The usual operations on type graphs are very similar to those of grammar graphs and grammars. The main difference is that the algorithms that create new type graphs, such as upper bound (∪) and intersection (∩), must ensure that the result is properly tree-shaped. To achieve this two changes are necessary. Firstly, the memoization structure Seen should be passed along as an argument so that a node n looked up in Seen can only become successor of a descendant of n. Secondly, when a pre-existing node n from a type graph G is included in a newly created type graph the nodes reachable from n must be rebuilt, and nodes duplicated, if there are nodes reachable from n that used to be ancestors of n in the original type graph G. Both of these changes will cause the resulting type graphs to become larger than the corresponding grammar graph.

Algorithm 3.3.6, construct(n), will rebuild a node n from a grammar graph (or type graph) and all nodes reachable from n so that the rebuilt n becomes a root of a proper type graph. This also gives a method to convert a grammar graph to a type graph, this will be used to evaluate grammar graph widenings on the type graph representation. Applying construct(n) will not affect the concretization, that is γ(n) = γ(construct(n)).
Algorithm 3.3.6 (Construct)
Construct a properly formatted type graph from an arbitrary node of a compact grammar graph. construct\( (n) = \text{build}(n, \emptyset) \).

\[
\begin{aligned}
\text{build}(n, \text{Seen}) = \\
&\quad \text{if } n \text{ is a simple node then return } n \\
&\quad \text{if } \exists m \in \text{Seen} \text{ then return } m \text{ — i.e., a backward arc to } m \\
&\quad \text{if } \text{lb}(n) = f \text{ then} \\
&\quad \quad \text{create a new functor-node } m \text{ with } \text{lb}(m) = f \\
&\quad \quad \text{Seen'} := \text{Seen} \cup \{(n, m)\} \\
&\quad \quad \text{for each } i \in [1, \text{arity}(n)] \text{ do} \\
&\quad \quad \quad m \downarrow i := \text{build}(n \downarrow i, \text{Seen'}) \\
&\quad \text{else} \\
&\quad \quad \text{create a new or-node } m \text{ with } \text{prlb}(m) = \text{prlb}(n) \\
&\quad \quad \text{Seen'} := \text{Seen} \cup \{(n, m)\} \\
&\quad \quad \text{for each } f \in \text{prlb}(n) \text{ do} \\
&\quad \quad \quad m \downarrow f := \text{build}(n \downarrow f, \text{Seen'}) \\
&\quad \text{end if} \\
&\quad \text{return } m \\
\end{aligned}
\]

As an example of the modifications necessary for type graphs the upper bound operations is presented in Algorithm 3.3.7. As discussed in Section 3.2.1 the upper bound on grammars, as well as grammar graphs, can result in quadratic increase in size. The requirement to maintain the tree-like shape for type graphs make these issues even worse. Later we will see that this can cause extremely large type graphs in practice even when the corresponding minimal grammar graphs are small. Intersection suffers from similar problems in theory but this does not appear to be a problem in practice for the analyses considered in this thesis.

For containment \((\subseteq)\) (and equality), however, the corresponding algorithms on grammar graphs can be used directly. That is, global memoization of encountered pairs of nodes will give an, asymptotically as well as practically, more efficient algorithm. With a non-global memo table the cost of containment for type graphs is the same as for upper bound which is not even polynomial in the sizes of the arguments. With a global memo-table the complexity is proportional to the product of the sizes of the arguments. This means that, especially for type graphs, it is advantageous to check the common case of comparable arguments to avoid creating an upper bound. This is done for all grammar based domains as part of the algorithms for upper bound and intersection.
Algorithm 3.3.7 (Upper Bound) Least Upper Bound (i.e., Deterministic Union) of (nodes of) two compact type graphs
\( n \sqcup m = \text{upp}(n, m, \emptyset) \).
Define \( n \downarrow f = n \) for a functor-node \( n \) with \( \text{lb}(n) = f \).

\[
\text{upp}(n, n', \text{Seen}) =
\begin{cases}
\text{any} & \text{if } n = \text{any} \text{ or } n' = \text{any} \text{ then return any} \\
\text{any} & \text{if } \exists m (n, n', m) \in \text{Seen} \text{ then return } m \text{ --- i.e., a backward arc to } m \\
\text{any} & \text{if } \text{lb}(n) = f \text{ and } \text{lb}(n') = f \text{ then}
\text{create a new functor-node } m \text{ with } \text{lb}(m) = f \\
\text{Seen'} := \text{Seen} \cup \{(n, n', m)\} \\
\text{for each } i \in [1, \text{arity}(n)] \text{ do} \\
\quad m \downarrow i := \text{upp}(n \downarrow i, n' \downarrow i, \text{Seen'}) \\
\text{else} \\
\text{create a new or-node } m \\
\text{Seen'} := \text{Seen} \cup \{(n, n', m)\} \\
\text{for each } f \in \text{prlb}(n) \cup \text{prlb}(n') \text{ do} \\
\quad \text{if } f \in \text{prlb}(n) \text{ and } f \in \text{prlb}(n') \text{ then} \\
\quad \quad m \downarrow f := \text{upp}(n \downarrow f, n' \downarrow f, \text{Seen'}) \\
\quad \text{elif } f \in \text{prlb}(n) \text{ then} \\
\quad \quad m \downarrow f := \text{construct}(n \downarrow f) \\
\quad \text{else } (f \in \text{prlb}(n')) \\
\quad \quad m \downarrow f := \text{construct}(n' \downarrow f) \\
\end{cases}
\]
\( \text{return } m \)
3.3.4 Regular Unary Logic Programs

A normalized Term Grammar can be represented as a logic program with unary clauses, a Regular Unary Logic (RUL) program, in the following way.

For each grammar production \( T_0 \rightarrow f(T_1, \ldots, T_n) \), use a clause

\[
p_{T_0}(f(X_1, \ldots, X_n)) \leftarrow p_{T_1}(X_1), \ldots, p_{T_n}(X_n).
\]

where \( X_i \) are distinct variables, \( f \) an \( n \)-ary function symbol, and each suffixed \( p_T \) a unary predicate that succeeds precisely for \( \gamma(T_i) \), the set of terms derivable from \( T_i \).

This way of presenting term grammars have been used to specify types for logic programs [40, 57, 13]. Due to the equivalence between (Deterministic) Regular Unary Logic Program and the kind of term grammars considered here some of the methods proposed should be applicable in our context with little or no modification.

Of particular relevance in this thesis is the work by Gallagher et al. [22, 50] where RUL programs are used to represent types in an abstract interpreter. Their analysis framework uses bottom-up analysis and is thus not directly comparable to the top-down framework used in this thesis\(^8\). The RUL-based domain is, however, sufficiently similar that their widening methods should apply also to the grammar representations used in this thesis. Two of their widenings are discussed in sections 6.5.5 and 6.2.4, where it is shown that they do not guarantee termination.

3.4 TERM GRAMMAR AS ABSTRACT DOMAIN

The analyzer will use one of the representations of deterministic term grammars as the underlying type-domain, \( D_{\text{type}} \), for the type-component of the abstract substitutions described in Section 2.3.

The type domain used by the abstract substitutions must be substitution closed, a property satisfied by term grammars by the use of the special symbol \texttt{any}. Furthermore the usual domain operations must be available. These operations, containment (\( \subseteq \), Section 3.2.3), upper bound (\( \sqcup \), Section 3.2.1), and intersection (\( \cap \), Section 3.2.4) have already been defined earlier in this chapter. These operations all have the necessary properties to be used with the abstract substitutions.

Two additional components are necessary in order to use the domains based on term grammars as part of the abstract substitutions. The first component is the operations necessary to interact with the same value component,\(^8\) See sections 2.2.2 and 4.1.
Figure 3.3: An infinite chain, $L_0 \sqsubseteq L_1 \sqsubseteq \ldots$, of non-terminals where $\gamma(L_i)$ is the set of lists of length at most $i$.

that is, operations to select and replace types as specified by (determinate) selectors, this will be described in the next section.

The second, and the most critical, component of the abstract domain is the choice of widening. Widening is necessary since term grammars do not have the ascending chain condition. Consider for example the types $L_0 \sqsubseteq L_1 \sqsubseteq L_2 \sqsubseteq \ldots$ of Figure 3.3, where the grammar (or non-terminal) $L_i$ denotes the set of lists of numbers where the length of each list is at most $i$. In the following chapters several widenings will be studied.

### 3.4.1 Type Selection and Replacement

The abstract substitution operations require operations to select and replace a type corresponding to a determinate selector. These operations are very simple for term grammars. The corresponding operations for grammar graphs can easily be derived\(^9\).

A selector $e$ is determinate in $T$ and selects $T$. A selector $p.(f,i)$ is determinate in $T$ and selects $T_i$, if $p$ is determinate in $T$ and selects $T'$, $f$ is an $n$-ary function symbol, $1 \leq i \leq n$, and there is only one production for $T'$, and that production is $T' \rightarrow f(T_1, \ldots, T_n)$.

\(^9\)Note that, for type graphs construct (Algorithm 3.3.6) is needed to convert an internal node of a type graph to a proper type graph.
Algorithm 3.4.1 (Select) \( \text{select}(T, s) \):
\[ s \text{ determinate in } T. \]
\[ T/s = \begin{cases} 
T & \text{if } s = \epsilon \\
T_i' & \text{if } s = p.(f, i), \text{ where,} \\
& \quad T' = T/p \\
& \quad \text{the production for } T' \text{ is } T' \rightarrow f(T'_1, \ldots, T'_n) 
\end{cases} \]

To replace a non-terminal referenced by a determinate selector the non-terminals, which all have a single production, along the path described by the selector, are copied. This can make some non-terminals along the path unreachable, if needed these can be removed.

Algorithm 3.4.2 (Replace) \( \text{replace}(T, s, T') \):
\[ T, T' \text{ non-terminals of the same grammar, } s \text{ determinate in } T. \]
\[ T/[s \leftarrow T'] = \begin{cases} 
T' & \text{if } s = \epsilon \\
T' & \text{if } s = (f, i).p, \text{ and} \\
& \quad \text{the production for } T \text{ is } T \rightarrow f(T'_1, \ldots, T'_n) \\
T_i' & = T_i/[p \leftarrow T'] \\
T'' & \text{is a new non-terminal with a single production,} \\
& \quad T'' \rightarrow f(\ldots, T_{i-1}, T_i', T_{i+1}, \ldots) 
\end{cases} \]

The final operation required of the type domain is a method to obtain all selectors \( s \text{ determinate in } T \) such that \( \gamma(T/s) = \{t\} \). This is used by the abstract substitution operations to add and remove same value constraints implicit in the type component, Definition 2.3.8.

For \( T \) a non-terminal in a normal deterministic reduced term grammar \( \exists t \gamma(T) = \{t\} \) if \( T \) has a single production with right-hand side \( f(T_1, \ldots, T_n) \) where \( f \) is a function symbol and either \( n = 0 \) or recursively \( \gamma(T_i) = \{t_i\} \) for all \( T_i \). This is easily formulated for the various representations of term grammars.

3.4.2 Occur Check

Recall from Section 3.1 that the term grammars considered in this thesis describe a set of finite terms. Of particular importance is that this rules out cyclic terms.
By considering only acyclic terms the analysis will only be correct if unification is done using the *occur-check*. Without the occur-check unification can create cyclic terms, whereas unification with occur-check would fail if an attempt is made to create a cyclic term. For efficiency reasons occur-check is typically *not* done in Prolog implementations
\(^\text{10}\).

The grammar domains used in this thesis shares this potentially serious deficiency with most analyses proposed for logic programs and for Prolog. There are two ways to attack this problem. The first is to require that the programmer guarantees that no cyclic structures will ever be created. Unfortunately it is not possible to automatically verify such a guarantee since the property that occur-check is unneeded is undecidable in general.

An alternative approach is to use unification with occur-check. While potentially expensive it is not clear how expensive this would be in practice since it is possible to use an analysis \([51, 5, 31]\) to detect opportunities for using the less expensive unification without the occur-check.

\(^{10}\)A Prolog implementation conforming to the ISO Prolog standard is allowed to but not required to implement the occur-check.
EXPERIMENTAL SETUP

This chapter describes the machinery necessary to evaluate the abstract domains. The necessary items are:

1. An abstract interpretation framework. The abstract interpretation framework will be parameterized by an abstract domain with associated domain operations and widening.
   The abstract interpreter repeatedly analyzes predicates and clauses until no changes occur in the abstract call and success patterns. A worklist, roughly a set of clauses, is used to keep track of the clauses that should be analyzed. The order clauses are taken off the worklist to be analyzed, the scheduling policy, turns out to have a large impact on the efficiency of the analysis. For this reason several scheduling policies are considered.
   The analyzer and the worklist scheduling policies are described in sections 4.1 and 4.2 respectively.

2. A way to measure performance. Performance is both how much resources the analysis require and also how good the analysis results are.
   Measuring analysis time and memory consumption says little about the inherent cost of an analysis. If time and memory consumption is low then it indicates that the analysis may be inherently inexpensive. If, on the other hand, the analysis is slow then this only shows a property of the particular implementation. For a number of reasons the implementation used in this thesis cannot be expected to use an asymptotically optimal amount of resources. For this reason resource usage will be measured in an implementation independent way. The main measure will be the size of the largest (representation of) grammar used in the type domain.
   The quality of the analysis result is not a well defined concept. The quality depend on how the information should be used. For this reason two simple estimates of precision will be used instead.
The various performance criteria are discussed in Section 4.3.

3. Benchmarks, Section 4.4. The analyzer must have some programs to analyze. Two classes of benchmark programs, synthetic and realistic are used.

The synthetic benchmarks are chosen to highlight particular aspects of the abstract domains. Most of the synthetic benchmarks are simple recognizers for recursively defined types. I will argue that an analysis method, in particular an abstract domain, is inherently infeasible if there are costs that increase exponentially (or worse) with the number of function symbols of the simple recursive type.

The realistic benchmarks are chosen to give an indication of the performance that could be expected analyzing real programs. The selection of a representative sample of programs is notoriously difficult. There is always the risk that the chosen programs lack some important characteristic present in other programs.

I consider two popular sets of benchmarks, the Aquarius benchmarks and the GAIA benchmarks. These benchmarks were assembled for the expressed purpose of being representative of real programs. They have been used to evaluate a number of analyses.

However, I found that these benchmarks can hardly be considered representative at all. This is especially true of the GAIA benchmarks for which the analysis results would be meaningless.

The benchmarks that will be used are the Aquarius benchmarks with and without modifications and also a number of larger benchmarks obtained from other sources.

4.1 ANALYZER FRAMEWORK

The analyzer implements an abstract version of the standard top-down, left-to-right execution model based on SLD-resolution. The main ideas were outlined in Section 2.2.2.

The input to the analyzer is a description of an initial goal corresponding to a set of possible concrete goals. The analyzer simulates the operations that a concrete Prolog interpreter would perform when interpreting one of the concrete initial goals. In particular the analyzer simulates the predicate calls that would occur during a concrete execution. A description of a call to a predicate \( p \) will be called a \emph{call pattern}, a description of the return from a call will be called a \emph{success pattern}.

When simulating a call to a predicate \( p \) with arguments \( T \) (a sequence of concrete terms corresponding to the arguments of \( p \)) the analyzer simulates the concrete interpretation of each of the clauses of \( p \) with these arguments.
When simulating the return from a predicate $p$ the analyzer combines the results from all the simulated clause executions to obtain the result of the simulated return from $p$. A worklist with clauses to analyze is maintained to ensure that changed information propagates to all affected parts of the program.

Soundness of the analysis require that if, during execution starting from an initial goal, the concrete interpreter would perform a call to a predicate $p$ with arguments $\tilde{t}$ then the description of the arguments to $p$ (the call pattern of $p$) must be a correct description of $\tilde{t}$. Similarly, if the concrete interpreter performs a call to a predicate $p$ with arguments $\tilde{t}$ that succeeds with arguments $\tilde{t}'$ then the success pattern of $p$ must be a correct description of $\tilde{t}'$.

Concrete arguments $\tilde{t} = (t_1, \ldots, t_n)$ is equivalent to a substitution $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ and the descriptions used will be the abstract substitutions $\beta$ from Section 2.3. The soundness condition that a description $\beta$ is a correct description of $\tilde{t} = \theta$ then becomes $\theta \in \gamma(\beta)$.

The result of the analysis is one call pattern and one success pattern for each predicate $p$, corresponding to all the calls to $p$. The analyzer used in this thesis is modelled after the analyzer that Getzinger used to evaluate various abstract domains for the Aquarius Prolog compiler [25]. It is also quite similar to a framework used by Debray [15].

A more precise analysis is possible by maintaining several pairs of call and success pattern for each predicate. Such an analysis is polyvariant as opposed to the mono-variant analysis performed by the analyzer in the next section. There are several reasons why a polyvariant analysis can be more precise. One is that the upper bound used to summarize descriptions may lose information if the upper bound is not precise$^1$. Another use of polyvariant analysis is to enable multiple specialization [56, 45], where several version of each predicate $p$ are compiled, each optimized for a particular call pattern. Multiple specialization requires that the call patterns can be associated with a particular program point.

The analyzer used by Janssens and Bruynooghe [32, 4] performs the abstract interpretation by a top-down construction of an abstract and-or graph that summarizes the proof trees corresponding to SLD-derivations for the initial queries. Their analyzer is polyvariant and distinguishes between calls to a predicate that occur at different places in the and-or tree (corresponding to different program points).

The analyzer framework PLAI [41] is a concretization and optimization of Bruynooghe’s framework that in each iteration tries to keep those parts of

---

$^1$The upper bound operation on the grammar based type domains (Chapter 3) does lose precision in this manner.
the and-or graph that are unaffected by the propagated changes. PLAI computes a fixpoint by analyzing all clauses of a called predicate until no change occurs in the success pattern of the predicate.

The analyzer framework GAIA [6] is also polyvariant. It can be seen as an instantiation of Bruynooghe’s framework but does not explicitly construct an and-or graph, instead multiple call-success-pattern pairs are maintained for each predicate. The analyzer tries to ensure that a call is not needlessly re-analyzed by recording for each pair of a call and success pattern the pairs on which it depends. To reach a fixpoint the clauses of a called predicate are analyzed repeatedly until no change occurs on which the analyzed call depends.

Note that neither GAIA not PLAI uses a worklist to propagate changes, instead they repeatedly analyze the clauses of a call until a local fixpoint have been reached. PLAI detects a local fixpoint of a call to a predicate $p$ when an analysis pass over the clauses of $p$ produced no change in the success pattern. Another technique is to detect when no changes has occurred in any information that would be used by the last analysis pass, thus avoiding the need to perform an analysis just to detect that it had no effect. This optimization is used by GAIA and a similar effect, although coarser due to the mono-variant analysis, is obtained by using a worklist as in the analyzer used in this thesis.

### 4.1.1 Overview

Starting from a description of an initial goal, the analyzer will collect for each predicate $p$ a pair $(\beta^p_{\text{call}}, \beta^p_{\text{success}})$ such that if a call $P(\overline{x})\theta$ ($\overline{x}$ is a sequence of distinct variables) with concrete substitution $\theta$ can occur during execution, resulting in $\theta_1, \theta_2, \ldots$, then the abstract substitution $\beta^p_{\text{call}}$ (the call pattern) will be a safe approximation of $\theta$ and the abstract substitution $\beta^p_{\text{success}}$ (the success pattern) will safely approximate any $\theta_i$, that is, $\theta \in \gamma(\beta^p_{\text{call}})$ and $\theta_i \in \gamma(\beta^p_{\text{success}})$. The superscript $p$ will be dropped when the predicate is apparent from the context.

#### 4.1.1.1 Abstracting Prolog Control

In a concrete execution a call to a predicate $p$ will cause each clause of $p$ to be tried in the order they occur in the program text. Each clause is executed by executing the goals in the body sequentially from left to right.

The abstract version of executing a clause is straightforward, the analyzer propagates an abstract substitution, or equivalently an abstract environment, along the clause body. The abstract environment describes the possible values of the program variables that occur in the clause. As goals are encountered abstract calls are performed and the effects incorporated into the abstract environment using an abstract version of unification.
Abstracting the clause selection rule, that is, which clauses of a predicate are executed and in what order, is more complicated. The problem stems from the use of the non-logical primitive “cut” (!). In the concrete execution, once a cut is encountered in a clause $C$ of $p$, no subsequent clauses of $p$ should be tried and no alternative solutions should be obtained for the goals earlier in the clause $C$.

A sound approximation of all the possible execution paths can be obtained by simply ignoring the cut. Some precision may be lost since the abstract interpretation will consider also some alternative clauses that would not have been encountered in a concrete execution. This is the method uses by the analyzer in this thesis.

By ignoring the cut clauses can be freely rearranged and analyzed in any order since the success patterns describe sets of substitutions (as opposed to ordered sequences of answer substitutions).

Consider the following example:

```prolog
p(X) :- number(X), !, q(X).
p(X) :- r(X).
```

If $p$ is called with a number then the number test will succeed and the cut will ensure that the second clause of $p$ is not used. The effect of this is that $r$ will never be called with an number as first argument (at least not from $p$).

Abstractly performing the call to $p$ will ignore the cut and will overapproximate the possible calls to $r$ to include numerical arguments. This is correct but imprecise.

There are a number of ways to improve the precision in the presence of cut. One possibility is to use an analyzer framework that directly handles the cut and clause selection rule [7].

Another possibility is to perform a simple preprocessing step to improve precision in the common case that the goals before the cut are simple tests, as above. The idea is that if there is a clause $C_i = "p(X) :- test(...), !, ..."$ then subsequent clauses “$p(X) :- q_0(...), q_1(...), ...$” are rewritten as “$p(X) :- not\,test(...), q_0(...), q_1(...), ...$” where $not\,test$ is the negated version of the test. The example above would then become:

```prolog
p(X) :- number(X), !, q(X).
p(X) :- not\,number(X), r(X).
```

Now the clauses can be analyzed in any order and with suitable handling of the negated test the analyzer will in many cases be able to obtain the same
4.1. Analyzer Framework

precision as if cut was handled directly. This method was used in [35, 36] and could easily be added to the analyzer used in this thesis.

4.1.1.2 Fixpoint Iteration As new call patterns $\beta^p_{call}$ become available the analyzer will analyze each of the clauses of $p$ from left to right, starting with the abstract substitution $\beta^p_{call}$.

The simulated (or symbolic) execution proceeds through the goals in a clause body, eventually the end of a clause will be reached, the abstract description of the arguments will be extracted and combined with the previous $\beta^p_{success}$.

Dependencies exist in two directions; firstly, an abstract goal $q(x)\beta$ encountered in a clause body may be such that the new abstract goal is not included in the present view on what arguments could possibly be passed to the predicate $q$; secondly, if the analyzer needs to update $\beta^p_{success}$ at the end of one of the clauses of $p$ then all predicates (or at least clauses) that call $p$ need to be reanalyzed.

To ensure that any changes to a call or success patterns are propagated to affected predicates the analyzer uses a worklist (a set of clauses) to keep track of which (clauses of) predicates need to be reanalyzed. As a special case, if, during analysis of a clause, a new call pattern occurs for some predicate $q$, then the necessary reanalysis of $q$ can be done either later, by putting (the clauses of) $q$ on the worklist, or immediately by recursively invoking the analyzer.

A widening is used when updating call and success patterns since the abstract substitution domains have infinite ascending chains.

4.1.2 The Analyzer

The global data maintained by the analyzer are as follows:

1. $WL$. Worklist of (ids of) clauses to (re-)analyze. At the start of the analysis this is initialized with the clauses of the entry point predicate. Procedure $ai\_closure$ repeatedly picks a clause off the worklist for analysis until the worklist is empty and a fixpoint has been reached.

2. $CP[p]$. Call pattern for predicate $p$. The call pattern of the entry point predicate, any undefined predicates and the builtins (see Section 4.1.3.3) are initialized to $\top$ signifying that they can be called with anything. The call pattern of all other predicates are initialized to $\bot$ signifying that they have not been called at all yet.

3. $SP[p]$. Success pattern for predicate $p$. The success pattern for the builtins are initialized as described in Section 4.1.3.3. The success pattern for undefined predicates are initialized to $\top$ to signify that
nothing is known about how they might succeed. The success pattern of all predicates defined in the program is initialized to $\bot$, signifying that none of the them have yet succeeded.

4. **Clauses**[$p$]. Static information about the clauses that belong to predicate $p$, that is $\{(p, i) | 0 < i \leq \text{number of clauses of } p\}$.

5. **Body**[$p, i$]. Static information about the body of clause $i$ of predicate $p$.

6. **Callers**[$p$] Static information about the clauses that contain a call to $p$. These are the clauses that need to be re-analyzed when a new success pattern is obtained for $p$. A set of $(p', i)$ such that clause $i$ of predicate $p'$ calls $p$.

For each predicate $p$ a single call pattern, that is, an abstract substitution over the argument variables, $(CP[p])$ is maintained that summarizes the information about all possible calls to the predicate. Similarly a single success pattern $(SP[p])$ is maintained that summarizes the possible results of $p$.

A global worklist ($WL$) contains clauses that should be analyzed. The procedure **ai_closure** repeatedly selects a clause to be analyzed and removes it from the worklist.

```
proc ai_closure:
    while $WL \neq \emptyset$
        select a clause identifier $ClauseId$ from $WL$
        ai_clause($ClauseId$)
    end while
```

When the worklist is empty then for each predicate $p$ all its clauses have been analyzed after the last change to $CP[p]$ and for each predicate $p$ all clauses that contain a call to $p$ have been analyzed after the last change to $SP[p]$.

A fixpoint has been reached and $CP[p]$ and $SP[p]$ are sound (Definition 2.2.1, page 23) approximations of the possible calls to $p$ and the possible ways $p$ may succeed respectively.

### 4.1.3 Analysis of a Clause

A clause of a predicate $p$ is analyzed when the call pattern of $p$ changes (that is, increases) and when the success pattern changes (that is, increases) for some predicate $q$ called in the body of the clause.
First the clause is removed from the worklist (if present) since the presence in the worklist is to signify that the clause should be (re-)analyzed with the present call pattern, which is just about to happen.

The body of the clause is then symbolically executed by \texttt{ai.body} to produce a success pattern $SP_{\text{clause}}$ for the clause, that is, an abstract substitution over the argument variables. If the success pattern for the predicate already includes (is less precise than) the result from the clause then nothing needs to be done. If, on the other hand, the success pattern of the clause is not already approximated by the success pattern of the predicate then this new information must be added to the success pattern of the predicate. Adding the success pattern of the clause is done by taking the upper bound and then performing a widening.

If the success pattern of $p$ changes then all clauses that call $p$ are added to the worklist so that the change eventually propagates into any clause, and thus into any predicate, that could depend on it.

For some widenings it will be advantageous to let the widening perform the upper bound. For these widenings line 4 disappears and line 5 instead becomes:

\begin{equation}
SP \downarrow \leftarrow SP[p] \lor SP_{\text{clause}}
\end{equation}

### 4.1.3.1 Analysis of Clause Body

The clause bodies are preprocessed into a simple instruction set that is then interpreted by the analyzer. The code for analyzing a clause body will be explained below.
**Experimental Setup**

```prolog
proc ai_body(Gs, β):
    if Gs is empty then β
    elif β = ⊥ then ⊥
    else
        — Gs is a list G₁ :: (G₂ :: …) of goals
        G :: G₀ = Gs
        cases
            G = call(P, ArgIndices) then
                see Section 4.1.3.2
            G = disj(Bₐ, Bₙ) then
                βₐ ← ai_body(Bₐ, β)
                βₙ ← ai_body(Bₙ, β)
                β₀ ← βₐ ∨ βₙ
            G = unif(C) then
                β₀ ← ai_unify(β, C)
            G = init(i, TermDesc) then
                β₀ ← β[i → TermDesc]
            G = kill(i) then
                β₀ ← β[i → ⟨ unused ⟩]
        end cases
        ai_body(G₀, β₀)
```

The following example clause will be used to exemplify the translation:

```prolog
nrev(c(X,Xs), Ys) :-
    nrev(Xs, XsR),
    app(XsR, c(X,nil), Ys).
```

The translation is performed as if a normal form was used where unification X = f(… ) between a variable and a nonvariable only is allowed if it is the first occurrence of the variable. This is slightly different from the normal form in Section 2.2.1. With variables renamed to the standard names \( \{X₁, \ldots \} \) the normal form of the clause and the corresponding instructions (described below) becomes:

---

2 The reader may recognize the recursive clause from naïve reverse. For typographical reasons the function symbol `c` is used instead of the usual list constructor `. `. 
When a clause is analyzed an abstract substitution, or (abstract) environment, is maintained that initially contains information about the argument variables, that is, the initial environment is equivalent to the call pattern. There are three instructions that manipulate the environment and one that corresponds to a disjunction:

1. init($i,T$). Add a new variable to the abstract substitution, denoted $\beta[i \mapsto T]$ in $ai$ above.

   This will always be applied when the current abstract substitution, or (abstract) environment, is ($\{X_1 \leftarrow T_1, \ldots, X_k \leftarrow T_k\},SVAL$) with $i = k + 1$. The type component is just extended to become $\{X_1 \leftarrow T_1, \ldots, X_k \leftarrow T_k, X_{i(k+1)} \leftarrow T\}$.

2. kill($i$). When a (non-argument) variable $X_i$ has occurred for the last time it is removed from the environment, denoted $\beta[i \mapsto \langle unused \rangle]$ in $ai$ above.

   The environment ($\{X_1 \leftarrow T_1, \ldots, X_i \leftarrow T_i, \ldots, X_k \leftarrow T_k\},SVAL$) becomes ($\{X_1 \leftarrow T_1, \ldots, X_i \leftarrow \langle unused \rangle, \ldots, X_k \leftarrow T_k\},SVAL'$) where $\langle unused \rangle$ is a special marker saying that the variable is not really present and $SVAL' = \{\{j/s, j'/p\} \in SVAL | j \neq i \wedge j' \neq i\}$, that is, all same value constraints involving the dead variable are removed.

3. unify($C$). Unify two subterms. Denoted $ai$unify($\beta,C$) in $ai$ above.
The same value constraint $C = \{i/p, j/s\}$ is added to the same value component of the environment after which the environment is normalized (Algorithm 2.3.9, page 35) to propagate the effects of the unifications. That is, if the old environment is $\beta$ then the new environment becomes normalize((TYPE$(\beta)$, SVAL$(\beta) \cup \{C\})).

4. disj$(B_a, B_b)$. Corresponding to a disjunction. The two preprocessed bodies are interpreted and the resulting environments are combined using upper bound. There are some tedious but straightforward details involved in the preprocessing step to ensure that the two environments obtained by analyzing $B_a$ and $B_b$ have the same number of variables and that if a variable is dead then it is $\langle\text{unused}\rangle$ in both environments.

The reason variables are removed from the environment is that unneeded variables affects the cost of maintaining the environment. In particular the cost of manipulating the same value component is quadratic in the number of variables in the environment.

In the actual implementation consecutive occurrences of the environment instructions are merged into one. Furthermore the init-instruction are generalized to allow the term description to have subterms as well, in the example this optimization affects the initialization of $X_8$ and variable $X_7$ would not be needed.

The last instruction call$(p, \langle i_1, \ldots, i_k \rangle)$ corresponds to a goal in the original body. The predicate $p$ are called with (the value of) $k$ distinct variables as argument. It is described below.

4.1.3.2 Analysis of Goal A goal $p(X_{i_1}, \ldots, X_{i_m})$, all $i_j$ distinct, is translated into an instruction call$(p, \langle i_1, \ldots, i_m \rangle)$ as outlined above. This is handled in three steps:

1. A call pattern $\delta$ corresponding to this call is obtained by creating an abstract substitution from the current environment $\beta$ that only involves variable $\{X_{i_1}, \ldots, X_{i_m}\}$, furthermore, the variables are renamed in the process so the first argument in the call $(X_{i_1})$ correspond to $X_1$ in the call pattern $\delta$ and so on. That is: TYPE$(\delta) = \{X_1 \leftarrow T_{i_1}^\beta, \ldots, X_m \leftarrow T_{i_m}^\beta\}$ and SVAL$(\delta) = \{(j/s_j, k/s_k) | \{i_j/s_i, i_k/s_k\} \in \text{SVAL}(\beta), 1 \leq j \leq m, 1 \leq k \leq m\}$.

2. Procedure $\text{ai\_clause}$ below is used to obtain a success pattern for this call, that is $\delta' = \text{ai\_call}(p, \delta)$.

3. The result $\delta'$ of the call is incorporated into the environment (back unification). For the abstract domains in this thesis this is treated as ordinary unification as for the instruction "unif" above.
If the success pattern is \( \bot \) then the current environment is set to \( \bot \), corresponding to a call that failed.

Otherwise, the type component of the success pattern is \( \text{TYPE}(\delta') = \{X_1 \leftarrow T_1, \ldots, X_m \leftarrow T_m\} \) and the type component of the current environment \( \beta \) is \( \text{TYPE}(\beta) = \{X_1 \leftarrow T_1^\beta, \ldots, T_n \leftarrow T_n^\beta\} \).

The environment will now be extended with the variables in the success pattern. A new type component is obtained \( \text{TYPE}' = \{X_1 \leftarrow T_1^\beta, \ldots, T_n \leftarrow T_n^\beta, X_{n+1} \leftarrow T_1, \ldots, X_{n+m} \leftarrow T_m\} \).

The same value component of the success pattern is renamed to refer to the new names of the argument variables that is: \( \text{SVAL}_{sp} = \{(i+n)/s_i, (j+n)/s_j\} \in \text{SVAL}(\delta)\).

The actual back unification is performed by same value constraints between each of the original variables \( X_{ij} \) in the call to the corresponding, now renamed, variables from the success pattern, that is: \( \text{SVAL}_{bu} = \{(i_j/e, (j+n)/e)\} \in SVAL(\delta) \).

Finally the normalization procedure (Algorithm 2.3.9, page 35) is called upon to propagate the result of the back unification, that is: \( \beta' = \text{normalize}((\text{TYPE}', \text{SVAL}(\beta) \cup SVAL_{sp} \cup SVAL_{bu})) \).

Unless the result of normalization was \( \bot \) the temporary variables \( \{X_{n+1}, \ldots, X_{n+m}\} \) will now be deleted from the environment as described for the 'kill' instruction above\(^3\).

The actual call to a predicate \( p \) from a clause \( C \) is performed by procedure \text{ai_goal}, below. If the call pattern \( CP_{call} \) is already approximated by the present call pattern \( CP[p] \) then the success pattern \( SP[p] \) is simply returned to \( \text{ai_body} \) for incorporation into the environment of the calling clause as described above. Otherwise \( CP[p] \) is updated to the new call pattern for \( p \), obtained by widening of the upper bound of \( CP_{call} \) and the old call pattern.

```
proc \text{ai_goal}(P, CP_{call});
1  \text{if } CP_{call} \subseteq CP[p] \text{ then }
2    \quad CP_{\cup} \leftarrow CP[p] \cup CP_{call}
3  \quad CP_{\cap} \leftarrow CP[p] \cap CP_{call}
4  \quad CP[p] \leftarrow CP_{\cap}
5  \text{if } \text{eager} \text{ then }
6    \quad \text{for } \text{ClauseID} \in \text{Clauses}[p] \text{ do }
7      \quad \text{ai_clause(ClauseID)}
8  \quad \text{else}
9    \quad \text{schedule(Clauses[p])}
10  \quad \text{return } SP[p]
```

\(^3\)The type component of the environment is actually implemented as a stack so this amounts to just popping off \( m \) entries.
There are two ways to make sure that the clauses of \( p \) are processed with the new call pattern. The default method used by the analyzer is to analyze the called predicate eagerly, that is, after the call pattern is updated \texttt{ai\_clause} is invoked on each clause of \( p \). When all clauses of \( p \) have been analyzed in this manner then the, possibly updated, value of \( SP[p] \) is returned to the caller, in this case \( C \).

Using eager analysis of calls means that the new information (the new call pattern) is propagated as soon as possible. Most (top-down) analyzer frameworks for Prolog appear to use eager processing of calls \cite{6, 41, 15, 32, 21}.

An alternative \textit{lazy} method is to return the current success pattern for \( p \) immediately but to put all clauses of \( p \) on the worklist. Eventually each clause of \( p \) will be removed from the worklist by \texttt{ai\_closure} and processed by \texttt{ai\_clause}. If this leads to an update of \( SP[p] \) then all clauses containing a call to \( p \), including \( C \), will eventually be reanalyzed. This ensures that also with this method the effect of the call from \( C \) will, eventually, be propagated through the clauses of \( p \) and back into \( C \).

One reason for not doing eager processing of calls is that the policy used to maintain the worklist gets more control over when clauses are analyzed. This makes it easier to observe differences between the various policies.

Eager processing of calls means that \texttt{ai\_goal} will call itself recursively (indirectly via \texttt{ai\_clause}, \texttt{ai\_body}). For some of the benchmarks this nesting can be hundreds of levels deep, a potential problem if each level prevents parts of the analyzer state from being garbage-collected.

As for \texttt{ai\_clause} it is sometimes beneficial to let the widening handle the upper bound. In this case line 2 of \texttt{ai\_goal} can be removed and line 3 becomes:

\begin{verbatim}
3  CP \perp \leftarrow CP[p] \perp CP\_call
\end{verbatim}

4.1.3.3 Analysis of Builtins Builtins are handled by providing an appropriate success pattern for each builtin. The success pattern of a builtin does not depend on the analyzed program in any way.

At the start of analysis the information for each builtin \( B \) is initialized by setting the call pattern \( CP[B] \) to \( \top \) and the success pattern \( CP[B] \) to an abstract substitution specific to the builtin. Undefined predicates are treated in the same manner (with success pattern \( \top \)). With this method calls to builtins and undefined predicates need not be handled specially during analysis, since the call pattern is \( \top \) it can never increase and since the builtin does not call any predicate there will never be a need to consider builtins when the success pattern of some predicate changes.
Some examples of (type-component of) the success patterns for builtins:

\[
\begin{align*}
SP[\text{fail}/0] &= \bot, \\
SP[\text{otherwise}/0] &= \langle \rangle, \text{i.e., } \top, \\
SP[\text{get}/1] &= \langle x_1 \leftarrow \text{num} \rangle, \\
SP[\text{is}/2] &= \langle x_1 \leftarrow \text{num}, x_2 \leftarrow \text{any} \rangle, \\
SP[\text{functor}/3] &= \langle x_1 \leftarrow \text{any}, x_2 \leftarrow \text{any}, x_3 \leftarrow \text{num} \rangle, \\
SP[\text{= ..}/2] &= \langle x_1 \leftarrow \text{any}, x_2 \leftarrow \langle \text{any}, L \rangle \rangle, \\
SP[\text{name}/2] &= \langle x_1 \leftarrow \text{any}, x_2 \leftarrow S \rangle, \\
SP[\text{list}/1] &= \langle x_1 \leftarrow L \rangle.
\end{align*}
\]

With \( L \rightarrow [\ ] \mid \langle \text{any}, L \rangle \), i.e., arbitrary list.

If explicit success types are provided through declarations or some other means then the same technique can be used to provide some precision for calls to predicates defined outside the program. At present this is done for some common library predicates such as \texttt{member/2} and \texttt{keysort/2}.

Using a fixed success pattern is a very imprecise method. As an example, the only information that can safely be drawn from a call to \texttt{member}(X, \texttt{Xs}) is that the second argument will be a list cell if the call succeeds. In particular the analyzer will not be able to infer anything more specific about the first argument even if the second argument is known.

A more precise treatment would often be possible by, in effect, inlining some builtins and handle these specially in the analyzer. As an example it would be possible to obtain quite precise results for the second and third arguments of \texttt{functor/3} given a type for the first argument. Also see below for an easily implemented method to get better precision also when using fixed success patterns.

In addition to ordinary builtins the analyzer will rewrite some control structures to ensure conservative approximation of the calls that can occur at run-time. The following are the most interesting ones, in addition \texttt{bagof}, \texttt{setof} and \texttt{on_exception} are handled in the obvious ways. Note that the handling of \texttt{findall} etc. could be significantly improved by using information from analysis of \langle \texttt{goal} \rangle to restrict the possible elements in the list of solutions.

\[\text{SP[}\text{list}/1\text{]} = \langle x_1 \leftarrow L \rangle.\]

\[\text{SP[}\text{= ..}/2\text{]} = \langle x_1 \leftarrow \text{any}, x_2 \leftarrow \langle \text{any}, L \rangle \rangle.\]

With \( L \rightarrow [\ ] \mid \langle \text{any}, L \rangle \), i.e., arbitrary list.

\[\text{SP[}\text{name}/2\text{]} = \langle x_1 \leftarrow \text{any}, x_2 \leftarrow S \rangle.\]

With \( L \rightarrow [\ ] \mid \langle \text{any}, L \rangle \), i.e., arbitrary list.

\[\text{SP[}\text{list}/1\text{]} = \langle x_1 \leftarrow L \rangle.\]

With \( L \rightarrow [\ ] \mid \langle \text{any}, L \rangle \), i.e., arbitrary list.

By also using the same value component of the success pattern it would be possible to handle the builtins \texttt{=/2} and \texttt{=/2} in the same manner. However, they are instead translated directly to \texttt{unif} instructions\textsuperscript{4}.

\textsuperscript{4}The only “builtin” with non-empty same value component is \texttt{‘C’/3}.\]
findall(⟨pattern⟩,⟨goal⟩,L) = (⟨goal⟩,fail,’_list’(L))
(⟨test⟩ -> ⟨then⟩ ; ⟨else⟩) = (⟨test⟩, ⟨then⟩ ; ⟨else⟩)
\+(⟨test⟩) = (⟨test⟩ -> fail ; true)

No attempt is made to handle explicit use of call/1. The problem with call/1 is that it can cause predicates to be called at runtime in ways that the analyzer cannot anticipate. A similar problem occurs when assert is used to add clauses with non-trivial bodies. The goals in the asserted bodies will cause predicates to be called with arguments that the analyzer knows nothing about, if the body is empty, however, then the use of assert is not a problem. At present the analyzer issues a warning if it detects the use of call/1 or unsafe uses of assert.\(^5\)

It is possible to obtain much better precision for many builtins if types such as atom, atomic, compound, int, float and nonvar (with the obvious meanings) can be represented. Adding these as separate types affects all domain operations\(^6\) and supporting them therefore represents a substantial implementation effort. A simple trick can obtain the same effect by only modifying the abstract version of the builtins without modifying any of the usual domain operations (except possibly the widening).

The idea is to pretend that all terms are “wrapped” using special unary functors that cannot occur elsewhere. As an example the program is analyzed as if an atom foo where represented as ’ATOM’(foo). The advantage with this representation is that it makes it possible to represent the information “unknown atom” by ’ATOM’(any).

This technique was not enabled in the measurements reported in this thesis.\(^7\) The reason is that this trick needs support in the implementation of some of the widenings to avoid excessive loss of precision.

4.1.3.4 Analysis of Program Entry-Point All the benchmarks are such that the entry-point is the zero arity predicate main/0. After reading in the program and setting up builtins and undefined predicates the call pattern for main/0 is set to \(\top\) and all clauses of main/0 are put on the worklist. Finally ai_closure is called to obtain a fixpoint.

The implementation actually allows multiple entry-points with arbitrary call patterns to be specified, the details are straightforward.

\(^5\)If the asserted clause has an empty body then no unknown calls can occur. The type-domain makes it easy to detect the safe uses of assert.

\(^6\)For example for a constant foo the comparison foo \(\subseteq\) atom should be true.

\(^7\)Actually the special type num is implemented in this manner by using NUM(any).


4.2 Worklist Scheduling Policies

The worklist of clauses to be analyzed can abstractly be regarded as a set of clauses with no further structure imposed. For the domains and widenings considered here it turns out that the scheduling policy, that is the order that clauses are selected from the worklist for analysis, is crucial to the efficiency of the analyzer.

The impact on analyzer efficiency of various scheduling policies was investigated by Kanamori [33]. He used a simple analyzer framework that performed a pointer analysis for a higher order functional language. He found significant differences in the number of iterations required (nodes processed to reach a fixpoint) between the best and the worst policies. Particularly interesting is that one of the best policies, least-recently-fired, does not require any information about the program (such as the call graph or its strongly connected components). He does not claim that the results are generally applicable, and indeed points out that he provides little insight into why a particular policy works well. Nevertheless his work makes it apparent that different worklist policies can significantly impact the number of iterations of an analysis, and not always in intuitive ways.

As will be discussed in Section 4.3.1 the number of iterations is not necessarily the best measure of efficiency for expensive abstract domains such as the grammar domains used in this thesis. In later chapters we will see that the worklist policy can have a large impact, not only on the number of iterations, but also on the size and associated cost of the domain elements that occur during analysis. Interestingly some settings will increase the number of iterations but decrease the cost of manipulating domain elements, and vice versa.

Several worklist scheduling policies were implemented, including one inspired by the least-recently-fired policy of Kanamori. They can be classified as fair or unfair:

**FAIR** The fair worklist policies ensure that analysis of a clause cannot be postponed indefinitely by the addition of other clauses to the worklist.

**FIFO** The worklist is a queue where the clause that have been in the worklist the longest is removed first.

**LRF** Inspired by the Least-Recently-Fired policy of Kanamori [33]. The clause that was least recently analyzed is selected first. Note that this is quite different from both LIFO and FIFO. The scheduling policy that is most fair, especially without eager analysis of goals (see Section 4.1.3.2). Kanamori found his version of LRF to require three to four times less iterations compared to FIFO or LIFO.
TWO-PHASE  A pair (Old,New) of worklists. Clauses are added to
New and removed from Old, until Old becomes empty at which
point they are switched. This is very similar to what Getzinger
used [25], he did not specify the policy used for the sub-worklists
Old and New. A FIXED worklist was used for Old and New,
fairness is nevertheless ensured by postponing reanalysis of newly
scheduled clauses until all clauses on Old have been analyzed.
This most closely corresponds to what Kanamori calls Round-
Robin (Random), where he uses a random but fixed order instead
of lexical order.

UNFAIR  These scheduling policies are unfair, that is, some clauses can
stay on the worklist a long time before eventually being analyzed.

FIXED  The worklist is implemented as an ordered set and the lex-
ically minimal element is removed first. In particular this means
that lower numbered clauses of a predicate always will be ana-
lyzed in preference to higher numbered clauses. This policy is
thus extremely unfair.

LIFO  A stack where the clause that have been in the worklist the
shortest time is removed first. This gives unfair scheduling.

4.2.1  Re-Analyzing Clauses of a Predicate

The default behavior of ai_closure is to select one clause for re-analysis
and to integrate the success pattern of this single clause with the success
pattern of the predicate, using a widening. The rationale is that the re-
resulting change to the success pattern will then be propagates as quickly as
possible, for example, if there are recursive calls in the other clauses for this
predicate.

An alternative is to remove all clauses that belong to a particular predi-
cate at the same time from the worklist, analyze them separately, combine
all their success patterns with upper bound and then, as a final step, use
a widening to incorporate the resulting success pattern with the success
pattern of the predicate.

One potential advantage with this method is that the widening operation
is the main source of imprecision and performing it less often may benefit
the precision. See Bourdoncle [3] for a discussion and algorithms to decide
few widening points.

\footnote{Also note that widening only is needed when there are cycles in the call graph, that
is, widening need not be performed for a call to \textit{q} from \textit{p} if \textit{q} cannot call \textit{p} directly or
indirectly. This possibility has not been investigated.}
A possible disadvantage is that changes to success patterns will not be propagated immediately. In particular if the first of several clauses of a predicate \( p \) would cause a change to the success pattern of \( p \) then this will not be “seen” by the subsequent clauses since it will not affect the globally visible success pattern \( SP[p] \) until after all the clauses have been analyzed. Clauses will thus be analyzed with obsolete information and this could potentially lead to more iterations.

Another possible disadvantage is that the large types mainly are the result of the upper bound operations whereas the widening operation is the main cause for types decreasing in size. Performing many upper bound operations before doing the widening may thus lead to larger types.

For simplicity this method is implemented in a somewhat roundabout way, each predicate is rewritten to single clausal form, that is, a single clause with one disjunct corresponding to each clause body. The effect is as if all clauses of the original predicate would be analyzed as soon as one clause is selected from the worklist, in effect unnecessarily analyzing also the clauses not on the worklist.

A potentially more serious effect of analyzing extra clauses is that there will be more success patterns from clauses and combining these with upper bound could lead to larger types than if only the clauses on the worklist were analyzed.

The amount of work caused by the unnecessarily analyzed clauses would cause a larger difference if typically only a small subset of the non-trivial clauses of a predicate will be on the worklist. For the synthetic benchmarks \texttt{tree} and \texttt{expr} this is not the case, all but the non-recursive clauses would be on the worklist at the same time. It is not clear to what extent it is an issue for the larger benchmarks.

Another effect is that it affects the way the number of iterations are measured. The number of iterations are defined as the number of clauses analyzed but in single clausal form there will be much fewer (but larger) clauses since several clauses in the untransformed program will be seen as a single clause in the analyzer.

### 4.2.2 The Impact of Scheduling Policy

This section briefly summarizes the impact of the methods used to schedule clauses for analysis. The details can be found in the subsequent chapters.

If only the domain independent work of the analyzer is considered, that is, the number of iterations and widenings, then the FAIR scheduling policies performs much better than the UNFAIR policies.
By default clauses are analyzed separately (not in single clausal form). When inferring the recursive types of benchmark \texttt{tree}, the UNFAIR policies typically lead to an extra factor \( n \) iterations and widenings. For the larger benchmarks the UNFAIR scheduling policies typically lead to several times more iterations and widening. The difference between the FAIR policies tend to be rather small, for this reason FIFO is the only FAIR scheduling used when analyzing the larger benchmarks in Chapter 7.

What makes the UNFAIR policies interesting is the way they interact with some of the widenings. In some cases using the unfair scheduling policy LIFO leads to more iterations and widenings but substantially smaller types, this is most noticeable for the topological clash widening in Section 5.4. For this reason the topological clash widening will be evaluated in Chapter 7 both using the fair policy FIFO and the unfair policy LIFO.

For some widenings the alternative technique of using single clausal form has a large effect on the behavior of the widenings. In particular it makes the topological clash widening (for type graphs) much more well behaved when inferring recursive types than if clauses are analyzed separately. For Janssens' widening on the other hand, single clausal form gives worse performance than when clauses are analyzed separately.

In summary, the interactions between the scheduling policy and the specific properties of the abstract domains and their widenings makes it clear that neither should be considered in isolation.

### 4.3 PERFORMANCE MEASURES

Performance of an analysis can be used to mean several, largely independent, things. One measure of performance is execution efficiency, or generally, resource usage. This can be measured in implementation dependent units such as seconds and bytes, or it can be measured in implementation independent units such as algorithmic "steps" and nonterminals of a grammar.

The second measure answers the question "how useful are the results obtained from the analysis?", the answer to which is likely to vary between applications. Typically some estimate of precision is used instead under the assumption that a more precise result should benefit any application.

#### 4.3.1 Resource Measures

One common measure of efficiency of program analyses is the (worst case) number of "iterations", where one iterations typically involves processing each part of some program representation. However, these complexity measures all have a multiplicative factor \( h \), the \textit{height} of the abstract domain, that is, the length of the longest increasing chain. Obviously this is not
directly useful for a domain with finite but unbounded increasing chains such as the grammar based domains with widening.

Even for the widenings that restrict the domain to a sub-domain with finite height, the height is typically too large to give a meaningful measure. As an example, for some of the methods in this thesis the height is exponential in the size of the program. If the worst case cost is proportional to the height of such a domain then the worst case is too bad to even consider.

Prohibitive worst case cost need not be a practical problem in itself if the programs encountering the worst case are easily identified and rare. As an example, ML type inference is, in the worst case, exponential in the size of the program. In practice however, ML type inference is linear and the worst case only happens for programs structured in an easily identifiable and uncommon way.

Both resource usage of the analyzer framework, such as number of iterations and widenings, and resource usage particular to the abstract domain, such as size of domain elements and cost of domain operations will be considered. The focus is on implementations independent measures to make the conclusions more generally applicable.

The measures that will be used are: framework specific measures such as the number of times a clause is analyzed; the size of the grammar graphs that occur during analysis; and analysis time. In many cases infeasibility of an analysis method is apparent from the huge size of the encountered types, other reasons will be discussed as they occur.

The details of each measure are as follows:

- **The number of iterations.** This is the number of times `ai_clause` is called to analyze a clause.

  When the programs are analyzed in single-clausal form then there will only be one clause for each predicate (consisting of a large disjunction of clause bodies). This means that the number of iterations when programs are analyzed in their original form is incomparable with the number of iterations when programs are analyzed in single-clausal form. This is unfortunate but not easily fixed.

  In the tables the number of iterations is denoted by “Iter”.

- **The number of widenings.** This is in practice always less than the number of iterations. The rationale for measuring the number of widenings is that iterations that lead to no changes (and thus no need for widening) will involve mainly the containment operation. The containment operation tend to be more efficient than the upper bound and widening operations.
Another reason to measure the number of widenings is that this represents work that is strictly necessary in some sense. In particular it cannot be avoided even by an analyzer that keeps better track of dependencies between changed success patterns and calling clauses.

In the tables the number of call pattern widenings and success pattern widenings are denoted by \( \nabla_{CP} \) and \( \nabla_{SP} \) respectively.

- **The number of nodes of the grammar graphs.**
  
  I measure the maximum size of any grammar graph that occurs during analysis. In practice this will almost always be the result of the upper bound, since, as explained in Section 3.2.2, the upper bound operation is the main source of large types (that is, large (representations of) grammars).

  The intermediate grammar graphs is always larger than the grammar graphs that occurs in the final call and success patterns. To illustrate these, often huge, differences the maximum size among the final grammar graphs are presented as well.

  To illustrate what would happen if upper bound and other operations on type graphs could create only minimal results the size (the number of nodes) of the largest minimized type graph encountered during analysis will be measured. This is a lower bound on the complexity of analysis, I know of no method to ensure that upper bound directly creates only minimal results.

  For type graphs the largest number of non-equivalent nodes in any encountered type graph is measured as well. This is the number of nodes that would be needed if the same type was represented as a grammar graph with a minimal number of nodes.

  In the tables the size of the largest encountered grammar graph is denoted “max size”. The size of the largest grammar graph in the result is denoted “max result”. The number of non-equivalent nodes is denoted “max \(|D|\)” (\( D \) for distinct denotations).

- **Asymptotic Costs.** Several of the widenings will cause the analyzer to encounter huge types or excessive number of iterations when used for inferring recursive types. In Chapter 5 and Chapter 6 a synthetic benchmark (tree\( _n \), Section 4.4.1) will be used to estimate how the size of the encountered types and other analysis costs varies with increasing \( n \), that is, the number of function symbols (constructors) of the recursive type.

  A rather severe abuse of notation will be used. The estimated lower, upper or tight bounds on the asymptotic cost as a function \( f \) of the number of constructors \( n \) will use the notation \( \Omega(f(n)) \), \( O(f(n)) \), and \( \Theta(f(n)) \) respectively. These bounds are almost never formally verified.
but they will only be used when the bound is “obvious” from the measurements.

- Analysis time. This is mainly for completeness. The time include only the type analysis and not the time for preprocessing the program, the analysis of uninitialized variables or for presenting the results. However, the analyzer contain code for collecting and recording measurements and also some debugging related code and this will affect the times significantly.

The implementation is written in SICStus Prolog 3#6 (fastcode), running on a Sun E3000, SunOS 5.6 with two 248MHz UltraSparc-II processors and 1.2GB memory. Times where collected using the builtin statistics(runtime,...) and does not include time for garbage collection (typically an additional 30-50%).

It should be noted that the performance model of Prolog is such that timings often are a poor indication of asymptotic complexity. This is in contrast with procedural languages like C that are sometimes \cite{6}, Page 52] chosen mainly to be able to observe asymptotic complexity from timings\footnote{So Prolog is a poor tool for measuring complexity of algorithms. On the other hand C is, in many cases, a poor tool for implementing them.}.

For Prolog the asymptotically most efficient algorithms are also sometimes impossible or awkward to implement. In addition the constant factors can be such that an asymptotically less efficient algorithm is faster for all but the largest inputs.

\subsection*{4.3.1.1 The Cost of Domain Operations} For the domains considered in this thesis the analysis cost is often dominated by the operations on the abstract domain. For simpler domains, such as those considered in Section 2.1 or traditional bit-vector domains \cite{1}, the cost of each domain operation is not only low, it is largely independent of the particular domain element.

For the grammar-based domains the domain elements, that is, the grammars, vary in size and so do the cost of manipulating them. Consider as an example that using one particular set of analyzer parameters the analysis of the tiny benchmark tree\textsubscript{6} produced a type graph consisting of over one million nodes. With other settings the size of the largest intermediate type graph was only a few hundred nodes\footnote{The final type was the same for both analyses, represented by a type graph consisting of only seven nodes.}.

For this reason even an accepted measure such as the number of iterations becomes next to useless by itself. The cost of manipulating a million node type graph can outweigh a substantial number of extra iterations if only smaller domain elements occur. This is even more so since the operations
typically have worse than linear complexity in the size of the grammar (graph).

The high and varying cost associated with the domain elements makes even “obvious” optimizations somewhat unpredictable. To illustrate this consider a case where predicate $p$ calls $q$ and both $p$ and $q$ are on the worklist since they have not been analyzed with their present call patterns ($\beta_p$ and $\beta_q$ respectively). Consider case (A) $q$ is analyzed first (using call pattern $\beta_q$), then $p$ is analyzed (using call pattern $\beta_p$), leading to update of the call pattern for $q$ obtaining $\beta_q^A$ as call pattern for $q$, which is then used to analyze $q$ again. The analysis using $\beta_q$ was thus wasted effort. Now consider case (B), $p$ is analyzed first (using call pattern $\beta_p$), leading to update of the call pattern for $q$ obtaining $\beta_q^B$ as call pattern for $q$, which is then used to analyze $q$. In case (B) $q$ is thus never analyzed with call pattern $\beta_q$, an “obvious” improvement suggesting a heuristic of analyzing calling predicates before called predicates. This and other heuristics are commonly employed in analyzer frameworks.

What is not so obvious, however, is the relative sizes, and associated costs, of $\beta_q$, $\beta_q^A$, and $\beta_q^B$ (for example the number of nodes of the type graphs in the type components). In fact it is possible that the single call pattern used in case (B) is much larger than the two call patterns $\beta_q$ and $\beta_q^A$ used in case (A). This example shows that with the kind of domains considered in this thesis many previously considered cost measures and associated optimality results and heuristics may not apply.

Effects such as these will indeed be seen for type graphs in Chapter 5, where “obviously” inferior analysis methods sometimes lead to significantly smaller and therefore less costly type graphs.

### 4.3.2 Precision Measures

If the intended application is optimizing compilers then the quality of the analysis results can roughly be taken as the quality of the code produced by the compiler. This approach was used by Getzinger [25], where the quality of the generated code was taken to be the execution speed of the optimized code.

Unfortunately this approach requires the availability of a highly optimizing compiler. An optimizing compiler for Prolog that can utilize type-information is not readily available and represents a non-trivial development effort. For this reason I have chosen to use a measure of precision (as opposed to usefulness) for the obtained analysis result.

The measurement methods that will be used are based on classifying each argument position of call patterns. The simplest method is based on modes while the more ambitious method classifies each argument into classes roughly
corresponding to the tags that would be used to encode the corresponding terms in a typical Prolog run-time system.

Not all argument positions will have the same impact on the execution speed of the compiled program. Firstly, some parts of the program will be “inner loops”, that is, predicates that stand for a relatively large part of the total execution time. Secondly, the argument positions of a predicate will have different impact on the speed of the predicate. For example, some arguments are just passed on to other predicates whereas some arguments participate in local computation.

The measures make no attempt to estimate the importance of an argument position. While this can be done, see for example [36], it would further complicate the presentation of the results and the comparison of the methods.

The precision measures used in this thesis are thus just to give a rough estimate of the relative precision obtained for the investigated widenings and should not be taken as an attempt to present the ultimate precision measure.

It would clearly be useful to have a better classification, for example a way to classify the recursive structure of arguments and not only the top-level information.

4.3.2.1 Uninitialized Arguments For argument positions that may be free variables a substitution closed domain will be unable to infer any type information at all. On the other hand, this is an inherent limitation of using an substitution closed abstract domain. For this reason it is reasonable to try to disregard such argument positions when evaluating the analysis results.

A significant fraction of such arguments consists of unaliased free variables. Intuitively an argument to a goal is such an uninitialized variable if it does not occur in preceding literals in the body or in the head.

It is possible to use an abstract domain that can represent uninitialized variables [47, 53, 52], but as with abstract versions of free variables in general this will give a domain that is not substitution closed. To preserve the substitution closedness of the type domains I chose to perform the analysis of uninitialized variables separately using a method described by Lindgren [34]. For many programs this analysis detects that more than a fourth of the arguments are uninitialized variables.

The analysis can detect that an argument position is uninitialized in some but not all calls to a predicate, intuitively corresponding to different directions (or modes) of use. Such conditionally uninitialized positions are denoted $C$ in the tables. While not true in general, these positions will typically be classified by the type analyzer as any for the same reason as the always uninitialized positions.
At present the result of the uninitialized variable analysis is not used by the ordinary analyzer. It would be possible to use the information about uninitialized variables to increase precision, in particular in conjunction with builtins such as \texttt{var/1} and \texttt{nonvar/1}. Another interesting use would be to create multiple variants of some predicates based on the conditionally uninitialized arguments.

The time to perform the uninitialized analysis is insignificant compared to the type analysis and is not included in the analysis times.

### 4.3.2.2 Modes

A simple precision measure is to consider the modes of arguments. I have chosen to classify the \textit{mode} of an argument as one of \texttt{nonvar (nv)}, \texttt{uninit (u)}, obtained from the uninitialized analysis, or \texttt{unknown (any)}. Sometimes \texttt{ground} is considered a mode as well, however, when used just as a precision measure it is less clear if this is worth the more complex presentation.

The mode precision measure, while being extremely simple-minded, has the advantage that there are hardly any optimizations that cannot benefit from knowing that an argument cannot be a variable, or that it is uninitialized.

### 4.3.2.3 Tags

To give some indication as to what type information is obtained Hentenryck et al. classified the \textit{tag} of each argument. The rationale was that “These tags will allow us to generate more efficient code by avoiding tests and specializing indexing.” [30, Page 205].

The tags they used are \texttt{NI ([], the empty list or nil)}, \texttt{CO (. ,2, the list constructor)}, \texttt{LI (list, [ ] or ./2)}, \texttt{ST (compound structures)}, \texttt{DI (atom)} and \texttt{HY (atom or compound structure)}. The article did not make it clear if they classify a list cell also as a structure and whether they classify the empty list also as an atom. They also said nothing about how numbers were classified.
The tags I used are based on those of Henriksen et al. and are as follows:

**Definition 4.3.1 (Tags)**

- **NI**  
  the empty list \( [ ] \).
- **CO**  
  the list constructor \( ./2 \).
- **LI**  
  union of NI and CO.
- **ST**  
  compound structure, including list constructor.
- **DI**  
  atom, including the empty list.
- **HY**  
  union of ST and DI.
- **NUM**  
  a number
- **UNINIT**  
  an uninitialized variable, as determined by the separate analysis of Section 4.3.2.1.
- **ANY**  
  unknown argument

Suppose, for some application, the tag-classification or modes are the only usable information about the arguments. Why not use the tags as an abstract domain, something that is indeed possible and likely to give a quite efficient analysis? The reason is that a more precise domain will avoid loss of precision during analysis, with the effect that the ultimate result will be more precise, even if it is at that point grossly over-approximated as when summarized by the tags or modes above.

### 4.4 BENCHMARKS

Selecting a set of benchmarks on which to evaluate the performance of the analyzer is a difficult task. The benchmarks should be a representative sample of “realistic” programs. It is also preferable if the benchmarks are readily available and have been used by other researchers so that results can be compared.

An obvious approach is to use a set of benchmarks that have already been used to evaluate other analyses, since one might expect that the suitability of the benchmarks has then already been established. In the following sections I will consider two such sets of benchmarks, Aquarius [26] and GAIA [6].

Unfortunately, I found these benchmarks to be lacking in several respects. A common property of these benchmarks is that they consist of a “realistic” program but there is also an entry point that exercise the program with some sample input data. The problem is that the sample input is present in the program and it is thus available to the analyzer. For this reason the analyzer will be able to infer quite precise information about the arguments to the rest of the benchmark program.
The presence of input data as part of the programs is unrealistic and it will increase the precision of the analysis artificially. For the Aquarius benchmarks I will propose how to make the programs more realistic by making the input data unavailable to the analyzer. For the second set of benchmarks, the GAIA benchmarks, I will argue that this and other deficiencies are so large that any analysis results would be meaningless.

4.4.1 Synthetic Benchmarks

The synthetic benchmarks are constructed to expose particular properties of various analysis parameters. Their predicates will always have call pattern \texttt{any} \footnote{The call from the entry point is with an unbound variable which correspond to an unknown argument for the substitution closed abstract domains considered here.}, so the analysis of these benchmarks will only require widening of success patterns.

The first synthetic benchmark \texttt{tree}_n, is a simple predicate that recognizes unary trees with \(n - 1\) different internal (unary) nodes and one kind of leaf, benchmark \texttt{tree}_4 is shown below. Benchmark \texttt{tree}_n is the essence of a predicate that traverse a recursively defined data structure. Data structures and program fragments similar to these are present in several of the benchmarks. By instantiating this benchmark for increasing values of \(n\) we can expose weaknesses in certain widenings and other analyzer parameters.

For \texttt{tree}_4 the success type for the single argument is \(T \rightarrow a | b(T) | c(T) | d(T)\), a grammar that would have one \texttt{or}-node and \(n\) \texttt{functor}-nodes both as a type graph and as a grammar graph.

\begin{verbatim}
main :- tree(_Any).

tree(a).

endwith tree(b(T)) :- tree(T).
endwith tree(c(T)) :- tree(T).
endwith tree(d(T)) :- tree(T).
\end{verbatim}

A larger benchmark, somewhat similar to \texttt{tree}_n is \texttt{expr}, Figure 4.1. It originated as a predicate to recognize the arithmetic expressions in the second argument to the Prolog builtin \texttt{is/2}.

Something similar to \texttt{expr} would be part of any compiler or interpreter for Prolog. As we will see, for many choices of analyzer parameters, the analysis of \texttt{expr} is infeasible.

The final synthetic benchmark, Figure 4.2, manipulates nested lists, it will be used throughout the thesis to exemplify how various widenings are able
4.4. Benchmarks

main :- expr(Any).

expr(X) :- X = 'NUMBER'(\).
expr(+X) :- expr(X).
expr(-X) :- expr(X).
expr(+X,Y) :- expr(X), expr(Y).
expr(-X,Y) :- expr(X), expr(Y).
expr(*X,Y) :- expr(X), expr(Y).
expr(/X,Y) :- expr(X), expr(Y).
expr(integer(X)) :- expr(X).
expr(float(X)) :- expr(X).
expr(#X,Y) :- expr(X), expr(Y).
expr<<(X,Y)) :- expr(X), expr(Y).
expr>>(X,Y)) :- expr(X), expr(Y).
expr([X]) :- expr(X).

T  \rightarrow  \text{NUMBER'(any)}
T  \rightarrow  +(T)
T  \rightarrow  -(T)
T  \rightarrow  +(T,T)
T  \rightarrow  -(T,T)
T  \rightarrow  *(T,T)
T  \rightarrow  /(T,T)
T  \rightarrow  m(T)
T  \rightarrow  i(T,T)
T  \rightarrow  n(T)
T  \rightarrow  s<(T,T)
T  \rightarrow  s>>(T,T)
T  \rightarrow  (T,[])

Figure 4.1: Benchmark expr recognizing arithmetic expressions. The success pattern will have T as the grammar describing the type of the argument.

main :- list_of_lists(Any).

list_of_lists([]).
list_of_lists([Ls]) :- int_list(L),
list_of_lists(Ls).
int_list([]).
int_list([Xs]) :- T, T, num, T

T  \rightarrow  []
T  \rightarrow  (T, T)
T  \rightarrow  [T, T]
T  \rightarrow  ,num, T)

Figure 4.2: Benchmark recognizing lists of integer lists. The optimal success pattern for list_of_lists/1 will have T as the grammar describing the type of the argument.
to keep apart the different types for the surrounding list and the element lists.

### 4.4.2 Aquarius Benchmarks

The Aquarius set of benchmarks [26], Figure 4.1, was used by Van Roy et al. in evaluating the performance of the Aquarius Prolog compiler [47, 48]. Some of them were also used by Taylor to evaluate his Prolog compiler, Parma [53]. Getzinger used these (and other) benchmarks to evaluate the impact of various abstract domains on the code produced by the Aquarius compiler [25]. The Aquarius benchmarks are classified as “small” and “large”, only the larger benchmarks are considered here (most of these are quite small too).

The size of each benchmark appears in Table 4.2. The size measures only include those predicates and goals that are reachable from the main entry-point and that are defined in the program. Thus calls to builtins and undefined predicates in a body will not be counted as a goal and the number of arguments only include arguments of predicates reachable from main/0.

This set of benchmarks have several attractive features,

- They are readily available. This makes it easy for other researchers to compare their results with those obtained in this thesis.
4.4. Benchmarks

| Name          | Preds | Cl  | Goals | Args | \(|F_0| | \(|F_{>1}| | Uninit |
|---------------|-------|-----|-------|------|------|------|-------|
| crypt         | 9     | 27  | 29    | 18   | 4    | 5    | 0%    |
| meta_qsort    | 8     | 26  | 18    | 10   | 8    | 8    | 30%   |
| prover        | 10    | 33  | 28    | 22   | 9    | 6    | 27%   |
| browse        | 14    | 29  | 29    | 42   | 7    | 7    | 24%   |
| unify         | 29    | 63  | 79    | 141  | 12   | 15   | 18%   |
| flatten       | 28    | 58  | 55    | 83   | 10   | 14   | 27%   |
| sdda          | 28    | 77  | 69    | 78   | 26   | 10   | 21%   |
| reducer       | 30    | 95  | 83    | 95   | 35   | 19   | 18%   |
| boyer         | 24    | 134 | 34    | 61   | 15   | 91   | 28%   |
| simple_analyzer | 67  | 136 | 140   | 254  | 19   | 25   | 24%   |
| nand          | 40    | 136 | 189   | 174  | 19   | 4    | 26%   |
| chat_parser   | 155   | 494 | 345   | 742  | 273  | 44   | 44%   |
| chat_big      | 412   | 2715| 926   | 1601 | 855  | 129  | 33%   |

Table 4.2: Size of Aquarius benchmark programs. The number of predicates, clauses, goals and predicate arguments refer only to calls and definitions of predicates that are both reachable and defined in the benchmark. The number of distinct non-numerical constants \(|F_0|\) and non-constant function symbols \(|F_{>1}|\) are counted over all predicates of the programs. The uninitialized arguments refer only to predicates actually reached during analysis (often the same as, but in general a subset of, the reachable predicates).

- They have been used previously and appear to be accepted by the research community.
- They are executable and all of them come with suitable input data. This has two benefits. Firstly, it gives a rudimentary verification that the programs are realistic and that previous benchmarks have not made analyzer-specific or domain-specific modifications that could make them unsuitable for the analyses considered in this thesis. Secondly, it would make it possible to obtain statistics of the run-time behavior to better estimate the impact of the precision obtained.
- Some of the benchmarks are of non-trivial size.

There is, however, one major problem with these benchmarks that make them less appropriate for the purpose of evaluating analyses. All but one of the benchmarks (chat_big) have the input data available as part of the program (as opposed to reading it from a file or some such). The input data is present in order to provide an entry point that can exercise the “real” program, and would not be present in a realistic use of these programs.

The availability of the input data will increase the precision obtainable through analysis for almost any abstract domain. In particular it will sig-
main :-
    init_state(0, NumVars, NumGs, Bounds, Gs),
    benchmark(NumVars, NumGs, Bounds, Gs).

benchmark(NumVars, NumGs, Bounds, Gs) :- ...

init_state(0, 2, 3, 100, [ 
    function(2, [1,2], [0,3], [], [], [], [], []),
    function(1, [2,3], [0,1], [], [], [], [], []),
    function(0, [1,3], [0,2], [], [], [], [], []))

init_state(1, ...).

Figure 4.3: Original code for benchmark nand

significantly increase the obtained precision using the type domains considered in this thesis. Since the presence of the input data is not realistic\footnote{There are cases when the input data might realistically be available, for example crypt that solves a “one-shot” problem.} the increase in precision will be artificial.

To simulate a use where the input comes from outside the program, be it a file or some other unknown part of a larger application, a version was written for each benchmark which reads the input from a stream. As could be expected this will significantly decrease the precision of the analysis, see Section 7.1.2.

It can be argued that a realistic program would not use \texttt{read/1} directly but rather use some external data-format that are parsed and translated to some internal format or at least verified before being used by the program. For this reason some of the benchmarks were modified to read and then perform simple validation of the input data. Analysis of the validation code will enable the analyzer to extract some information about the input data and one could expect the precision obtained for this version of a benchmark to lie somewhere in-between the precision for the original version and the version that reads the input data without validation.

To illustrate these issues consider the original code for the entrypoint \texttt{main/0} of the benchmark \texttt{nand} (Figure 4.3). The input of the program consists mainly of the list \texttt{Gs}, the last argument from \texttt{init_state/4}. Here the analyzer will be able to obtain quite precise information about this list. With a numerical domain that can represent numerical constants the analyzer would even be able to conclude that only the first clause of \texttt{init_state/4}
% main_read :-
    read(in_data(NumVars, NumGs, Bounds, Gs)),
    benchmark(NumVars, NumGs, Bounds, Gs).

main_validate :-
    read(in_data(NumVars, NumGs, Bounds, Gs)),
    validate(NumVars, NumGs, Bounds, Gs),
    benchmark(NumVars, NumGs, Bounds, Gs).

validate(NumVars, NumGs, Bounds, Gs) :-
    integer(NumVars), NumVars>=0,
    integer(NumGs), NumGs>=0,
    integer(Bounds), Bounds>=0,
    validate_gs(Gs, 0, NumGs).

validate_gs([], NumGs, NumGs).
validate_gs([G|Gs], NG0, NumGs) :-
    validate_function(G),
    NG1 is NG0+1,
    validate_gs(Gs, NG1, NumGs).

validate_function(F) :-
    F = function(Num, True, False, CIn, IPred, ISucc, Pred, Succ),
    list_of_naturals(True), list_of_naturals(False),
    list_of_naturals(CIn), list_of_naturals(IPred),
    list_of_naturals(ISucc), list_of_naturals(Pred), list_of_naturals(Succ).

list_of_naturals([]).
list_of_naturals([X|Xs]) :-
    integer(X),
    X >= 0,
    list_of_naturals(Xs).

Figure 4.4: More realistic code for benchmark nand
applies and the analysis could proceed with precise knowledge about the input data, a quite unrealistic scenario.

The modified code, Figure 4.4, contains two entry-points, main_read/0 and main_validate/0, that read the input from some external source instead. The difference between them is that main_validate/0 will call the validation predicate validate/4 and during analysis this will give the analyzer some information about the arguments that are then passed on to the rest of the program via benchmark/4. In particular the analyzer, given a suitable widening, will be able to obtain a, still quite precise, description for the argument Gs to benchmark/4, such as T in the following grammar,

\[
\begin{align*}
T & \rightarrow [] \mid (F, T) \\
F & \rightarrow f(\text{num}, L, L, L, L, L, L, L, L) \\
L & \rightarrow [] \mid (\text{num}, L)
\end{align*}
\]

The modified versions of the Aquarius benchmarks will be used in Section 7.1.2 to illustrate that the unmodified benchmarks give significantly increased and therefore unrealistically good absolute precision. However, in this thesis the benchmarks will be used to compare precision and the absolute precision is less significant. For this reasons the unmodified benchmarks will be used in the comparisons of precision and performance in Chapter 7 and Appendix A.

4.4.2.1 Benchmark Specifics  The benchmarks boyer and meta_qsort have a structure that will make analysis imprecise, regardless of the widening used.

For boyer the reason is that it uses functor/3 and arg/3 to fill in a partially instantiated term. With an substitution closed domain and with such a simple-minded treatment of builtins as used in the analyzer this is bound to be imprecise, regardless of the widening used.

The main data structure manipulated by boyer is recursively defined with a large number of non-constant functors. However, since all precision is lost this does not cause a problem for any of the abstract domains.

The entry-point and main loop of meta_qsort has the following structure:

main :- loop((top-level goal)).

loop(Xs) :-
    step(Xs, Ys),
    ( nonvar(Ys) ->
The argument to \texttt{loop} will always have the same structure as when called from \texttt{main}. However, the “continuation” \texttt{Ys} can be a free variable to signal that \texttt{loop} should \textit{not} be called. For an substitution closed domain the success-pattern for the second argument of \texttt{step} will have to be \texttt{any}, and so the recursive call to \texttt{loop} will get \texttt{any} as argument and all precision is lost\footnote{By a more precise treatment of the built-in \texttt{nonvar}, e.g., as outlined in Section 4.1.3.3, it would be possible to infer that \texttt{loop} never will be called with a variable even though no other type information could be obtained.}.

The structure of the predicate that corresponds to \texttt{step} has a structure similar to the benchmark \texttt{expr} but fewer non-constant functors (four). However, since all precision is lost the recursive data structure is not a problem for any of the abstract domains or widenings.

### 4.4.3 GAIA Benchmarks

The GAIA benchmarks are available from Brown University\footnote{\texttt{ftp://ftp.cs.brown.edu/pub/fix.tar.Z}}. They were used in the evaluation of the GAIA abstract interpretation framework [6]. These benchmarks have also been used by other researchers [22, 14, 49, 2, 21].

This set of benchmarks was used by Van Hentenryck, Cortesi and Le Charlier to evaluate their widening for type graphs [30]. For this reason it would be natural to use the GAIA benchmarks also in this thesis so that the results could be compared\footnote{I sent an email to the authors of [30] asking about the set of benchmarks used in their article. I got a reply (email from Cortesi 970324) with the same programs as those available from Brown University.}. However, the GAIA benchmarks have deficiencies that make them unsuitable as benchmarks.

There are several kinds of problems: ordinary “bugs” such as period used instead of comma, possibly these are transcription errors; presence of input data to exercise the benchmark; intentional modifications to fit a particular domain; modifications that make the programs unsuitable for other domains and no longer executable; explicit definitions of some builtins such that the precision will be artificially high.

One problem with the GAIA benchmarks is that they contain a number of errors. For example, the predicate name is misspelled in some clauses. This causes some predicates to lack a non-recursive clause, enabling the analyzer to (correctly) infer that the predicates cannot succeed. Other errors
are: use of syntactic inequality ($\triangleq$ instead of arithmetic inequality ($=$))
and in one case a period is used instead of comma, leading to an extraneous variable-only non-recursive clause instead of the intended recursive call. Some of these errors will affect analysis for any abstract domain.

As for the Aquarius benchmarks these benchmarks contain entry points that exercise the main parts of the program with sample input data and the input data is available as part of the program. As argued above this is an unrealistic situation.

A second class of problems are caused by, what appears to be, modifications to the original executable versions of the programs to make them suitable for analysis by a simpler (groundness?) domain. In particular some benchmarks have input data such that the program would fail immediately. Presumably the input data in the benchmarks have the same groundness properties as valid input data. For these cases the type analysis will detect that the program fails before most of the predicates are reached. As an example consider the following somewhat simplified fragment of the GAIA benchmark kalahr.p,

```prolog
play(Game,Result) :-
    initialize(Game,Position,Player),
    displaygame(Position,Player),
    play(Position,Player,Result).

displaygame(Position,computer) :-
    show(Position).

show(board(H,K,Y,L)) :-
    reverse(H,Hr),
    ...

initialize(kalah,board(a,0,a,0),opponent).
initialize(kalah,toto(b,1,b,1),computer).16
initialize(kalah,board(c,2,c,2),computer).
```

The analyzer will determine that the call to the list reversal predicate \texttt{reverse/2} will fail when the first argument is one of the atoms from \texttt{initialize/3}, analysis will correctly conclude that the playing part of the program can never be reached.

---

16This clause does not make sense at all, the function symbol \texttt{toto/4} does not appear elsewhere in the program.
The final class of problems comes from explicit definitions of some builtins such as `arg/3` and `functor/3`. Providing such definitions adapted to a particular analyzer or domain is problematic in itself if the definitions remain when evaluating with other analyzers or domains. A more serious deficiency with the explicit definitions as they appear in the GAIA benchmarks is that they are not independent of the analyzed program. This will artificially improve the precision of the analysis. As an example the following explicit definitions of `compound/1` and `constant/1` appears in the GAIA benchmark `press1.pl`,

```prolog
compound(X * Y).
compound(X - Y).
compound(X + Y).
compound(X / Y).
compound(X \ Y).
compound(X = Y).
constant(1).
constant(0).
compound(x).
compound(y).
```

Presumably the two clauses `compound(x)` and `compound(y)` were really intended to be `constant(x)` and `constant(y)`. The success pattern for `compound/1` would then have the type `T` as the type of the argument:

\[
T \rightarrow *\text{(any,any)} \\
T \rightarrow -\text{(any,any)} \\
T \rightarrow +\text{(any,any)} \\
T \rightarrow /\text{(any,any)} \\
T \rightarrow ^\text{(any,any)} \\
T \rightarrow =\text{(any,any)}
\]

Since the type domains used in this thesis do not distinguish between numerical constant the success type for `constant` would be:

\[
T' \rightarrow x \mid y \mid \text{num}
\]

Consider now what will happen when the analyzer encounters a call with unknown argument to `compound/1` and `constant/1`, for example:

```prolog
main :- ..., read(X), p(X).
p(X) :- compound(X), q(X).
p(X) :- constant(X), r(X).
```
With the above types for `compound/1` and `constant/1` the analyzer will be able to conclude that, even though the argument to `p` is unknown the argument to `q` will be one of the functors `*`, `-`, `+`, `^`, and `=`. It will also be able to conclude that `r` could only be called with a number or one of the constants `x` and `y`.

This result is clearly unrealistically precise considering that the analyzer cannot know anything about what is returned from the goal `read(X)`. The precision can be affected also for cases when the arguments are not completely unknown and so, in general, defining the primitives in such a program dependent way will give artificially high precision.

In summary, I found several kinds of problems with the GAIA benchmarks: ordinary programming errors, presence of input data, domain specific modifications, and explicit and too precise definitions of builtins. For these reasons I consider it meaningless to evaluate a domain as precise as type graphs using the GAIA benchmarks in their present form. Since the primary reason for considering these benchmarks would be to enable a comparison with the results published in [30] there is not much point in modifying the benchmarks to remove the problems. I will not consider the GAIA benchmarks further.

### 4.4.4 Other Large Benchmarks

To complement the few large Aquarius benchmarks I chose to include some larger programs obtained from various sources. Their common characteristics are that they are large, realistic programs and not written specifically to become benchmarks\(^\[17\]\). The size information appear in Table 4.3. They are only briefly described here, all benchmarks and more detailed comments about any modifications are available from the author.

- **aquarius_compiler**
  
  This large program is a standalone version of the Aquarius Prolog compiler. This is a sizeable and realistic program.

---

\(^{17}\)But see a remark at benchmark `kishandi`.
This program contains a predicate for compiling arithmetic expressions that manipulates a recursively defined data structure similar to the data structures manipulated by the synthetic benchmark \textit{expr}. However, the structure of the predicate would require better handling of variables and aliasing for the analyzer to extract the appropriate recursive type. With the abstract domains used in this thesis all information is lost. For this reason the predicate does not cause a performance problem even for the widenings that are inefficient when inferring recursive types.

The Aquarius compiler was also used by Fecht [21], however, the version distributed with his analyzer framework contains a bug\textsuperscript{18} that leads to artificially high precision for any domain and enables the analyzer used in this thesis to conclude that less than 300 of the more than 1400 predicates could ever be reached. For this reason a version obtained from Van Roy’s homepage\textsuperscript{19} is used instead.

- \textit{kish\_andi}

  A parallelism simulator written by Kish Shen. I have no further information about this program. The entry-point used for analysis is intended for benchmarking and gives some artificial increase in precision. The improvement in precision is marginal.

  This program contains a predicate (an expression evaluator) that manipulates a recursively defined datastructure. The structure of the predicate is similar to the synthetic benchmark \textit{expr} and will cause problem for several of the widenings and abstract domains.

- \textit{symbolic\_1}

  Apparently a simulator for a Prolog processor, origin unknown, comments in the code indicate that it may be related to the Aquarius project.

  This benchmark contains a predicate (an expression simplifier) with a structure simulator to the synthetic benchmarks \textit{expr}. However all precision is lost due to the poor handling of builtins and variables in the analyzer. For this reason the predicate never causes a performance problem.

- \textit{tricia}

  A prolog compiler for Tricia Prolog developed at the department.

  Manipulates recursively defined data (like synthetic benchmark \textit{expr}) but it is not clear if even an abstract domain that handles variables and builtins better would be able to detect this.

\textsuperscript{18}A version of \texttt{expand_term} that fails for ordinary clauses.

\textsuperscript{19}http://www.info.ucl.ac.be/people/PVR/
- reader
  The portable reader and tokenizer by R.A. O’Keefe and D.H.D. Warren. A classical benchmarks, a number of versions of this has been used as benchmark by other researchers. This version is the concatenation of the files read and rdtok.gen in the Edinburgh tools archive\textsuperscript{20}. Does not manipulate recursively defined data structures except lists (of numbers mostly).

- bamspec
  Executable specification for the Berkeley Abstract Machine, used by the Aquarius project. Does not appear to use recursively defined data structures with many function symbols.

### 4.5 IMPLEMENTATION OF METHODS

The main part of this thesis consists of an investigation of the execution performance of widenings, and to a lesser extent their precision characteristics and the performance of other operations, for grammar-based abstract domains. To evaluate previously proposed operations I have implemented the operations from the descriptions in the literature. Often I adapted the descriptions and thus implementation from the original description. Such adaptations have been performed for the following reasons:

- Concretization. Often a method has been described at a rather high level and many choices are not specified in sufficient detail to allow a direct implementation. As an example several methods use non-deterministic choice using formulations such as “select and remove an element $x$ from the set $S$ and traverse/process/...$ x$”. All these details must be decided and in the cases where they appear to affect efficiency or precision I try to provide insights into the issues involved. Some of the methods have also been described using certain restrictions on the input to the widenings. Where advantageous I have tried to remove such restrictions. The type graph widening of Henenryck et al.\textsuperscript{[30]} (Section 5.4), for example, puts constraints on the shape of the involved type graphs that potentially require exponentially more nodes to represent a particular type graph.

- Corrections. All the previously proposed methods investigated in this thesis contain mistakes in the algorithms or their correctness proofs, in particular mistakes involving the termination properties of the proposed widenings. I have corrected such mistakes when possible, and discuss the issues involved.

\textsuperscript{20}ftp://src.doc.ic.ac.uk/packages/prolog-pd-software/tools.tar.Z
- Heuristics and enhancements. In most cases the methods do not perform very well in their basic form, particularly with respect to efficiency. In these cases I have tried to investigate the reasons for this and to suggest and implement enhancements, often of a heuristic nature.

- Finally, I have investigated some widenings originally formulated on other grammar based domains than those implemented in the analyzer. The methods have then been reformulated to obtain methods closely related but not directly comparable to the originals.
Methods using Type Graphs

Recall that a widening is needed to ensure that the analysis terminates since, for the grammar based type domains, there are infinite chains of successively less precise types. The widening and any requirement it imposes on the grammar representation is the most important component of an analyzer that use grammars as type domain. If the widening is too aggressive in its approximation then the resulting analysis will be imprecise. If it is not aggressive enough then the analysis may become too inefficient, both in terms of iterations and as we will see, more importantly, in terms of the sizes of the encountered grammar representations.

Another important aspect is to what extent nominally domain independent properties of the analyzer will affect the efficiency when using a particular widening. We will see several examples of such interactions where, for example, single clausal form (Section 4.2.1) makes an analysis require type graphs linear in the size of the program whereas the default method where each clause is independently analyzed makes for exponentially large type graphs. Similar interactions will also be seen between the widening and the worklist scheduling policy (Section 4.2).

In this chapter methods, and in particular widenings, will be investigated that use type graphs to represent grammars. Each widening will first be described. This includes any adaptations or corrections used.

The performance when inferring the type of a recursive data structure will then be studied using the synthetic benchmarks tree and expr described in Section 4.4. In many cases the performance is unacceptable when recursive types are inferred. The reason for this can be found both in the representation of term grammars as type graphs and in the widenings.

The precision and performance of the widenings on “real” programs will be studied in Chapter 7.
Janssens and Bruynooghe [31, 32] introduced type graphs and a widening based on the idea to limit the number of occurrences of a function symbol along a forward path. When this constraint is violated nodes are combined and backward arcs introduced until the constraint is satisfied and the resulting type graph is a safe approximation of the original type graph. That this gives stationary chains is easily shown since the widening, in effect, limits the maximum number of nodes of a type graph, and thus maps the type graphs onto a finite subdomain. This widening is studied in Section 5.3.

Van Henntenck, Cortesi and Le Charlier [30] proposed an alternative widening that compares the type graphs, representing the old and new approximation of a call or success pattern, to see where they differ. These differences, what they call topological clashes, are then used to guide a transformation of the type graph. This is the only grammar widening studied in this thesis that makes use of the previous approximation to steer the widening. This makes the proof of the stationary property more involved. The topological clash widening is studied in Section 5.4. An alternative formulation that constructs the required type graph directly is described in Section 6.6.1.

To give a baseline for efficiency and precision two simpler widenings are presented as well. The first one, functors, is a very simple widening that, roughly, creates a single or-node with one child for each functor in the type graph. Each functor-node is then created with a backward-arc to the root or-node. This widening is described in Section 5.1.

The second of the simpler widenings is based on the concept of a type-jungle [36], essentially a grammar graph with at most one node for each function symbol. Algorithm 3.3.6 (construct) is then used to obtain a proper type graph. The type-jungle widening is described in Section 5.2.

The chapter ends with a summary and conclusions in Section 5.4.6.

5.1 Functors Widening

The functors widening is perhaps the simplest imaginable widening that still maintains some information about the recursive structure. The idea is to create an or-node with one child for each function symbol in the original type graph. The outgoing arcs from these functor nodes are then backward arcs to the root or-node.

5.1.1 Definition

The widening just passes the (upper bound of) the argument types to procedure functors (Algorithm 5.1.2) that will then create an type graph with an or-node as root as outlined above.
Definition 5.1.1 (Functors Widening)

\[ T \lor T' = \begin{cases} T & \text{if } T' \subseteq T \\ \text{functors}(T \cup T') & \text{otherwise, Algorithm } \text{5.1.2} \end{cases} \]

The functors algorithm creates an or-node \( r \) that will be a safe approximation of every node, except the any-nodes, in the original type graph \( T \). Each successor of \( r \), a functor-node with label \( f \), should be a safe approximation of the nodes in \( T_{if} \), that is, the nodes labeled \( f \) in the original type graph \( T \). Since \( r \) is a safe approximation of every node in \( T \) except any-nodes, \( r \) will safely approximate the \( i \)'th successor of every node labeled \( f \) in \( T \) unless one of the nodes with label \( f \) have an any-node as successor \( i \).

Algorithm 5.1.2 (Functors Widening) \text{functors}(T):

1. if \( T = \text{any} \) then return any
2. create an or-node \( r \)
3. for each \( f \in \mathcal{F} \) such that \( T_{if} \neq \emptyset \)
   1. create a node \( n \) with \( \text{lb}(n) = f \)
   2. for \( i \in [1, \text{arity}(f)] \)
      1. if \( \text{any} \in T_{if} \downarrow i \) then
         1. \( n \downarrow i := \text{any} \)
      else
         1. \( n \downarrow i := r \)
   
4. \( \text{compact the result} \)
5. if \( \text{prnd}(r) = \{r'\} \) then return \( r' \) else return \( r \)

The algorithm maps any grammar graph onto a finite set of type graphs since the size of the result is limited by the number of function symbols, which is finite.

Algorithm 5.1.2 will produce a safe approximation of the input type graphs. The algorithm does not, however, produce the most precise result. In fact, a most precise result does not even exist in general.

Consider the type graph \( T \) of Figure 5.1, here Algorithm 5.1.2 will produce \( T_1 \), but both \( T_2 \) and \( T_3 \) are more precise than \( T_1 \) while being incomparable to each other. What happens is that some nodes in the original type graph are reached through argument positions that will refer to any-nodes in the new type graph.
Figure 5.1: No most precise type graph exists for the functors widening.

$T_1$ is functors($T$) but $T_2$ and $T_3$ are more precise but incomparable.
Methods using Type Graphs

Methods using Type Graphs

<table>
<thead>
<tr>
<th></th>
<th>Iter</th>
<th>$\forall \delta_P$</th>
<th>max size</th>
<th>max result</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree4</td>
<td>9</td>
<td>5</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>tree5</td>
<td>11</td>
<td>6</td>
<td>23</td>
<td>6</td>
</tr>
<tr>
<td>tree6</td>
<td>13</td>
<td>7</td>
<td>33</td>
<td>7</td>
</tr>
<tr>
<td>tree7</td>
<td>15</td>
<td>8</td>
<td>45</td>
<td>8</td>
</tr>
<tr>
<td>tree8</td>
<td>17</td>
<td>9</td>
<td>59</td>
<td>9</td>
</tr>
<tr>
<td>tree_n</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>expr</td>
<td>31</td>
<td>16</td>
<td>364</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of using type graphs with widening functors on the synthetic benchmarks. FIFO worklist scheduling. Number of iterations (Iter), number of widenings of success patterns ($\forall \delta_P$), maximum number of nodes in any intermediate type graph (max size), largest type graph occurring in a final call or success-pattern (max result).

5.1.2 Performance

The results for the synthetic benchmarks are summarized in Table 5.1. The results are similar for all worklist scheduling policies\(^1\).

The functors widening gives optimal result in negligible time for the synthetic benchmarks, tree_n and expr. This may seem unremarkable but we will see later that these benchmarks can require surprisingly large amount of resources for some widenings and analyzer parameters. Note however that for tree_n, even with this imprecise widening, the maximum size of intermediate type graphs grows quadratically with n.

5.1.3 Precision

Interactions between the analyzer framework and the domains and widenings will sometimes result in more precise results for a less precise widening than for a more precise widening. To give a rough indication of the precision properties of each widening predicate list_of_lists (Figure 4.2, page 91) will be used as a running example. For this predicate the type component of the success pattern should ideally denote a list of lists of numbers, that is

$$T_{ll} \rightarrow [\cdot] \mid (T_i, T_{ll})$$

$$T_i \rightarrow [\cdot] \mid \text{num}, T_i$$

The functors widening is the least precise of all the widenings considered in this thesis. It will result in the type

$$T \rightarrow [\cdot] \mid \text{num} \mid (T, T)$$

\(^1\)The only difference is the absolute, but not asymptotic, number of iterations.
5.2 TYPE-JUNGLER WIDENING

5.2.1 Description

A Type-Jungle [36] is a grammar graph with at most one node for each function symbol. It was originally proposed as a finite type-domain that permits a particularly efficient implementation of the domain operations, in particular upper bound.

Here it will instead be used as a finite subdomain of grammar graphs to provide a widening by converting a type graph (or grammar graph) into the finite domain of type-jungles and then back to a type graph.

A type-jungle differs from the result of the functors widening of the previous section in that there can be more than one or-node, it is thus strictly more precise than the functors widening. As an example consider a type \( T \) describing trees, where each tree-node is either a leaf or the node has some datum of type \( D \) and a list \( TL \) of children:

\[
\begin{align*}
T & \rightarrow \text{leaf} \mid t(D, TL) \\
D & \rightarrow \text{any} \\
TL & \rightarrow [] \mid .(T, TL)
\end{align*}
\]

This is already a type-jungle whereas the functors widening would give the less precise type \( T' \) below with a concretization that also include nonsensical terms such as \( .([], t(\omega, \text{leaf})) \).

\[
\begin{align*}
T' & \rightarrow \text{leaf} \mid t(\text{any}, T') \mid [] \mid .(T', T')
\end{align*}
\]

5.2.2 Definition

The type-jungle widening uses a nonmonotone restriction operation (jungle, Algorithm 5.2.2) to map a type graph, or any grammar graph, onto the finite subdomain of type-jungles\(^2\).

The result is a grammar graph\(^3\) with at most one functor-node for each function symbol.

\(^2\)As for the functors widening there is no most precise type-jungle in general. In fact the example in Figure 5.1 illustrates this also for the type-jungle widening.

\(^3\)There is little reason to use type graphs instead of grammar graphs when this widening is used. The reason this is done here is that a simple and deterministic widening such as this makes it easier to study the effects of the scheduling policies and the type graph representation. The performance and precision when using the type-jungle widening with minimized grammar graphs will be evaluated in Section 6.3 and Section 7.2.
To obtain a proper type graph Algorithm 3.3.6 (construct) is used to duplicate nodes as needed to obtain the required tree-shape. This means that there can be multiple nodes with the same function symbol as label but all these nodes will be equivalent.

**Definition 5.2.1 (Type Graph Jungle Widening)**

\[ T \triangledown T' = \begin{cases} T & \text{if } T' \subseteq T \\ \text{construct}(\text{jungle}(T \sqcap T')) & \text{otherwise (Algorithm 5.2.2)} \end{cases} \]

**Algorithm 5.2.2 (Jungle Widening)**

\[ \text{jungle}(T): \]
\[ \text{if } T = \text{any} \text{ then} \]
\[ \text{any} \]
\[ \text{else} \]
\[ \text{Nodes} := \emptyset \]
\[ \text{build(\text{prlb}(T))} \]

\[ \text{proc build(}lbs\text{):} \]
\[ \text{if } \exists m (lbs, m) \in \text{Nodes} \text{ then} \]
\[ \text{return } m \]
\[ \text{else} \]
\[ \text{create a node } m \]
\[ \text{Nodes} := \text{Nodes} \cup \{lbs, m\} \]
\[ \text{if } lbs = \{f\} \text{ then} \]
\[ \text{lb}(m) := f \]
\[ \text{for } i \in [1, \text{arity}(f)] \]
\[ \text{if } \exists n \in T_i \downarrow i \text{ then} \]
\[ m \downarrow i := \text{any} \]
\[ \text{else} \]
\[ m \downarrow i := \text{build(\text{prlb}(T_i \downarrow i))} \]
\[ \text{else} \]
\[ \text{lb}(m) := \text{or} \]
\[ \text{for each } f \in lbs \]
\[ m \downarrow f := \text{build(\{f\})} \]
\[ \text{return } m \]

The main component of Algorithm 5.2.2 is the procedure *build*. It takes a set *lbs* of function symbols and will return a node that is an upper bound of all functor nodes in the grammar graph (or type graph) labeled with one of
the function symbols in \( \text{lbs} \). If there is already a node for this set of functors then it is looked up in \( \text{Nodes} \) and returned directly. Otherwise there are two cases:

If \( \text{lbs} \) is a singleton set \( \{ f \} \) then the result will be a functor node \( m \) that should be an upper bound of all nodes in the grammar graph with label \( f \), that is, \( T_{lf} \). If successor \( i \) of some node with label \( f \) is an any-node then successor \( i \) of \( m \) must be as well, otherwise an upper bound of all the \( i \)-successors is obtained by calling \( \text{build} \) recursively.

If on the other hand \( \text{lbs} = \{ f_1, \ldots, f_k \} \) is not a singleton then the result must be an or-node with \( k \) successor functor-nodes, one for each function symbol in \( \text{lbs} \). The appropriate functor-nodes are obtained by a recursive call to \( \text{build} \) with a singleton set \( \{ f_i \} \) that will therefore result in a functor node, either by lookup in \( \text{Nodes} \) or as described in the previous paragraph.

Termination is ensured since there will be at most one node \( m \) constructed for each subset of the function symbols in the input type \( T \).

That the result is a safe approximation of the original type, that is, \( \forall \forall T T \subseteq \text{jungle}(T) \), can be shown by induction on the depth of the terms in the concretization.

### 5.2.3 Performance

The result of the type jungle widening will be a grammar graph of minimal size and this will also be true of the \( \text{constructed} \) type graph. For this reason minimizing the potentially non-minimal results of upper bound and intersection will have no or very little impact on the largest type encountered during analysis. The results below are without such minimization but the results (not shown) with minimization added confirm that the difference is insignificant.

Table 5.2 summarizes the resource usage on the synthetic benchmarks \( \text{tree}_n \) and \( \text{expr} \) using the type-jungle widening with type graphs and various worklist scheduling policies.

For \( \text{tree}_n \) the number of iterations is linear in \( n \) for the fair worklist scheduling policies FIFO, LRF and TWO-PHASE, while FIXED and LIFO cause a quadratic number of iterations. The number of widenings (that is, changes in success pattern) is the same for all and linear in \( n \), the extra iterations of FIXED and LIFO are thus unproductive and, in some sense, unnecessary.

The size of the largest type graph encountered appears to be (at least) exponential in \( n \), even when only the size of minimized type graphs (in parenthesis) is taken into account. In sharp contrast, column \( \max |D| \) shows that the largest number of non-equivalent nodes in any intermediate type graph is small and linear in \( n \). The exponential size is thus only caused by...
Table 5.2: Summary of using type graphs with Type-Jungle widening on synthetic benchmarks. The number of iterations for the worklist scheduling policies (FAIR is either of FIFO, LRF or TWO-PHASE); the number of success pattern widenings ($\nabla_{SP}$); the size of the largest type graph encountered depending on worklist scheduling, the size of the largest minimized type graph in parenthesis; the largest number of non-equivalent nodes in any encountered type graph ($\max |D|$); the size of the type graph in final success pattern (max result).

<table>
<thead>
<tr>
<th>tree_4</th>
<th>13</th>
<th>18</th>
<th>15</th>
<th>34 (29)</th>
<th>11</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree_5</td>
<td>16</td>
<td>25</td>
<td>21</td>
<td>359 (121)</td>
<td>69</td>
<td>13</td>
</tr>
<tr>
<td>tree_6</td>
<td>19</td>
<td>33</td>
<td>28</td>
<td>1399 (327)</td>
<td>139 (119)</td>
<td>15</td>
</tr>
<tr>
<td>tree_7</td>
<td>22</td>
<td>42</td>
<td>36</td>
<td>5527 (901)</td>
<td>279 (239)</td>
<td>17</td>
</tr>
<tr>
<td>tree_8</td>
<td>25</td>
<td>52</td>
<td>45</td>
<td>21975 (2535)</td>
<td>559 (479)</td>
<td>19</td>
</tr>
</tbody>
</table>

expr 136 30 $\geq 4M$ ($\geq 2M$) 5.7M (3.5M) 34 18

The conclusion thus far is clear. Representing grammar domains with type graphs will, for an important class of programs, lead to types with infeasibly and unnecessarily large size. The size is not caused by large precision but only by the tree shape imposed on type graphs. Furthermore, using a scheduling policy that is fair (FIFO, LRF, TWO-PHASE) appears to avoid a lot of unnecessary iterations, that is, iterations that will not lead to changes to a call or success pattern.

However, slight modifications to the scheduling policy can have a dramatic effect. Consider what happens when analyzing the benchmark tree_4, say

```prolog
main :- tree(_,Any).
tree(a). tree(b(T)) :- tree(T). tree(c(T)) :- tree(T). tree(d(T)) :- tree(T).
```

\[\text{Actually, column max } |D| \text{ was obtained by analyzing with minimized grammar graphs as domain. It is the largest grammar graph occurring before minimization, this means that the number of non-equivalent nodes can be somewhat smaller.}\]
Table 5.3: Type-Jungle widening on synthetic benchmarks. Clauses sorted non-recursive last. The FAIR (FIFO, LRF, TWO-PHASE) policies encounter type graphs with a number of nodes exponential in \( n \), this is true also for the size of the largest minimized type graph encountered (not shown). The unfair policies (FIXED, LIFO) encounter smaller type graphs, the largest minimized intermediate type graph for expr is about 4500 nodes (not shown). The largest number of non-equivalent nodes (not shown) is never more than twice the size of the final result.

Analysis starts with analyzing main and reaches the call to tree in the body. This invokes ai_goal (Section 4.1.3.2), with the call pattern tree(any). By default eager analysis of calls is used so all clauses of predicate tree are analyzed by ai_goal. Since the base case of tree is the first clause it will also be analyzed first and lead to an initial success pattern tree(a). The recursive clauses will then be analyzed and each of them contribute to the success pattern that will be used for the recursive call in the body of later clauses.

So, regardless of the scheduling policy there will be an identical eager analysis pass that install the same initial success pattern for tree.

However, if no such initial success pattern occurs then the analysis will behave quite differently. There are several ways that scheduling could be affected to obtain this effect. Either because the base case were instead last in tree, or eager analysis of calls is not used, or because the analyzer rearranges the clauses prior to analysis and moves non-recursive clauses last.

In either of these cases the order clauses are analyzed will only depend on the scheduling policy for the worklist. The results when the analyzer rearranges

\[^5\text{Since the call pattern already is tree(any) there will be no recursive analysis from ai_clause.}\]

\[^6\text{In combination with the order clauses are added to the worklist when several clauses, of the same or different predicates, need to be reanalyzed (line 7 of ai_clause Section 4.1.3). The analyzer adds clauses of predicates with lexically smaller names first.}\]
all predicates so that non-recursive clauses come last are summarized in Table 3.3.

As can be seen the fair scheduling policies FIFO, LRF and TWO-PHASE appears robust in that they are largely unaffected. They still use a linear number of iterations and widenings but will still encounter exponentially large intermediate types.

The behavior using the unfair scheduling policies LIFO and FIXED is significantly different. The number of iterations and widenings are now much worse, cubic and quadratic in \( n \) respectively for \( \text{tree}_m \). However, the largest type graph encountered is now much smaller and does no longer appear to grow exponentially.

The largest type graph for benchmark \( \text{expr} \) is still excessive with FIXED and LIFO scheduling. For the synthetic benchmarks with these schedules all type graphs have less than 23 non-equivalent nodes. For the large benchmark \( \text{chat} \_\text{big} \) the largest type graph is over half a million nodes and for the slightly smaller benchmark \( \text{chat} \_\text{parser} \) the analyzer exhausts memory after having encountered a type graph with close to eight million nodes. In contrast, using (minimized) grammar graphs with the fair FIFO scheduling (Table 7.4), the size of the largest grammar graph, and thus the largest number of non-equivalent nodes, is about 300 nodes when analyzing \( \text{chat} \_\text{parser} \) and about 700 nodes for \( \text{chat} \_\text{big} \).

The conclusion thus far is that the behavior of the analysis is very sensitive to the scheduling policy and, as argued in Section 4.3.1, the number of iterations says little about the total cost since the size of the type graphs vary much more than the number of iterations (or widenings).

Note that for the cases where large types occurs it is only because type graphs are used to represent the types. If (minimized) grammar graphs are used directly then choosing one of the fair scheduling policies would indeed be advantageous and give linear number of iterations and widenings and also linear sized types for \( \text{tree}_m \), see Section 6.3.

The results are much better, and for the synthetic benchmarks independent of the scheduling policy, if each predicate is rewritten to single clausal form (Section 4.2.1). This has the effect of combining the result of all clauses with upper bound before applying widening. As discussed in Section 4.2.1 this could be expected to lead to better precision but potentially larger types. Contrary to expectations, however, the result is an enormous decrease in the size of the involved type graphs and no significant change to the precision even for the non-synthetic benchmarks. The sizes of the intermediate types decreases to quadratic in \( n \), see Table 5.4.

and adds clauses in the order they appear in the program text. This mainly affects FIFO and LIFO.
### 5.2. Type-Jungle Widening

| Iterations | $\nabla_{sp}$ | max size | max $|D|$ | max result |
|------------|-------------|----------|----------|------------|
| tree;      | 6-8         | 4        | 29       | 12         | 5          |
| tree;      | 6-8         | 4        | 46       | 14         | 6          |
| tree;      | 6-8         | 4        | 67       | 16         | 7          |
| tree;      | 6-8         | 4        | 92       | 18         | 8          |
| tree;      | 6-8         | 4        | 121      | 20         | 9          |
| tree;      | $\Theta(n) = 6-8$ | 4 | $\Theta(n^3)$ | $\Theta(n)$ | $\Theta(n)$ |
| expr       | 6-8         | 4        | 1490     | 35         | 18         |

Table 5.4: Type-Jungle widening on synthetic benchmarks. Predicates rewritten to single-clausal form prior to analysis. Each iteration with tree corresponds to analysis of all its $n$ clauses. The number of iterations varies slightly with scheduling depending on when main is reanalyzed.

At least for the synthetic benchmarks the performance is now of reasonable complexity. Even so, the use of type graphs is more expensive than using grammar graphs as the encountered type graphs are quadratic whereas for (minimized) grammar graphs the encountered types have linear size.

For the larger realistic benchmarks the cost is still unacceptably high. For chat.parser the analysis times out after having encountered a type graph consisting of 3.5 million nodes\(^7\), chat.big encounters a type graph consisting of half a million nodes.

#### 5.2.4 Precision

The precision when using the Type-Jungle widening will be analyzed in detail in Section 7.2 where it is used together with minimized grammar graphs instead of type graphs. As hinted above this is a much more efficient representation of types with this widening.

The Type-Jungle widening is in theory strictly more precise than the simple functors-widening in the previous section. In Section 7.2 this will be seen to hold also in practice especially for the more detailed precision measure “tags” (Section 4.3.2.3).

For the list_of_lists predicate the result becomes $T$,

\[
\begin{align*}
T & \rightarrow [] \mid T_1.T \\
T_1 & \rightarrow [] \mid T_1.T \mid \text{num}
\end{align*}
\]

This result is strictly better than for the functors widening in Section 5.1.3. The type captures the fact that the top-level is a list, however the type

---

\(^7\)As mentioned above the largest grammar graph encountered during analysis using (minimized) grammar graph consists of about 300 nodes.
loses precision when describing the list elements since the widening will force the two list-constructors to be equivalent, that is, to have the same concretization. Note however that, unlike the case with functors widening, the type captures the fact that the tail of any list constructor will be a proper list.

5.2.5 Conclusions

The type-jungle widening maps a grammar graph onto a finite subdomain consisting of grammar graphs with at most one functor-node for each function symbol. Type graphs were used as type-domain so the type-jungle resulting from the widening had to be converted to a type graph.

The performance when used to infer recursive types were investigated using the synthetic benchmarks expr and tree_n. As an estimate of cost the number of nodes of the largest encountered type graph was measured.

For the default analysis method, where each clause is analyzed separately the costs are prohibitive, despite of, or perhaps because of, the low precision of the type-jungle widening. The size of the largest intermediate type graph appears to grow exponentially with \( n \). The exponential growth holds even if only the size of the minimized type graphs are taken into account.

Rearranging the clauses will give the worklist scheduling policy more control, in this case the unfair policies perform much worse in terms of iterations and widenings but manage to keep the encountered type graphs much smaller.

Finally, by postponing the widening and the update of the global success pattern until the result of all clauses are available (using single-clausal form) the number of iterations becomes linear and the type graphs grow only quadratically with \( n \).

The results show that, with domains with as high and varying cost as type graphs, differences in the size of the types and the associated cost of the domain operations is likely to have a larger impact on the efficiency of the analysis than the framework specific costs such as the number of analyzed clauses. In particular, as witnessed by the single-clausal case, heuristics such as propagating information as soon as possible may be counterproductive.

However, in all these cases the number of non-equivalent nodes within a type graph is small and linear in \( n \), the quadratic to exponential size of the type graphs is thus unnecessary and is only caused by the tree-like shape imposed on type graphs. The type-jungle widening is therefore a poor choice for the type graph domain and will instead be studied using grammar graphs in Section 6.3 and Section 7.2.
5.3 JANSSENS’ WIDENING

The widening in this section is what Janssens and Bruynooghe used together with the original formulation of type graphs [31, 32].

In this section Janssens’ widening is described and a termination problem present is the original presentation is corrected [31]. The performance, in terms of type graph size and iterations, when inferring recursive types is then investigated. This is studied by analyzing the synthetic benchmarks tree and expr using a number of ways to schedule clauses. Finally the impact of various optimizations is investigated.

The conclusion is that using type graphs together with Janssens’ widening is infeasible, at least for inferring the types of programs that manipulate recursively defined data structures. The reasons are that type graphs are troublesome by themselves as they tend to grow exponentially and that the widening appears ineffective in restricting the growth.

Analysis using Janssens’ widening is also extremely sensitive to what scheduling policy is used for re-analysis of clauses and to when widening is applied. The number of iterations and widenings vary from linear to exponential depending on the parameters of the analyzer framework.

5.3.1 Definition

Janssens’ widening for type graphs uses a restriction operator that maps the input type graph onto a finite subdomain in the manner of Definition 2.1.2.

The finite subdomain consists of the type graphs that only have at most \( k \) occurrences of each function symbol along any forward path. That the resulting subdomain is finite follows immediately since the depth, and thus the size, of each type graph is limited by the maximally \( k \) occurrences of each function symbol (of which there are finitely many) along any forward path.

**Definition 5.3.1 (Type Graph Depth-\( k \) Widening)**

\[
T \triangledown T' = \begin{cases} 
T & \text{if } T' \subseteq T \\
\text{restrict}_k(T \cup T') & \text{otherwise (Algorithm 5.3.2)}
\end{cases}
\]

The restriction operator \( \text{restrict}_k \) works by removing any forward path in which the number of occurrences of some function symbol is larger than some fixed constant \( k \). The restriction operator will replace such paths with circular paths to obtain a safe approximation.
The type graphs used by Janssens differ slightly from the type graphs defined in Section 3.3.3 in that nested or-nodes are permitted as long as no two principal nodes have the same function symbol as label. By allowing nested or-nodes a more compact representation is possible in some cases, in particular it allows more precise type graphs for a fixed $k$, see Figure 5.2.

The actual implementation used in the evaluation does allow nested or-nodes and so do all the other domain operations when used together with this widening. Supporting nested or-nodes adds significant complexity to the implementation. It is not clear if the potential benefits are significant in practice. To simplify the presentation a formulation on type graphs without nested or-nodes will be used.

### 5.3.2 The Restriction Operator

The main part of Algorithm 5.3.2 is the procedure $\text{restrict}$ that creates an upper bound of a set of nodes from a type graph while ensuring that the created type graph adheres to the necessary shape requirements, that is,
Algorithm 5.3.2 (Restrict$_k$) $k$-restriction. Used to build an upper approximation of a set of nodes while at the same time ensuring that no functor occurs more than $k$ times along any forward path from the root. The underlined condition is necessary to ensure termination and not present in [31].

\[
\text{restrict}_k(n) = \text{restrict}([n], 0, 0)
\]

\[
\text{restrict}(S_0, \text{level}, \text{ANC}) =
\]
\begin{algorithmic}[1]
  \STATE if any $\in S_0$ then return any
  \STATE $S := \text{pmd}(S_0)$
  \STATE if $\exists m, l \in (S, m, l) \in \text{ANC}$ then return $m$ — i.e., a backward arc to $m$
  \STATE \hspace{1em}if $\text{prlb}(S) = \{f\}$ then
  \hspace{2em}ANC$_f := \{(S', m, l) \in \text{ANC} \mid \text{prlb}(S') = \{f\}\}$
  \hspace{2em}if $|\text{ANC}_f| = k$ then
  \hspace{3em}— $(k+1)$th occurrence of $f$, must not descend
  \hspace{3em}if there is an $(S', m, l) \in \text{ANC}_f$ such that $S \subseteq S'$ then
  \hspace{4em}return $m$ — $m$ is a less precise ancestor
  \hspace{3em}else
  \hspace{4em}chose a $(S', -l) \in \text{ANC}$ such that $f \in \text{prlb}(S'), S \subseteq S'$
  \hspace{4em}— at least $k$ such are guaranteed to exist.
  \hspace{4em}restart the call to restrict at level $l$
  \hspace{4em}with $S \cup S'$ as first argument
  \hspace{3em}else
  \hspace{4em}create a new functor-node $m$ with $\text{lb}(m) = f$
  \hspace{4em}ANC$' := \text{ANC} \cup \{(S, m, \text{level})\}$
  \hspace{4em}level$' := \text{level} + 1$
  \hspace{4em}for each $i \in [1, \text{arity}(f)]$ do
  \hspace{5em}$m \downarrow i := \text{restrict}(S \downarrow i, \text{level}', \text{ANC}')$
  \hspace{4em}return $m$
  \hspace{3em}else
  \hspace{4em}create a new or-node $m$
  \hspace{4em}ANC$' := \text{ANC} \cup \{(S, m, \text{level})\}$
  \hspace{4em}level$' := \text{level} + 1$
  \hspace{4em}for each $f \in \text{prlb}(S)$ do
  \hspace{5em}$m \downarrow f := \text{restrict}(S \downarrow f, \text{level}', \text{ANC}')$
  \hspace{4em}return $m$
  \end{algorithmic}
that no forward path in the created type graph has more than \( k \) occurrences of any functor node.

If line 6 to 16 of procedure `restrict` are ignored then the remaining code is a straightforward generalization of the upper bound for type graphs (Algorithm 3.3.7, page 59) where the first argument \( (S_0) \) is a set of nodes whose upper bound should be created instead of a pair of nodes as in the ordinary upper bound operation. See also Algorithm 6.6.4, page 179 that implements such a generalized upper bound.

As in the ordinary upper bound algorithm the type graph that should become the upper bound is created depth first. The argument \( ANC^6 \) represents the ancestors of the node that should be constructed and \( level \) is the number of ancestors, that is, the number of elements of \( ANC \).

The difference compared to ordinary upper bound is that `restrict` also ensures that each functor occurs no more than \( k \) times along any forward path in the created type graph.

When a functor node is about to be created (line 5) then `restrict` checks if it already has \( k \) ancestor functor nodes with the same label. If it has not then the algorithm continues as an algorithm for upper bound (line 17–22). If, on the other hand, there are already \( k \) such ancestors then there are two possibilities:

1. A backward arc to an proper ancestor functor node with label \( f \) that is less precise can be used instead of creating a new node. A sufficient condition for this is that the ancestor is created as the upper bound of a larger set of nodes (all with label \( f \)).

2. There is no ancestor node such that a backward arc can be used. In this case the depth first creation of the upper bound is restarted at some level closer to the root using a larger set of nodes (line 12–15).

An example with a trace appears in Figures 5.3 and 5.4.

Algorithm 5.3.2 can be seen as an instance of a general class of algorithms that create an upper bound of a set of (type graph or grammar graph) nodes while ensuring that the resulting type satisfies certain structural constraints. In Section 6.6.1 and Section 6.5 the same idea will be used with other structural constraints.

\(^8\)In Algorithm 3.3.7 argument \( S_0 \) is used for the same purpose.

\(^9\)A more precise test could be used that compared the nodes in \( S \) and \( S' \) using \( \sqsubseteq \), this has not been evaluated. It would also be possible to use ancestors corresponding to or-nodes in some cases.
5.3.2.1 Termination Termination of Algorithm 5.3.2 hinges on two facts:

- The size of the resulting type graph is limited: Each created node corresponds to a set of nodes from the original type graph and no such set occurs more than once along a forward arc in the created type graph.

- Restarts (at line 14) can only happen a finite number of times: This is ensured by requiring that a restart always will happen with a strictly larger set of nodes. Since the number of nodes is finite a call to restrict can only be restarted a finite number of times.

The underlined test $S \not\subseteq S'$ in Algorithm 5.3.2 ensures that such a restart will be made with a strictly larger set of nodes. No such precaution was present in the algorithm presented in [31] and this will cause non-termination for some benchmarks.

For an example of the significance of the condition $S \not\subseteq S'$, consider step 13 of the example and trace in figures 5.3 and 5.4. Here $S = \{n_6\}$ and there is an $(S', m, l) \in \text{ANC}$ such that $S' = \{n_3, n_4, n_6, n_7\}$ and $l = 1$. Restarting at level 1 with restrict($S \cup S' = S', \ldots$) would not lead to progress and would cause non-termination.

5.3.2.2 Choosing an Ancestor When restrict encounters the occurrence $k + 1$ for a functor $f$ such that it cannot simply create a backward arc to some ancestor with the same functor then it must restart the computation at a level closer to the root. This amounts to selecting a suitable entry $(S', \downarrow\text{level}^\downarrow\text{level})$ from $\text{ANC}$.

A selected entry should preferably make the final result of the analysis as precise as possible, presumably this is the reason Janssens used the additional condition $f \in \text{prlb}(S')$. In addition it was suggested, in essence, that the intersection of $S \cap S'$ should be as small as possible, that is, to select an $S'$ as similar as possible to $S$ [31, p. 49].

It turns out that this heuristic can lead, to a lot of unnecessary work before restrict arrives at the final type graph, and also to non-termination as described in the previous section.

In particular, selecting an ancestor at level $l$, corresponding to an or-node, below the nearest ancestor with label $f$ is likely to lead to another restart when restrict is called upon to create the child labeled $f$ of the or-node being created at level $l$. The reason is that this child too will have $k$ ancestors with label $f$, introducing a backward-arc will not be possible so restrict will have to be restarted at some level closer to the root. It thus seems appropriate to never restart at a level below the nearest $f$-ancestor.
This effect can be seen in the example. At step 7 a depth conflict for functor $c$ is detected. The root, level zero, is the closest level corresponding to a functor-node with label $c$ but instead level one is restarted. If in step 10, functor $c$ would have been chosen instead of $b$ then the above situation would occur. As it now stands step 10 instead descends using label $b$ and a restart is forced in step 13.

An even simpler heuristic is to always restart at one of the $k$ ancestor levels corresponding to functor-nodes with label $f$. This choice makes intuitive sense as well and also solves the termination problem. Informal measurements indicate that this can save a lot of work in $\textit{restrict}$.

While these heuristics may affect the number of "steps" within the widening operation itself they will not affect the fundamental problems that will be seen for this widening. For this reason neither of these heuristics are used, instead a method based on Janssens' description is used. The ancestor $S'$ with the largest number of nodes in common ($|S' \cap S|$) is used for restarts of $\textit{restrict}$, if a tie then the deepest one is chosen.

It would be interesting to characterize the complexity of Algorithm 5.3.2. In the worst case it would seem that it could progress very slowly since it discards all work below a certain level when a restart happens and in the worst case a restart will extend the set of nodes in the argument with only one node.
Original type graph. \( n_4, n_6 \) violates depth-\( k \) restriction \((k = 1)\).

\( \text{Step 7} \) First depth conflict detected

\( \text{Step 13} \) Second depth conflict detected

\( \text{Step 20} \) First backward arc created

Final result. All depth-conflicts resolved

Figure 5.3: Depth-1 restriction,  restrict((\( n_1 \), 0, \( \emptyset \)). Constructed nodes are subscripted with \( S \), the set of nodes from the original type that they approximate. See Figure 5.4 for a detailed trace.
<table>
<thead>
<tr>
<th>step</th>
<th>level</th>
<th>$S$</th>
<th>$\text{ANC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>${n_1}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>${n_3, n_4, n_7}$</td>
<td>${{n_1}, 0}$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>${n_7}$</td>
<td>${{n_3, n_4, n_7}, 1, {n_1}, 0}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>return a functor node $n_{{n_7}}$ with label <code>a</code>.</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>${n_3}$</td>
<td>${{n_3, n_4, n_7}, 1, {n_1}, 0}$</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>${n_6, n_8}$</td>
<td>${{n_3}, 2, {n_3, n_4, n_7}, 1, {n_1}, 0}$</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>${n_8}$</td>
<td>${\ldots}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>return a functor node $n_{{n_8}}$ with label <code>a</code>.</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>${n_6}$</td>
<td>${{n_6, n_8}, 3, {n_3}, 2, {n_3, n_4, n_7}, 1, {n_1}, 0}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Depth limit $k$ exceeded, line 11 applies</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$[[S', I] \in \text{ANC}, \text{prlb}(S') = {c}] = {0} \implies</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Could restart at either ${n_3, n_4, n_7}, 1$ or ${n_1}, 0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Arbitrarily restart level 1, with ${n_3, n_4, n_7} \cup {n_6}$</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>${n_3, n_4, n_6, n_7}$</td>
<td>${{n_1}, 0}$</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>${n_7}$</td>
<td>${\ldots}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>return a functor node $n_{{n_7}}$ with label <code>a</code>.</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>${n_3}$</td>
<td>${\ldots}$</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>${n_6, n_8}$</td>
<td>${\ldots}$</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>${n_8}$</td>
<td>${\ldots}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>return a functor node $n_{{n_8}}$ with label <code>a</code>.</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>${n_6}$</td>
<td>${{n_6, n_8}, 3, {n_3}, 2, {n_3, n_4, n_7}, 1, {n_1}, 0}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Depth $k$ exceeded for $n_6$ (again)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Restart level 1 or 3 would lead to nontermination</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Restart level 0 with ${n_1} \cup {n_6}$</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>${n_1, n_6}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>${n_3, n_4, n_6}$</td>
<td>${{n_1, n_6}, 0}$</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>${n_7}$</td>
<td>${\ldots}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>return a functor node $n_{{n_7}}$ with label <code>a</code>.</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>${n_3}$</td>
<td>${{n_3, n_4, n_7}, 1, {n_1, n_6}, 0}$</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>${n_6, n_8}$</td>
<td>${{n_3}, 2, {n_3, n_4, n_7}, 1, {n_1, n_6}, 0}$</td>
</tr>
<tr>
<td>19</td>
<td>4</td>
<td>${n_8}$</td>
<td>${\ldots}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>return a functor node $n_{{n_8}}$ with label <code>a</code>.</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>${n_6}$</td>
<td>${{n_6, n_8}, 3, {n_3}, 2, {n_3, n_4, n_7}, 1, {n_1, n_6}, 0}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Depth $k$ exceeded for $n_6$ (again) line 9 applies</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>backward arc to the node $n_{{n_6, n_6}}$ being created at level 0</td>
</tr>
<tr>
<td>18 cont.</td>
<td>3</td>
<td>Create or-node $n_{{n_6, n_6}}$</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>4</td>
<td>${n_4}$</td>
<td>${{n_6, n_8}, 3, {n_3}, 2, {n_3, n_4, n_7}, 1, {n_1, n_6}, 0}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$k$ exceeded, restart level 0 with ${n_1, n_6} \cup {n_4}$</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>${n_1, n_4, n_6}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Figure 5.4: Trace of calls to $\text{strict}$ corresponding to figure 5.3. The node-component is not shown for the entries on $\text{ANC}$.
### 5.3. Janssens' Widening

<table>
<thead>
<tr>
<th></th>
<th>Iterations</th>
<th>$\lor_{sp}$</th>
<th>\text{max size}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FAIR</td>
<td>FIXED</td>
<td>LIFO</td>
</tr>
<tr>
<td>tree_1</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>tree_2</td>
<td>12</td>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>tree_3</td>
<td>16</td>
<td>28</td>
<td>19</td>
</tr>
<tr>
<td>tree_4</td>
<td>20</td>
<td>31</td>
<td>13</td>
</tr>
<tr>
<td>tree_5</td>
<td>24</td>
<td>48</td>
<td>16</td>
</tr>
<tr>
<td>tree_r</td>
<td>28</td>
<td>71</td>
<td>19</td>
</tr>
<tr>
<td>tree_n</td>
<td>timeout</td>
<td>timeout</td>
<td>$\geq 200k$</td>
</tr>
<tr>
<td>expr</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^4)$</td>
</tr>
</tbody>
</table>

Table 5.5: Janssens' unmodified widening (depth bound $k = 1$) on synthetic benchmarks. Number of iterations for the FAIR worklist scheduling policies (FIFO, LRF, TWO-PHASE) and the unfair policies FIXED, LIFO. Number of widenings of success patterns ($\lor_{sp}$). Number of nodes of the largest type graph encountered during analysis (max size). Missing entries and entries with $\geq$ exhausted either memory or time limits, in this case the largest type graph encountered before interruption is shown.

### 5.3.3 Performance

Table 5.5 summarizes the resource usage on the synthetic benchmarks using Janssens' widening with $k = 1$. As can be seen, for $\text{tree}_n$, the size of the largest type graph that occurs during analysis appears to grow at least exponentially with $n$. Also here unfair scheduling causes an asymptotically larger number of iterations and widenings.

The measurement in Table 5.5 used a version of upper bound that optimizes the case when the arguments are comparable. The result is even worse if the method in [31] is used, that is, if an upper bound is explicitly created also when the arguments are comparable.

#### 5.3.3.1 Minimization

Janssens and Bruynooghe noted that “execution times of the prototype are rather excessive” [32, p. 251]. They suggest some possible improvements to the analyzer framework and also note that the type graphs are non-minimal and that this may have an adverse effect on the efficiency. In her thesis Janssens reports some measurements for three very small benchmarks for which the analysis time decreased somewhat when type graphs were minimized.

To investigate whether this could stop the exponential growth minimization was performed after all procedures that produce type graphs ($\text{restrict}_{k}$, intersection and upper bound).
The size of the largest minimized type graph was measured making the conservative but unrealistic assumption that all operations could be modified to produce minimal type graphs directly.

The results are summarized in Table 5.6, and as can be seen, the size of the largest minimized type graphs still appears to grow (at least) exponentially with \( n \). The number of iterations and widenings is the same as in Table 5.5.

Two methods was also tried that improved the type-jungle widening in Section 5.2, single clausal form and rearranging the clauses.

The rightmost column of Table 5.6 shows the result, with minimization, when the predicates are rewritten in single clausal form. Unfortunately it is not an improvement when used together with Janssens’ widening. If no minimization is done then the result is even worse.

Recall that for the type-jungle widening, Section 5.2.3, moving the non-recursive clauses last affected the analysis with the unfair FIXED and LIFO scheduling policies so that more iterations were needed but also much smaller intermediate type graphs.

Unfortunately this somewhat encouraging result does not carry over to Janssens’ widening. Rearranging the clauses in this manner still causes the size of the largest type graph to grow (at least) exponentially. This result holds even if minimization is performed as in Table 5.6. Furthermore, for \( \text{tree}_n \), the unfair scheduling policies FIXED and LIFO in addition require (at least) an exponential number of iterations as well as widenings.

\[
\begin{array}{l|c|c|c|c|c}
\text{clauses separate} & \text{fair} & \text{fixed} & \text{lifo} & \text{single clausal} \\
\hline
\text{tree}_2 & 7 & 7 & 7 & 7 \\
\text{tree}_3 & 22 & 22 & 18 & 28 \\
\text{tree}_4 & 67 & 70 & 33 & 242 \\
\text{tree}_5 & 203 & 306 & 61 & 32294 \\
\text{tree}_6 & 685 & \geq 995 & 121 \\
\text{tree}_7 & \geq 27115 & 241 \\
\text{tree}_8 & 481 \\
\text{tree}_n & \Omega(2^n) \\
\hline
\text{expr} & \geq 2123946 & \geq 624006
\end{array}
\]

Table 5.6: Janssens’ widening (depth bound \( k = 1 \)) on synthetic benchmarks. Minimization applied after widening, intersection, upper bound. Size of largest minimized type graph shown. For single-clausal the result is the same for all worklist policies.
### 5.3. Janssens’ Widening

<table>
<thead>
<tr>
<th>clauses separate</th>
<th>FIFO</th>
<th>LIFO</th>
<th>FIXED</th>
<th>LIFO</th>
<th>single clausal</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>tree3</td>
<td>14</td>
<td>14</td>
<td>13</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>tree4</td>
<td>43</td>
<td>41</td>
<td>28</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>tree5</td>
<td>139</td>
<td>136</td>
<td>56</td>
<td>686</td>
<td></td>
</tr>
<tr>
<td>tree6</td>
<td>517</td>
<td>436</td>
<td>102</td>
<td>2492</td>
<td></td>
</tr>
<tr>
<td>tree7</td>
<td>2295</td>
<td>≥109</td>
<td>199</td>
<td>7148</td>
<td></td>
</tr>
<tr>
<td>tree8</td>
<td>≥249</td>
<td>≥246</td>
<td>399</td>
<td>≥17222</td>
<td></td>
</tr>
<tr>
<td>tree9</td>
<td>≥1062922</td>
<td>≥354307</td>
<td>≥624006</td>
<td></td>
<td></td>
</tr>
<tr>
<td>expr</td>
<td></td>
<td></td>
<td>Θ(2^n)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.7: Janssens’ widening (depth bound k = 1) performed as part of upper bound on synthetic benchmarks. Minimization applied after widening, intersection, upper bound. Size of largest minimized type graph shown. For single-clausal the result is the same for all worklist policies.

#### 5.3.3.2 The Number of Non-Equivalent Nodes

For the type-jungle widening in Section 5.2 the largest number of non-equivalent nodes within a type graph was linear in \( n \), even though the tree shape of type graphs caused the representation as type graph to grow exponentially. To see if this is the case also for Janssens’ widening the number of non-equivalent nodes was measured for the analysis of Table 5.5.

For the fair scheduling policies and for FIXED the number of non-equivalent nodes is smaller than the type graph sizes, as expected, but still appear to grow exponentially with \( n \). This means that even if all redundancy caused by non-equivalent nodes were removed the analysis would still be infeasible.

For LIFO the number of non-equivalent nodes within any encountered type graphs does appear to grow only linearly with \( n \). This means that, at least in this case, the exponential type graph size is caused only by the tree shape of type graphs.

Unfortunately, it is not clear how to utilize the result for LIFO. In particular since it is quite fragile, rearranging the clauses as above will cause the number of non-equivalent nodes and the number of iterations to grow exponentially.

#### 5.3.3.3 Avoiding Upper Bound

The main cause of large type graphs is the upper bound operation. A promising idea is then to avoid this operation...
altogether by combining upper bound with the widening operation. This requires slight modifications to the analyzer framework.

Let \( \text{restrict}_\infty \) be \( \text{restrict}_k \) without any depth restriction. Upper bound can then be defined as:

\[
T_0 \cup T_1 = \begin{cases} 
T_0 & \text{if } T_1 \subseteq T_0, \\
T_1 & \text{if } T_0 \subseteq T_1, \\
\text{restrict}_\infty (\text{prnd}(T_0) \cup \text{prnd}(T_1)) & \text{otherwise.}
\end{cases}
\]

Defining upper bound in terms of \( \text{restrict}_\infty \) immediately gives an alternative definition of the widening that avoids constructing the intermediate upper bound,

\[
T_{old} \vee T_{new} = \begin{cases} 
T_{old} & \text{if } T_{new} \subseteq T_{old}, \\
\text{restrict}_k(\text{prnd}(T_{new})) & \text{if } T_{old} \subseteq T_{new}, \\
\text{restrict}_k(\text{prnd}(T_{0}) \cup \text{prnd}(T_{1})) & \text{otherwise.}
\end{cases}
\]

This version of the widening was tried together with minimization after all operations, see Table 5.7. For \( \text{tree}_n \) the size of the largest minimized type graphs, while smaller than in Table 5.6, still appears to grow exponentially with \( n \).

### 5.3.4 Precision And The Effect Of \( k \)

In the implementation of Janssens’ widening used in this thesis the same value is used for the depth limit \( k \) throughout the analysis and for all functors. This is not necessary, it is possible to have different values for each function symbol. It would also be possible to vary it as the analysis proceeds, for example to decrease \( k \) to speed up the analysis after it has consumed some specific amount of resources.

Intuitively, increasing the value of \( k \), the depth-limit, should decrease the loss of precision due to the depth-\( k \) restriction. In Janssens’ thesis it was suggested that the depth limit \( k \) should be set for each functor based on the syntactic structure of the program.

---

10The same optimization can be done for other widenings such as the functors and type-jungle widenings.

11It would also have to be modified to take a set of nodes instead of a single root node.
Consider the predicate `list_of_lists/1` of Example 4.2, where the desired success type `Tll` for `list_of_lists/1` is a list of lists of numbers, that is,

\[
Tll \rightarrow \mathit{[]} \mid (T_l, Tll) \\
T_l \rightarrow \mathit{[]} \mid \mathit{num, T_l}
\]

The type graph for `Tll` only have functor-nodes with the list constructor, two deep and thus applying `restrict_2` on it would not affect the precision.

Unfortunately, the precision when using a larger `k` is not necessarily as high as one would expect. The primary reason is that `restrict` has too little information to guide it when it selects what sets of nodes to combine (at line 13 of Algorithm 5.3.2). In particular it always combines the node\(^{12}\) that violated the depth-restriction with some of its ancestors. For the `list_of_lists/1` example the two ancestors will originate with what would ideally become `Tll` whereas the conflicting node corresponds to `T_l`. Thus `restrict_2` will in fact combine `T_l` and (the deepest node corresponding to) `Tll` into a single disjunctive type, eventually producing the less precise type

\[
T_{ll}^{k=2} \rightarrow (T_l, T_2) \mid \mathit{[]} \\
T_l \rightarrow \mathit{num, T_l} \mid \mathit{[]} \\
T_2 \rightarrow (T_3, T_2) \mid \mathit{[]} \\
T_3 \rightarrow (T_3, T_2) \mid \mathit{[]} \mid \mathit{num}
\]

This type says that the first element `T_l` of the top-level list `T_{ll}^{k=2}` is a list of numbers. For the tail `(T_2)` the type is the same as for `k = 1` and identical to the result with the type-jungle widening in Section 5.3.4. In the likely case that `T` would appear as call pattern to a predicate that recurse over the input list the more precise information about the first element would be discarded. In practice, for `list_of_lists`, the type with `k = 2` is thus likely to be no more useful than the result with `k = 1` or with type-jungle widening.

It is quite possible that details in the widening, such as the order type graph nodes are created by `restrict`, and details of the analyzer, such as the order clauses are analyzed, could affect the result of this widening. In particular there may be details that would give the optimal result for `list_of_lists`. None of the FAIR or UNFAIR scheduling policies improves the precision for the implementation used in this thesis. The possibility to vary the order nodes are created in `restrict` has not been evaluated, Janssens and Bruynooghe did not provide details of the order nodes are created in their version of `restrict`. In Section 6.5 a related widening is presented that gives the optimal result using the default parameters but gives less precise results for some scheduling policies.

\(^{12}\)Or, rather, the set \(S\) of nodes that would otherwise have caused a node to be created.
On the bright side, as will be seen in Chapter 7, in practice a larger \( k \) sometimes \textit{does} increase the obtained precision, only not necessarily as much as expected.

5.3.5 Conclusions

The performance of the type graph widening proposed by Janssens and Bruynooghe \cite{JanssensBruynooghe1, JanssensBruynooghe2} was evaluated on programs that manipulate recursive data structures. The syntactic synthetic benchmark \texttt{tree}_{\text{n}} was used to make it possible to vary the number of functors (constructors) of the recursive data structure.

For such programs the size of the largest type graphs that occur during analysis appears to grow exponentially with the number of function symbols, or equivalently, the number of clauses. This holds even for the least precise setting of the widening and even if only the size of minimized type graphs are considered and also if the widening is done directly as part of the upper bound operation. In some cases the size of the type graphs even becomes exponential in the number of non-equivalent nodes in the represented type.

Since predicates that manipulate recursively defined data structures are likely to be part of many programs I conclude that type graphs together with the widening of Janssens and Bruynooghe is unsuitable as a general technique for analysis of Prolog programs.

Many of the larger programs used in the evaluation in Chapter 7 does manipulate recursively defined data-structures. Not all of them cause problem with Janssens’ widening but in these cases it is because the information about the recursive data structures is lost due to poor precision.

If the widening is nevertheless used, then the method used to combine type graph nodes should be modified both to avoid unnecessary work and to ensure termination.

As is the case for the other widenings in this chapter, also for Janssens’ widening the scheduling policy has a large impact on efficiency. The scheduling policy not only affects the size of the type graphs encountered but also the number of iterations and widenings, ranging from a linear number of iterations and widenings for the fair policies to an exponential number for the unfair policies.

Finally, increasing the precision parameter \( k \) may not produce the expected increase in precision. In particular \( k = 2 \) will not enable the analyzer to derive the expected optimal result for a list of lists of elements.
5.4 TOPOLOGICAL CLASH WIDENING

This widening is the only grammar widening in this thesis that does not just consider the upper bound of the two arguments, instead the shapes of successive approximations are compared and the differences in shape are used to guide a transformation to obtain the resulting type graph. Since it does not use a finite subdomain there are no a priori restrictions on the resulting type graphs and with appropriate heuristics this widening gives more precise results than any of the others in this chapter.

This widening was used by Hen tenryck et al. together with their analyzer framework GAIA. Slightly different variants of the topological clash widening was presented in [28, 29, 30]. The presentation in this section is based on that in [30]. The definitions and proofs differs for two main reasons. Firstly, the widening presented here is applicable to ordinary compact type graphs, without the further “cosmetic restrictions” used in [30]. Secondly, there are errors in the termination related proofs presented in [30].

In this section I describe the widening, the modifications I have made and the heuristics I use to obtain a concrete implementation. The section concludes with an investigation of the performance when inferring recursive types. The performance and precision on the larger benchmarks are evaluated in Chapter 7.

The description in this section, and that of Hen tenryck et al., are formulated as a number of transformations on a type graph that eventually will produce a type graph with certain properties. An alternative and new formulation is presented in Section 6.6.1, where the type graph is constructed directly with the desired properties.

5.4.1 Description

The top-level of the widening is similar to the ones used by the widenings based on restriction earlier in this chapter. The difference is that this widening also will use the old approximation as input\(^\text{13}\).

\[
T_{\text{old}} \triangledown T_{\text{new}} = \begin{cases} 
T_{\text{new}} & \text{if } T_{\text{old}} = \bot, \\
T_{\text{old}} & \text{if } T_{\text{new}} \sqsubseteq T_{\text{old}}, \\
\text{widen}_{\text{TC}}(T_{\text{old}}, T_{\text{old}} \sqcup T_{\text{new}}) & \text{otherwise.}
\end{cases}
\]

In the following we are only concerned with the procedure \(\text{widen}_{\text{TC}}(T, T')\) and will assume that \(T \sqsubseteq T'\). The underlying idea is to compare these

\(^{13}\text{The special case when } T_{\text{old}} = \bot \text{ can be used for other widenings as well. It may improve precision, especially for non-recursive predicates.}\)
two type graphs, in the following called simply graphs, using a depth-first traversal to try to determine in what direction the new graph $T'$ is growing compared to the old graph $T$. This information will enable this widening to make more informed choices than what is possible with the restriction based widenings as to when to allow the new graph to grow and when to prevent it.

The spanning trees of the type graphs $T, T'$ are traversed in pre-order and pairs of nodes $n, n'$ ($n \in T, n' \in T'$) are processed in a manner similar to how the containment algorithm would perform $T \subseteq T'$. However, when a backward arc is encountered in either graph or when two nodes with different sets of principal labels are encountered then the descent is halted along that path.

The pairs $(n, n')$ of nodes where the traversal is stopped are where the new type graph $T'$ grows compared to the old type graph $T$, although not all such pairs represent growth. In particular, leaving out a lot of details, if the set of principal labels of the new node $n'$ is strictly larger than that of the corresponding old node $n$ then this growth is only allowed if the principal labels of the new node $n'$ is not a subset of the principal labels of any of its proper ancestors. If such an ancestor does exist then the arc used to reach $n'$ is rerouted to the ancestor in a way that ensures that the resulting type graph $T''$ is a safe approximation of the original $T'$.

### 5.4.2 Definition

The following definitions formalize the above intuition and incorporate a number of additional conditions to ensure the correctness of the widening. It follows the description in [30], but with changes to allow the widening to operate directly on compact (and thus normal deterministic and reduced) type graphs.

Some definitions are the amalgamation of definitions appearing in [30] and [28, 29].

Since much of what follows is more concerned with the graph properties of the type graphs than with the fact that they represents types $G$ will be used to denote a (type-)graph, with the convention, as above, that $n, e$ refer to nodes, arcs (or edges) of the original type graph $G$, whereas $n', e'$ refer to components of the new type graph $G'$.

The widening collects pairs of nodes $(n, n')$ from $G, G'$ encountered while traversing the graphs in a lock-step fashion, starting from the roots. For technical reasons each such pair of “matched” nodes are associated with the arc $e'$ that was used to reach $n'$. The set $C(G, G')$ formalizes this.

---

14Recall that the spanning trees are unique for type graphs.
Definition 5.4.1 (C) Define the set $C(G, G')$ as the smallest set $R$, such that

- $\left< (r, r'), e'_{\text{dummy}} \right> \in R$ where $r, r'$ are the roots of $G, G'$ respectively. The dummy arc $e'_{\text{dummy}}$ will never be used.
- if $(n, n') \in R \downarrow 1$ and they have the same depth and prlb-set (and thus the same arity $a$), then $\left< (n \downarrow i, n' \downarrow i), e'_i \right> \in R$, $e'_i = (n', i, n' \downarrow i)$ for $1 \leq i \leq a$. In this case we say that $(n, n')$ introduce $(n \downarrow i, n' \downarrow i)$ via the introducing arcs $e_i = (n, i, n \downarrow i)$ and $e'_i = (n', i, n' \downarrow i)$\(^{15}\).

Note that $n \subseteq n'$ for pairs $(n, n') \in C(G, G') \downarrow 1$ by construction, and also that $n' = \text{any}$ or $\text{prlb}(n) \subseteq \text{prlb}(n')$. We will sometimes drop the arguments of $C(G, G')$ and regard it as set $C$ when the arguments $G, G'$ are clear from the context. The same will be done for other similar functions in this section.

The pairs $(n, n') \in C(G, G') \downarrow 1$ that have the same depth and set of principal labels (prlb-set) corresponds to the similar parts of the (spanning) trees. The remaining pairs in $C(G, G') \downarrow 1$ correspond to Topological Clashes, that is, the pairs of nodes where $G$ and $G'$ begin to differ,

Definition 5.4.2 (Topological Clashes)

\[
\text{TC}(G, G') = \{ \left< (n, n'), e' \right> | \quad \left< (n, n'), e' \right> \in C(G, G') \\
\quad \neg(\text{prlb}(n) = \text{prlb}(n') \land \text{depth}(n) = \text{depth}(n')) \}
\]

The widening focuses on those pairs of $\text{TC}$ that corresponds to growth of $G'$ relative to $G$, that is, pairs $(n, n')$ where $n'$ is not any and either have a different, and thus larger, prlb-set or lies further from the root than $n$.

\(^{15}\)To say that $(n, n')$ introduce $(m, m')$ corresponds to the “function” $\text{ca}(m, m')$ in [30]. This is, however, not a function as there is no unique introducing pair, a pair may be introduced by more than one other pair (through back-ward arcs).
Methods using Type Graphs

Definition 5.4.3 (Widening Topological Clashes) Define the set of Widening Topological Clashes as\(^\text{16}\):

\[
\text{WTC}(G, G') = \\
\{ \langle (n, n'), e' \rangle \mid \langle (n, n'), e' \rangle \in \text{TC}(G, G') \\
\text{and} (n' \neq \text{any}) \\
\text{and} (\text{depth}(n) = \text{depth}(n') \wedge \text{prlb}(n) \neq \text{prlb}(n')) \\
\text{and} \text{depth}(n) < \text{depth}(n') \}\]

The widening will then transform \(G\) such that the only Widening Topological Clashes that remain are those where \(n\) and \(n'\) have the same depth (thus the \(\text{prlb}\)-set of \(n'\) is strictly larger than that of \(n\) and therefore \(n'\) is an or-node) but there is no proper ancestor \(n_{anc}'\) of \(n'\) with \(\text{prlb}(n') \subseteq \text{prlb}(n_{anc}')\)

A clash \(\langle (n, n'), e' \rangle\) in \(\text{WTC}\), introduced by \((m, m')\), that should be removed can be resolved in one of two ways, informally,

**Arc Replacement** This is the preferred way. This is possible when the introducing arc can be rerouted to a proper ancestor \(n_{anc}'\) of \(n'\) such that \(n' \subseteq n_{anc}'\) with \(\text{depth}(n) \geq \text{depth}(n_{anc}')\). In this case replacing the (introducing) arc \(e' = (m', i, n')\) with a (backward) arc \(e'' = (m, i, n_{anc}')\) will cause \((m, m')\) to introduce \((n, n_{anc}')\) via \(e''\) instead of \((n, n')\). By requiring that \(\text{depth}(n) \geq \text{depth}(n_{anc}')\) the tuple \(\langle (n, n_{anc}), e'' \rangle\) will not be a Widening Topological Clash, that is, the size of \(\text{WTC}\) will decrease.

**Node Replacement** If there is no suitable ancestor available for arc replacement then the clash will have to be resolved by selecting an ancestor \(n_{anc}'\) of \(n'\) that is not deeper than \(n\) and replacing \(n_{anc}'\), with a new node (and sub-graph) in a way such that safeness and termination is ensured. Safeness requires that the new node replacing \(n_{anc}'\) have at least as large concretization as that of \(n_{anc}'\). To ensure termination we also require that the size of the resulting type graph will decrease.

\(^{16}\)A less direct but simpler formulation is possible by the way \(\text{TC}\) is constructed. The condition \((\text{depth}(n) = \text{depth}(n') \wedge \text{prlb}(n) \neq \text{prlb}(n')) \vee \text{depth}(n) < \text{depth}(n')\) is equivalent to \(\text{depth}(n) \leq \text{depth}(n')\) when \(n' \neq \text{any}\). Also, since \(n \nsubseteq n'\) by construction and \(n' \neq \text{any}\) the condition \(\text{prlb}(n) \neq \text{prlb}(n')\) is equivalent to \(\text{prlb}(n) \not\subseteq \text{prlb}(n')\).
Note that, in some sense, arc replacement is just a particularly simple case of node replacement.

There are some additional conditions imposed on the above operations. Arc replacement is done when there is a clash \((n, n')\) in \(\text{WTC}(G, G') \downarrow 1\) that can be resolved by replacing (re-directing) an (introducing) arc from a node \(m'\) to \(n'\), with an arc to an ancestor of \(n'\) that safely approximates \(n'\).

To prevent nested or-nodes we have to ensure that we never re-direct an arc \((m', i, n')\) from an or-node \(m'\) (originally targeting a functor-node \(n'\)) to an or-node \(n'_{\text{anc}}\). Preventing nested or-nodes was not an issue in \([30]\) since the OR-cycle restriction used there ensured that a predecessor \(m\) of an conflicting node \(n'\) could never be an or-node.

**Definition 5.4.4 (Arc Replacements)** Define \(\text{CI}(G, G')\), the set of possible arc replacements as:

\[
\text{CI}(G, G') = \{ (e', n'_{\text{anc}}) \mid (n, n'), e' \in \text{WTC}(G, G') \\
\text{n'}_{\text{anc}} \in \text{anc}^+(n') \quad \text{a proper ancestor} \\
\text{depth}(n'_{\text{anc}}) \leq \text{depth}(n) \quad \text{not deeper than } n \\
\text{a safe approximation, implies } \text{prlb}(n') \subseteq \text{prlb}(n'_{\text{anc}}) \\
\text{n'} \subseteq n'_{\text{anc}} \\
\text{prevent nested or-nodes, here } e' = (m', i, n') \\
\text{lb}(e' \downarrow 1) = \text{or} \implies \text{lb}(n'_{\text{anc}}) \neq \text{or}
\}
\]

Transforming \(G'\) by selecting an entry \((e', n'_{\text{anc}}) \in \text{CI}(G, G')\) and re-directing \(e'\) to \(n'_{\text{anc}}\) will make the transformed graph safely approximate the original graph \(G'\). Each such arc replacement will decrease the size of both \(\text{CI}\) and \(\text{WTC}^{17}\). Repeated transformation will thus eventually produce a \(G''\) from the original \(G'\) such that \(G \sqsubset G' \sqsubseteq G''\) and \(\text{CI}(G, G'') = \emptyset\).

The arc replacement procedure (replace\_arc\((G', e', n'_{\text{anc}})\), not shown) simply obtain a new graph from \(G'\) by replacing the arc \(e' = (m', i, n')\) with \(e'_{\text{anc}}\).
an arc \((m', l_n', n_{anc}')\). This will, in general, make nodes unreachable and thus make the new graph smaller.

Node replacement is used as a last resort when there are widening clashes in WTC that must be resolved but where simple arc replacement is not possible.

**Definition 5.4.5 (Node Replacements)** Define \(CR(G, G')\), the set of possible node replacements, as:

\[
CR(G, G') = \{ (n', n_{anc}') \mid (n, n') \in WTC(G, G') \downarrow 1 \quad n_{anc}' \in \text{anc}^+(n'), \exists n_p' : (n_p' = \text{parent}(n_{anc}'), \text{lb}(n_p') = \text{or}) \\
((\text{depth}(n) = \text{depth}(n') \land \text{prlb}(n') \subseteq \text{prlb}(n_{anc}')) \\
\lor \\
(\text{depth}(n) < \text{depth}(n') \land \text{depth}(n_{anc}') \leq \text{depth}(n))) \}
\]

\[\square\]

Thus \(CR(G, G')\) consists of pairs \((n', n_{anc}')\) for \((n, n') \in WTC(G, G') \downarrow 1\), satisfying either of the following two requirements:

**Definition 5.4.6 (Disallowed Topological Clashes)**

- **functor clash** \(n\) and \(n'\) have the same depth (and thus \(\text{prlb}(n) \subset \text{prlb}(n')\)) and there is also an ancestor \(n_{anc}'\) with a \(\text{prlb}\)-set at least as large as that of \(n'\).

- **depth clash** the new node \(n'\) is further from the root than \(n\). Such clashes must always be removed so there is no further requirement on the ancestors. Select some ancestor close enough to the root, that does not have an or-node as parent, this is always possible.

\[\square\]

In both these cases, and also for arc replacement, there may be more than one suitable ancestor. Hentenryck et al. did not describe any selection criteria, The heuristic used in this thesis attempts to select an ancestor that is as similar as possible to \(n'\), see Section 5.4.3 for details.

The node replacement transformation then replaces one such \(n_{anc}'\) (and thus all its descendants) with a new node \(n''\) (with accompanying descendants)
while at the same time ensuring that the size of the transformed graph $G''$ is strictly less than the size of the original graph $G'$. To ensure that $G''$ safely approximates $G'$ we must also have $n'_\text{anc} \subseteq n''$.

The simplest way to obtain a suitable $n''$ is to use any. As pointed out by Henenryck et al., a more precise result is possible by using some heuristic to obtain a suitable $n''$ while still ensuring that the size of the type graph decreases. The method used to obtain such an $n''$ is crucial to the precision of the analysis since using any will give low precision. It is not clear what heuristics, if any, was used by Henenryck et al. to obtain an appropriate $n''$.

In the description by Henenryck et al. they also require $n' \subseteq n''$, that is, $n''$ should be an upper bound of $n'$ and $n'_\text{anc}$. This is not required and was not motivated. Intuitively it makes sense if we assume that the new subgraph consisting of $n''$ and its descendants (all of which are new nodes) have a similar structure to the replaced subgraph consisting of $n'_\text{anc}$ and the nodes below it. The requirement that $n''$ should be an upper bound of $n'$ and $n'_\text{anc}$ should make it easier to find an ancestor with larger concretization so that arc replacement can be applied in subsequent transformation steps.

If such an upper bound should be non-trivial, that is, not any, then we need an upper bound operation that can be applied on two nodes in the same type graph, neither of which may be the root. The upper bound must at the same time decrease the size of the resulting type graph, but, as discussed in Section 3.3.3.1, upper bound tend to produce results much larger than its arguments, especially for type graphs, so it is not obvious how an upper bound operation that ensures shrinkage could be obtained.

As mentioned above no method to achieve this was discussed by Henenryck et al. Several heuristics were tried most based on the following idea. First compute $n''$ as the ordinary upper bound $n' \cup n'_\text{anc}$ and perform the replacement obtaining $G''$. Typically $G''$ would now be larger than $G'$. Second, apply one or more shrinkage heuristics on $n''$ and its descendants or on the whole $G''$ that will decrease the size of the type graph in a safe manner. Should these heuristics fail to decrease the size of the type graph sufficiently then, as a last resort, any is used for $n''$ as in the naive method. The implementation of procedure replace_node($G', n', n'_\text{anc}$) is described in Section 5.4.3.

By removing the, not strictly necessary, requirement that $n''$ should be an upper bound of $n'$ makes it possible to avoid computing a possibly huge upper bound and instead apply the shrinkage heuristics directly on $n'_\text{anc}$.

\[18\] This was implemented but not systematically evaluated. The alternative method in Section 6.6.1 avoids replace node completely and appears to be a better and more precise method.
Finally we can define the topological-clash widening on type graphs,

\[
\text{widen}_{TC}(G, G') =
\begin{cases}
\text{widen}_{TC}(G, \text{replace}_\text{arc}(G', e', n'_{\text{anc}})) & \text{if there is a } (e', n'_{\text{anc}}) \in CI \\
\text{widen}_{TC}(G, \text{replace}_\text{node}(G', n', n'_{\text{anc}})) & \text{if } (n', n'_{\text{anc}}) \in CR, CI = \emptyset. \\
G' & \text{if } CI = \emptyset, CR = \emptyset
\end{cases}
\]

Correctness is proved in Appendix B. This includes a proof, adapted with corrections from [30], of the intriguing way in which stationarity (Definition 2.1.3) and thus termination of the analysis is ensured by this widening.

5.4.3 Implementation

The implementation is similar in structure to the containment operation. The type graphs are traversed in lock step depth first and when a topological clash occurs then descent is stopped along this path.

When a topological clash occurs and the ancestor selection, described in the next section, determines that the clash must be resolved and that it can be resolved with arc replacement then the arc replacement is performed in place, and traversal is continued with the remaining parts of the type graph. If arc replacement cannot be used then this clash is recorded and traversal continues, applying arc replacement wherever possible.

After the traversal is finished and if there are no recorded conflicts then all conflicts have been resolved and the widening terminates; if there are recorded conflicts but some arc replacements were performed then any recorded clashes are discarded and the algorithm iterates as the arc replacement may have made more arc replacements possible; if there are recorded conflicts and no arc replacement was done then the first of the recorded conflicts is resolved using node replacement any remaining recorded clashes are discarded and the algorithm iterates once again.

5.4.3.1 Choosing an Ancestor

When a topological clash occurs for a pair \((n, n')\) then an ancestor \(n'_{\text{anc}}\) of \(n'\) should be chosen, both to determine if the clash must be resolved and, if so, to select an appropriate node to use with \(n'\) in the arc replacement or node replacement transformations. The method I use tries to select an ancestor that is as similar to \(n'\) as possible.

Given a topological clash \((n, n')\) define \(N_0\) as the set of nodes with low enough depth and large enough \(\text{prlb}\)-set:

\[N_0 = \{ m \mid m \in \text{anc}^+(n'), \text{depth}(m) \leq \text{depth}(n), \text{prlb}(n') \subseteq \text{prlb}(m) \}\]
If $N_0$ is non-empty then select as $n'_\text{anc}$ the first $m \in N_0$ according to the following order (breaking ties by preferring deep $m$),

- $m$ equivalent to $n'$ ($n' \subseteq m, m \subseteq n'$), or
- $m$ with smallest $\text{prlb}$-set such that $n' \subseteq m$, or
- any $m$

For $(n, n') \in TC \downarrow 1, n' \neq \text{any}$ such that $\text{depth}(n) = \text{depth}(n')$ the existence of such an $n'_\text{anc} = m$ implies that $(n, n') \in WTC \downarrow 1$ and that the conflict is a functor clash that should be resolved using this $n'_\text{anc}$. If $n' \subseteq n'_\text{anc}$, then arc replacement can be used unless it creates a nested or-node, otherwise node replacement must be used. If, on the other hand, no such $m$ exists then this conflict should not be resolved and $G'$ is allowed to grow at $n'$.

Conflicts $(n, n') \in TC \downarrow 1, n' \neq \text{any}$ such that $\text{depth}(n) < \text{depth}(n')$ are depth clashes and should always be resolved. In this case, if there exists an $m$ as above then use it for arc replacement or node replacement as appropriate. If, on the other hand, no such $m$ exists, then node replacement must be used. In the latter case I use the ancestor $n'_\text{anc}$ of $n'$ such that $\text{depth}(n'_\text{anc}) = \text{depth}(n)$.

Hentenryck et al. did not describe what selection criteria they used.

5.4.3.2 Node Replacement Heuristic When a widening topological clash cannot be resolved using arc replacement then node replacement is performed instead, replace-node Algorithm 5.4.7. Its purpose is to combine a node $n$ with one of its ancestors $n_\text{anc}$. The main problem is to do this while reducing the size of the type graph and without losing too much precision.

The requirement that the size should decrease is needed to ensure termination of the widening operation itself. Size was taken to be the number of nodes of the type graph (as always all type graphs are assumed compact and thus reduced as well as normal and deterministic).

Other measures could be used as well, one possibility would be to only count functor-nodes, so as not to penalize the introduction of or-nodes when shrinking the type graph.
Algorithm 5.4.7 (Node Replacement)

\[
\text{replace\_node}(G, n, n_{\text{anc}}):
\]

1. if \( n_{\text{anc}} \subseteq n \) and rooted\((n) \) then
2. \hspace{1em} replace \( n_{\text{anc}} \) in \( G \) with \( n \) to obtain \( G' \)
3. \hspace{1em} return \( G' \)

4. if \( n \subseteq n_{\text{anc}} \) or not use\_upp then
5. \hspace{1em} \( m := n_{\text{anc}} \)
6. \hspace{1em} \( H_0 := G \)
7. else
8. \hspace{1em} \( m := n \cup n_{\text{anc}} \)
9. \hspace{1em} replace \( n \) in \( G \) with \( m \) to obtain \( H_0 \)
10. if \( |H_0| < |G| \) then return \( H_0 \)
11. minimize \( H_0 \) at and below \( m \) to obtain \( H_1 \)
12. if \( |H_1| < |G| \) then return \( H_1 \)
13. \( w := \text{jungle}(m) \)
14. replace \( m \) in \( H_1 \) with \( w \) to obtain \( H_2 \)
15. if \( |H_2| < |G| \) then return \( H_2 \)
16. give up
17. replace \( w \) in \( H_2 \) with any to obtain \( H_3 \)
18. return \( H_3 \)

\( \square \)

The first heuristic, at line 1–3, is to check for the special case when the ancestor is approximated by the descendant \( n \). This is the inverse of the condition utilized by the arc replacement transformation. However, the treatment in replace\_node is not as straightforward. If there are backward arcs going from \( n \) or one of its descendants targeting a node on the forward path between \( n_{\text{anc}} \) and \( n \) then replacing \( n_{\text{anc}} \) with \( n \) would violate the tree shape requirement on type graphs and re-establishing the tree shape may increase the size of the type graph. I use a somewhat more restrictive condition and check if \( n \) is rooted, that is, that there are no backward arcs from a descendant of \( n \) to a proper ancestor of \( n \).

If the first heuristic does not apply then a number of shrinkage heuristics is used. By default the shrinkage is applied after having replaced \( n_{\text{anc}} \) with \( m = n \cup n_{\text{anc}} \) but, as discussed in Section 5.4.2, it is also possible to refrain from performing the upper bound and the initial replacement and using \( m = n_{\text{anc}} \) instead.
The following shrinkage heuristics are then applied, in order, and if any of them lead to a result with smaller size than the original type graph then no further work is needed. Note that the parent of \( n_{\text{anc}} \), if it exists, is a functor-node so it is possible to use any kind of node to replace \( n_{\text{anc}} \).

- None at all, the upper bound may have caused a decrease in size already.

- Minimization at \( m \) and below, line 14. That is, replacing each forward arc to a descendant of \( m \) with a backward arc to an equivalent ancestor node if possible. Note that this may be applicable even if the upper bound operation itself applies minimization since some proper ancestor of \( m \) may be used as well.

- A (type graph corresponding to a) type-jungle, Section 5.2, that approximates \( m \), is created and used to replace \( m \).

- If the type graph is still not small enough an any-node is used, as in the naive method. In practise this happens rarely, but it does happen for many benchmarks. The impreciseness caused by introducing any-nodes is avoided in the alternative formulation, based on direct construction, presented in Section 6.6.1.

A number of other possibilities exists. After, or instead of, minimization it would be possible to apply arc replacement at and below \( m \), that is, to replace forward arcs to ancestors that are not necessarily equivalent but that may have strictly larger concretization, see also Section 6.2. As a final method before using an any-node it would be possible to use the same approximation as in the functors-widening, Section 5.1.

It would also be possible to apply some of these shrinkage heuristics on other parts of the type graph and not just consider the descendants of \( m \).

### 5.4.4 Performance

Table 5.8, summarizes the resource usage on the synthetic benchmarks using the Topological Clash widening. For the synthetic benchmarks only the replace arc transformation applies so these results are unaffected by the heuristics in replace\textunderscore node.

As can be seen, for tree\(_n\), the size of the largest type graph that occurs during analysis appears to grow at least exponentially with \( n \), while the benchmark expr times out after having encountered a huge type graph. The largest minimized type graphs are smaller but still have (at least) exponential size in \( n \). As for the other widenings, unfair scheduling causes a larger number of iterations and widenings.
Methods using Type Graphs

|           | Iterations FAIR | max size FAIR | max $|D|$ FAIR |
|-----------|-----------------|---------------|------------|
| tree$_2$  | 7               | 4             | 3          |
| tree$_3$  | 10              | 34 (19)       | 6          |
| tree$_4$  | 13              | 152 (49)      | 9          |
| tree$_5$  | 16              | 686 (125)     | 12         |
| tree$_6$  | 19              | 3128 (303)    | 15         |
| tree$_7$  | 22              | 14316 (713)   | 18         |
| tree$_8$  | 25              | 65370 (1673)  | 21         |

Table 5.8: Topological-Clash widening on synthetic benchmarks. Number of iterations for the FAIR worklist scheduling policies (FIFO, LRF, TWO-PHASE) and the unfair policies (FIXED, LIFO). Number of widenings of success patterns ($\nabla_{SP}$). Number of nodes of the largest type graph encountered during analysis (max size), size of largest minimized type graph in parenthesis. Largest number of non-equal nodes in any type graph max $|D|$. Benchmark expr exhausted time limits, the largest type graph encountered before interruption is shown.

The largest number of non-equivalent nodes in any type graph, (max $|D|$), grows only linearly. The linear number of distinct nodes means that, for these benchmarks, this widening is much better than Janssens’ widening at limiting the number of uselessly precise nodes.

5.4.4.1 More on Scheduling Policy Table 5.9 shows the result when the predicates are transformed so that non-recursive clauses come last. As discussed in Section 5.2.3 this gives the worklist scheduling policy more control over when clauses are analyzed, and for the synthetic benchmarks it is equivalent to disabling eager analysis of calls (See Section 4.1.3.2, page 74).

The unfair scheduling policies perform much worse in terms of the number of iterations and widenings but compensate this by type graph sizes that are, even asymptotically, much smaller.

No explicit minimization is done. Nevertheless, for the unfair scheduling policies the sizes shown are the same as the size of the largest minimized type graphs. For the fair scheduling policies the largest minimized type graphs are much smaller but still exponential in $n$.

This counterintuitive correlation between poor worklist policies and good domain behaviour is similar to what could be observed with the type-jungle widening, and is opposite to the case with Janssens’ widening.
5.4. Topological Clash Widening

<table>
<thead>
<tr>
<th>Tree</th>
<th>Fixed</th>
<th>LIFO</th>
<th>UNFAIR</th>
<th>Fixed</th>
<th>UNFAIR</th>
<th>Fixed</th>
<th>UNFAIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree₂</td>
<td>9</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>tree₃</td>
<td>13</td>
<td>19</td>
<td>14</td>
<td>6</td>
<td>7</td>
<td>34</td>
<td>17</td>
</tr>
<tr>
<td>tree₄</td>
<td>17</td>
<td>34</td>
<td>25</td>
<td>8</td>
<td>11</td>
<td>152</td>
<td>32</td>
</tr>
<tr>
<td>tree₅</td>
<td>21</td>
<td>55</td>
<td>41</td>
<td>10</td>
<td>16</td>
<td>686</td>
<td>54</td>
</tr>
<tr>
<td>tree₆</td>
<td>25</td>
<td>83</td>
<td>63</td>
<td>12</td>
<td>22</td>
<td>3128</td>
<td>85</td>
</tr>
<tr>
<td>tree₇</td>
<td>29</td>
<td>119</td>
<td>92</td>
<td>14</td>
<td>29</td>
<td>14316</td>
<td>126</td>
</tr>
<tr>
<td>tree₈</td>
<td>33</td>
<td>164</td>
<td>129</td>
<td>16</td>
<td>37</td>
<td>65370</td>
<td>180</td>
</tr>
<tr>
<td>expr</td>
<td>696</td>
<td>815</td>
<td>121</td>
<td>≥ 10M</td>
<td>≈ 2800</td>
<td>≥ 32</td>
<td>31</td>
</tr>
</tbody>
</table>

Table 5.9: Topological Clash widening, non-recursive clauses last, on synthetic benchmarks. Number of iterations for the FAIR worklist scheduling policies (FIFO, LRF, TWO-PHASE) and the UNFAIR policies FIXED, LIFO. Number of widenings of success patterns (\( \nabla SP \)). Number of nodes of the largest type graph encountered during analysis (max size) (For expr FIXED 2742, LIFO 2910). Largest number of non-equivalent nodes, max [D].

5.4.4.2 Single-Clausal Form  Transforming the synthetic benchmarks to single clausal form will enable the topological clash widening to reach the final, optimal result with just two non-trivial widening steps.

Example 1 (Topological Clash and Single-Clausal Form) This example illustrates how the Topological Clash widening will arrive at the final, optimal, type using only two non-trivial uses of widening when the synthetic benchmarks are rewritten to single-clausal form.

main :- tree(_Any).  
    tree(_a).  
    tree(b(_X)) :- tree(_X).  
    tree(c(_X)) :- tree(_X).  

During the first iteration over all the clauses of tree the non-recursive clause will contribute

\[ T₁ \rightarrow a \]

and this also becomes (the type component of) the first success pattern \( T₁^{SP} \).

In the second iteration the success pattern of the recursive clauses will be

\[ \begin{align*}  
    T₂ & \rightarrow b(T₁^{SP}) \\
    T₃ & \rightarrow c(T₁^{SP})  
\end{align*} \]
The upper bound of the success patterns from all clauses becomes
\[ T^2 \rightarrow a \mid b(T^1_{SP}) \mid c(T^1_{SP}) \]

The widening \( T^1_{SP} \uplus T^2 \) will encounter a functor clash at the root with \( \text{prlb}(n) = \{a\} \) and \( \text{prlb}(n') = \{a, b, c\} \) and since \( n' \) is the root there is no ancestors with a containing prlb-set so the type will be allowed to grow giving \( T^2_{SP} = T^2 \).

In the third iteration the success pattern of the recursive clauses will be \( T^3_2 \rightarrow b(T^2_{SP}) \) and \( T^3_3 \rightarrow c(T^2_{SP}) \). The upper bound for all clauses then becomes \( T^3 \rightarrow a \mid b(T^2_{SP}) \mid c(T^2_{SP}) \). There will not be a functor clash at the root but there will be at the nodes corresponding to \( T^2_{SP} \) where there will be a functor clash with \( \text{prlb}(n) = \{a\} \) and \( \text{prlb}(n') = \{a, b, c\} \).

In this case however there will be an ancestor with a containing prlb-set, the root. Since the root is also strictly less precise than the clashing node the arc replacement transformation applies and the result from the widening becomes \( T \rightarrow a \mid b(T) \mid c(T) \) giving \( T^3_{SP} = T \).

In the fourth and final iteration there will be no changes, a fixpoint has been reached and the analysis is done. Note that the number of widenings is the same regardless of the number of clauses (and thus functors of the data structure) of \( \text{tree}_n \) in this case.

\[ \Box \]

As should be clear from the example the surprisingly good result when using single clausal form is due entirely to the specifics of this widening. For unrelated reasons single clausal form also improved the performance with type-jungle widening but made the performance much worse for Janssens’ widening.

The example also highlights another useful fact. The nodes deeper in the type graph tend to correspond to types created at earlier iterations. Since call and success patterns become increasingly imprecise as the analysis proceeds this means that deeper nodes tend to be comparable to and more precise than their ancestors. A transformation similar to arc replacement can therefore often replace entire subtrees with backward arcs, thus significantly decreasing the size by introducing recursive types. A method based on this observation is defined in Section 6.2. In Section 7.6 this method is shown to more precise than any other method in this thesis on the larger benchmarks.

---

\[ 19 \] This is abusing notation, the corresponding type graph cannot share \( T^1_{SP} \) as this would violate the tree shape requirement.
5.4.5 Precision

The Topological Clash widening is in theory strictly more precise than any of the other widenings in this thesis. Since the widening will not a-priori disallow any type in the result in fact, any type graph could become the result of the widening provided that the type graph does not grow too much relative the previous approximation.

For the \texttt{list\_of\_lists} predicate the result becomes the optimal

\[
\begin{align*}
T^*_{il} &\rightarrow \emptyset \\ T^*_{i} &\rightarrow \emptyset \mid (\text{num}, T^*_i)
\end{align*}
\]

This is the only widening considered in this chapter that is able to precisely capture the success pattern of this predicate. We will see in Chapter 7 that this widening is, also in practice, among the most precise of the widening methods considered in this thesis. However, another formulation that never has to resort to introduce any-nodes gives even more precise results. That formulation uses minimized grammar graphs and is therefore presented separately, in Section 6.6.1.

5.4.6 Summary and Conclusions

I described and evaluated the topological clash widening of Hentenryck et al. [30]. The description in this section is more concrete than theirs in that it describes the important heuristic parts of the widening. The description is also more general in that it allows the widening to operate directly on the compact type graphs used in this thesis. Hentenryck et al. required that backward arcs only go to or-nodes, a requirement that can, in the worst case, lead to exponentially larger type graphs. In addition the termination related proofs of Hentenryck et al. were adapted and corrected (in Appendix B).

In the evaluation the synthetic benchmarks \texttt{expr} and \texttt{tree} was used to determine the performance of this widening when used to infer recursive types. The evaluation shows that this widening is extremely sensitive to the worklist scheduling policy, or, in general, to the order clauses are analyzed and when widening is applied. These issues were not discussed by Hentenryck et al. [30]. They indicate that they use the GAIA analyzer framework [6] with, what they call, prefix optimization [20].

This would seem to indicate that their analyzer, even though it does not use an explicit worklist, would perform the same sequence of upper bounds and widenings as the FAIR schedulings in Table 5.8 and would encounter the same huge, unminimized, and exponentially growing type graphs.

Note that only the arc replacement transformation is used for the synthetic benchmarks so any heuristics they may have used to implement the node replacement transformation could not improve the situation.
These are the main findings, \( n \) refers to the number of clauses of tree, that is the number of function symbols used to define the recursive type.

- The default analysis method used by the analyzer, that is, when widening is applied after each analyzed clause and eager analysis of calls, will cause the type graphs to grow exponentially with \( n \). This would also seem to apply to this widening in combination with the analyzer used by Hentenryck et al.

- In all cases the fair scheduling methods require a linear number of iterations and widenings which is asymptotically optimal. The unfair scheduling policies perform much worse in this respect, with a quadratic to cubic number of iterations and widenings.

- If eager analysis is not used then the unfair scheduling policies require even more iterations and widenings, cubic and quadratic respectively, but the size of the largest type graph is now “only” cubic in \( n \). The fair scheduling policies behave as badly as before with a linear number of iterations and widenings, and exponentially large type graphs.

- Postponing the widening and the update of the global success pattern until the results of all clauses are available will make the analysis require a constant number of widenings and linear sized type graphs to infer the recursive types used in the synthetic benchmarks. This can be achieved by transforming the predicates to single clausal form.

- In all cases the largest number of non-equivalent nodes in any type graph is small and grow linearly with \( n \). The excessive sizes are thus entirely due to the tree shape imposed on type graphs.

These findings show that optimal efficiency of an analyzer, measured as the number of iterations, can be, not only irrelevant, but also counterproductive, for the kind of complex and costly domains considered in this thesis.

The findings also show that the tree-shape requirement imposed on type graphs will, using otherwise reasonable analyzer frameworks, require a representation that is exponential in the number of non-equivalent nodes.

The findings also show that it may indeed be possible to engineer this widening to be practical. The performance on the realistic benchmarks are presented in Chapter 7.

An alternative formulation of the topological clash widening appears in Section 6.6.1. It has the same performance characteristics when inferring the recursive types of the synthetic benchmarks. Its advantage is that it directly constructs the desired type graph and thus never has to use the replace-node algorithm. For this reason it never has to fall back to using

\[^{20}\text{At least not for any of the benchmarks.}\]
any-nodes and the evaluation in Chapter 7 shows that the alternative direct construction method in Section 6.6.1 gives more precise results than the implementation presented in this section.

5.5 SUMMARY AND CONCLUSIONS

Abstract domains based on term grammars have infinite ascending chains and therefore require that a widening is used (Section 2.1.3). The widening is not only responsible for ensuring a finite number of iterations, its ability to keep the size of the grammar representation within reasonable bounds is also a critical property.

Type graphs where proposed by Janssens and Bruynooghe [32] and can be viewed as a concrete representation of term grammars. An abstract domain based on type graphs were implemented as part of an analyzer for Prolog [31].

A number of widenings for the type graph representation of term grammars were investigated in this chapter.

The main reason to use term grammars as an abstract domain is that this gives the analyzer the ability to infer the structure (or type) of recursively defined data structures. Of particular importance is therefore how efficiently the analyzer can infer the types of programs that manipulate such data structures.

To investigate this each widening was used to analyze the synthetic benchmarks tree and expr. These benchmarks manipulate recursive data structures with increasing number of function symbols (sometimes known as constructors). The efficiency when analyzing these programs illustrate the behaviour in the ideal case when no precision is lost. The results were discouraging. The type graphs that appear during analysis tend to grow very quickly with the number of function symbols. In many cases the growth appears to be at least exponential in the number of function symbols. The reason for this excessive growth can be found in both properties of the type graphs and properties of the widenings. The first reason is inherent in the type graph representation. In the worst case the tree-like shape imposed on type graphs can make a type graph exponentially larger than the term grammar it represents.

The impact of the type graph representation could be seen for the “type-jungle” widening in Section 5.2. The type-jungle widening will always create a grammar graph that has linear size in the number of function symbols. However, when used as a type graph widening the grammar graph must be converted to a type graph. To obtain the proper tree-like shape multiple equivalent nodes must be created. This conversion results in an exponentially larger type graph when analysing the synthetic benchmarks.
Another reason for types with impractical size could be seen with the widening used by Janssens and Bruynooghe, Section 5.3. In this case it appears that also the number of non-equivalent nodes increase exponentially with the number function symbols. This means that with Janssens’ widening any representation of term grammars would need excessive size to represent the types that occur during analysis. It should be noted that a more precise handling of variables, as was done in the integrated types used by Janssens and Bruynooghe, would make the problem with their method even worse.

First of all it would not help for the synthetic benchmarks since they create ground terms, secondly, it would make it less likely that precision is lost when analyzing the larger benchmarks. As was discussed in Section 4.4 many of the larger benchmarks manipulate recursive data structures but the precision of the analysis will be too low to make this a problem even for the widening of Janssens and Bruynooghe. However, with integrated type this is likely to become a problem also for some of the benchmarks. The behaviour of Janssens’ widening makes it unsuitable when inferring the types of recursively defined data structures.

Hen tenryck, Cortesi and Le Charlier proposed an alternative widening for type graphs [30], Section 5.4. Also with this widening the analyzer encounters what appears to be exponentially large type graphs. However, applying the widening less often completely eliminates this problem.

By default the analyser in Section 4.1 will apply the widening as soon as the result (success pattern) from one clause is available. For the widening of Hen tenryck et al. this leads to exponentially large type graphs.

An alternative method is to combine the results from all clauses using upper bound and then apply the widening once when the clause results are combined with the previous success pattern for the predicate. This was implemented in the analyzer by rewriting each predicate as a single clause with a large disjunction corresponding to the original clauses. When the synthetic programs are rewritten to this single-clausal form analysis using the widening by Hen tenryck et al. require a constant number of iterations to infer the recursive types of the synthetic benchmarks. Even more important, the size of the type graphs are now only linear in the number of function symbols.

The conclusion for the widening by Hen tenryck et al. is that, at least for programs with a simple structure it could lead to an efficient method for inferring recursive types.

The fact that a type graph may be exponentially larger than the corresponding term grammar is still troublesome. This property means that analysis always runs the risk of encountering huge types regardless of the widening used.
The next chapter will investigate methods that use grammar graphs, and in particular grammar graphs of minimal size. Although not presented in this way grammar graphs can be viewed as type graphs where the troublesome tree-shape is not enforced.
This chapter discusses widenings and other methods that use grammar graphs directly, without the tree shape requirement of the type graphs used in the previous chapter. In particular minimized grammar graphs will be investigated. That is, grammar graphs where no two nodes are equivalent. All of the techniques and widenings would also work if the grammar graphs were not minimal, this was however not investigated.

The results in the previous chapter show that type graphs have a tendency to be exponentially large in the number of non-equivalent nodes. By only maintaining the non-equivalent nodes it should therefore be possible to obtain grammar graphs with, in comparison, negligible size.

Algorithm 3.3.6 (Page 58) and grammar graph minimization makes it possible to convert back and forth between minimized grammar graphs and type graphs. It is therefore possible to use the type graph widenings also with minimized grammar graphs (and vice versa).

For widenings that can instead be formulated directly on (minimized) grammar graphs the huge type graphs will not be an issue at all, not even in the widening. Several widenings defined directly on minimized grammar graphs will be investigated in this chapter.

Using minimized grammar graphs will make all domain operations less expensive, including subsidiary operations such as minimization. One such operation, $\leq$-minimization, will be defined and investigated in Section 6.2. While ordinary minimization of type graphs will reroute a forward arc to a node $n$ to an equivalent ancestor $n_{anc}$ the $\leq$-minimization will use the more permissive condition $n \subseteq n_{anc}$. When used as a method to obtain type graphs from minimized grammar graphs this appears to be very good at reducing type graph size without undue loss of precision.
The rest of this chapter is organized as follows. First general issues are discussed such as adapting the domain operations and type graph widenings to minimized grammar graphs. Then $\square$-minimization is discussed in Section 6.2.

The chapter concludes with a number of sections discussing widenings. The type-jungle widening was described in the previous chapter but can also be applied directly on grammar graphs, is briefly reconsidered in Section 6.3. The type graph widenings are revisited in Section 6.4 and 6.6 where their combination with $\square$-minimization is also discussed. In 6.6.1 an implementation of the Topological Clash widening is described that ensures termination without the ad-hoc size requirement in Section 5.4 and also avoids creating an explicit upper bound. Section 6.5 discuss several generalization of Janssens’ widenings that focus on a spanning tree of a minimized grammar graph instead of the spanning tree of a type graph, and that consider other path properties than functor occurrences.

6.1 GENERAL ISSUES

6.1.1 Domain Operations for Minimized Grammar Graphs

All the grammar graph operations in Chapter 3 apply to minimized grammar graphs. The operations that create new grammar graphs must, however, be followed by a minimization step.

A rather naive method to minimize grammar graphs is used by the analyzer. When a minimized grammar graph is extended with new nodes then each of these is compared for equality with each of the pre-existing non-equivalent nodes. To speed this up somewhat a simple hashing scheme is used based on hashing the graph to a fixed depth. There are better methods [8] but this was simple to implement and not overly expensive, especially since the size of minimized grammar graphs tends to be small. Alternative implementations has not been evaluated but could probably give substantially more efficient methods.

For containment a method is used that caches the result of all the sub-comparisons. An optimal implementation of this would never have to compare two nodes more than once and the result of comparing all nodes in a

\[^{1}\text{Standard techniques for minimizing deterministic finite automata \cite{8, 55} should make it possible to minimize a normal deterministic term grammar in time }|N| \log |N| \text{ where } |N| \text{ is the number of non-terminals (assuming a constant number of function symbols). Another possibility would be to use the quadratic algorithm to compare all nodes for equality and then use union-find to group equivalence classes. Such an algorithm should be able to minimize a grammar (graph) in “almost” quadratic time.}\]
grammar graph $G$ can be obtained in $\Theta(|G|^2)$ time and space\(^2\), where $|G|$ is the number of nodes in $G$. The result could be represented as a boolean array indexed by pairs of nodes. Subsequent queries $n \subseteq n'$ could then be answered in constant time\(^3\).

Not only does this make subsequent comparisons cheap, it also makes it possible to use the containment test exhaustively to avoid creating new nodes in the upper bound and intersection operations. This avoids creating new nodes that would then have to take part in minimization.

The implementation of containment for grammar graphs used in the analyzer is based on similar ideas but computes containment for two nodes on demand instead of precomputing the $|G|^2$ pairs. The implementation does not use an optimal algorithm nor representation. It has not been systematically evaluated.

### 6.1.2 Adapting Type Graph Widening

Janssens' widening and the Topological Clash widening depend on the shape of type graphs. In this case the type graphs can be made explicit at the widenings while minimized grammar graphs can be used for all other domain operations. For Janssens' widening a hybrid is possible since the input to the widening can be a grammar graph while the output will be a type graph.

By using minimized grammar graphs all the domain operations can be made more efficient, also asymptotically. This can give a significant speedup, particularly since the other domain operations are much more common than widening. For upper bound the advantage is two-fold, not only will the upper bound operation be less costly, the desired minimization of the result will be too. The downside, of course, is the substantial cost of converting a type graph to a minimized grammar graph and vice versa.

If the type graph is only made implicit for the widening and not explicitly created then the widening operations can be sped up as well. The Topological Clash widening can take advantage of less expensive containment and upper bound operations, whereas Janssens' widening will be able to introduce backward arcs easier since there will be fewer possible (sub-)sets of nodes to consider on $ANC$, the list of ancestor sets.

If, however, the widening requires an exponentially large type graph then converting the representation, explicitly or implicitly, between minimized grammar graphs and type graphs will be the dominating cost.

\(^2\)This technique does not require that the grammar graph is minimal, the same idea can be used for type graphs and normal deterministic grammars as well.

\(^3\)Note that the asymptotic time complexity of comparing two nodes for containment is $\Theta(|G|^2)$ and thus the naive method to compare all $|G|^2$ pairs of nodes would require $\Theta(|G|^4)$ time.
For the Topological Clash widening the best settings required only polynomially large type graphs when inferring the recursive types of the synthetic benchmarks, using minimized type graphs should therefore be advantageous.

For Janssens’ widening the situation would unfortunately not improve enough when inferring the recursive types of the synthetic benchmarks. The number of non-equivalent nodes tends to grow exponentially and thus minimized grammar graph would not help, even though they could decrease the cost of both the widening and the other domain operations.

6.2 $\square$-Minimization

The type graphs created during analysis often have the property that nodes further from the root correspond to types created at earlier iterations. Since calls and success patterns get successively less precise this means that for a deep node there often exists a comparable and less precise ancestor. This was illustrated in the example in Section 5.4.4.2 and is a property utilized by the arc replacement transformation of the Topological Clash widening.

The $\square$-minimization transformation replaces any forward arc $(m,l,n)$ with a backward arc $(m,l,n_{\text{anc}})$ if $n \sqsubseteq n_{\text{anc}}$ ($n_{\text{anc}} \in \text{anc}^+(m), \text{lb}(n) = \text{lb}(n_{\text{anc}})$).

By requiring equal labels nested or-nodes are avoided as well as some loss of precision, especially when $n$ is a constant node.

Algorithm 6.2.1, \textit{construct}$_{\square}$, can be used to transform a grammar graph, and in particular a minimized grammar graph, to a type graph while at the same time replacing nodes with less precise ancestors. The algorithm is almost identical to Algorithm 3.3.6. The difference is that for $\square$-minimization a backward arc is introduced not only if there is an identical ancestor but also if there is an ancestor (with the same label) that is less precise.
Algorithm 6.2.1 (Construct<\subseteq>)

Construct a properly formatted type graph from a node of a grammar graph, replacing nodes with less precise ancestors when possible.

\text{construct}_{\subseteq}(n) = \text{build}(n, \emptyset).

\text{build}(n, \text{Seen}) =
\begin{align*}
&\text{if } n \text{ is a simple node then return } n \\
&\text{if } \exists (n', m) \in \text{Seen} : \text{lb}(n) = \text{lb}(n'), n \subseteq n' \text{ then} \\
&\quad \text{return } m \quad \text{a backward arc to containing } m \\
&\text{if } \text{lb}(n) = f \text{ then} \\
&\quad \text{create a new functor-node } m \text{ with } \text{lb}(m) = f \\
&\quad \text{Seen'} := \text{Seen} \cup \{(n, m)\} \\
&\quad \text{for each } i \in [1, \text{arity}(n)] \text{ do} \\
&\quad \quad m \downarrow i := \text{build}(n \downarrow i, \text{Seen'}) \\
&\text{else} \\
&\quad \text{create a new or-node } m \\
&\quad \text{Seen'} := \text{Seen} \cup \{(n, m)\} \\
&\quad \text{for each } f \in \text{prlb}(n) \text{ do} \\
&\quad \quad m \downarrow f := \text{build}(n \downarrow f, \text{Seen'}) \\
&\quad \text{end if} \\
&\text{return } m
\end{align*}

This algorithm does not always produce a minimized result, it is even possible that \( \subseteq \)-minimization might apply if the algorithm is iterated. The reason is that one pass of the algorithm may change the concretization of some nodes in such a way that previously incomparable nodes becomes comparable. New backward arcs could then be introduced. I have not investigated the impact of iterating this algorithm, it seems unlikely that it would matter in practice.

Algorithm 6.2.1 can be used to produce a \( \subseteq \)-minimized type graph that is an upper bound of two type graphs (or grammar graphs). This variant of upper bound, Algorithm 6.2.2, will be used both as a widening and to decrease the size of the type graphs when other type graph widenings are used.
Algorithm 6.2.2 (UppLEQ) Construct a \( \sqsubseteq \)-minimized type graph as the upper bound of two (not necessarily minimal) grammar graphs.

\[
\text{uppleq}(n, m) = \\
n' := \text{minimize}(n) - \text{no-op if } n \text{ is a minimized grammar graph} \\
m' := \text{minimize}(m) \\
r := n' \sqcup m' \\
\text{return } \text{construct}_\sqsubseteq(r)
\]

The \( \sqsubseteq \)-minimization upper bound that will be used in the evaluation together with type graph widenings only apply Algorithm 6.2.2 when the arguments are incomparable, that is:

Algorithm 6.2.3 (Type Graph \( \sqsubseteq \)-Minimization Upper Bound)
Upper bound of two type graphs using \( \sqsubseteq \)-minimization.

\[
T_0 \sqcup T_1 = \begin{cases} 
T_0 & \text{if } T_1 \subseteq T_0, \\
T_1 & \text{if } T_0 \subseteq T_1, \\
\text{uppleq}(T_0, T_1) & \text{otherwise}.
\end{cases}
\]

If this kind of upper bound is used with either Janssens’ widening or the Topological Clash widening then, when inferring the recursive type of the synthetic benchmarks, the widenings will never affect the types. Using \( \sqsubseteq \)-minimization of the result from upper bound is more aggressive than these widenings and, in effect, makes them superfluous. The use of Janssens’ widening with the upper bound in Algorithm 6.2.3 will be investigated in Section 7.3.1 when analyzing the larger benchmarks.

A “widening”\(^4\) using Algorithm 6.2.2 can be defined as in Definition 6.2.4 below. It will be used with type graphs as type domain\(^5\) in Section 6.2.2 and Section 7.6 to analyze the synthetic and the larger benchmarks respectively.

---

\(^4\)It is not a widening, see the next section.

\(^5\)Since the input to this method is a minimized grammar graph and the output is a type graph it would perhaps make even more sense to use it with (minimized) grammar graphs as domain, this was however not investigated.
Definition 6.2.4 (\(\sqsubseteq\)-Minimization “Widening”)

\[
T \sqsubseteq T' = \begin{cases} 
T' & \text{if } T = \bot \\
T & \text{if } T' \sqsubseteq T \\
\uppleq(T, T') & \text{otherwise (Algorithm 6.2.2)}
\end{cases}
\]

6.2.1 \(\sqsubseteq\)-Minimization Is Not a Widening

Using the method in Definition 6.2.4, that is, upper bound followed by \(\sqsubseteq\)-minimization is enough to guarantee termination for the synthetic benchmarks. A natural question is then if this will hold in general, that is, if \(\sqsubseteq\)-minimization can be used to obtain a widening as in Definition 2.1.2.

Clearly the method in Definition 6.2.4 terminates and gives conservative approximations. It remains to show that it would make any sequence of increasing types stationary. This is, however, not the case.

Consider the type \(T^k\) with finite concretization \(\gamma(T^k) = \bigcup_{i \leq k} \gamma_i(q(s^i(z)))\), so for example \(\gamma(T^2) = \{q(z), r(q(s(z))), r(r(q(s(z))))\}\), that is,

\[
\begin{align*}
T^k & \rightarrow R^k_0 \\
R^k_i & \rightarrow q(S^k_i) \quad (0 \leq i \leq k) \\
R^k_i & \rightarrow r(R^k_{i+1}) \quad (1 \leq i + 1 \leq k) \\
S^k_0 & \rightarrow z \\
S^k_i & \rightarrow s(S^k_{i-1}) \quad (0 \leq i - 1 < k)
\end{align*}
\]

None of the non-terminals (or right-hand sides) in this grammar are comparable, in particular this means that \(\sqsubseteq\)-minimization applied to the corresponding grammar graph or type graph would have no effect. Finally the sequence \(\ldots \sqsubseteq T^{(k-1)} \sqsubseteq T^k \sqsubseteq T^{(k+1)} \sqsubseteq \ldots\) is an infinite, strictly increasing, sequence of types, forming a counterexample against the claim that \(\sqsubseteq\)-minimization would be a widening\(^6\).

It is not clear that the above example or any other counterexample could occur within the particular framework used in this thesis. In fact \(\sqsubseteq\)-minimization can be used as if it were a widening, the analysis will terminate for all but one of the benchmarks\(^7\) of Section 4.4, see Section 7.6.

\(^6\)This acyclic counterexample is due to Sven-Olof Nyström, a more complex, cyclic, counterexample appears in Section 6.2.4.

\(^7\)Analysis exhausted memory for chat\_parser.
6.2. $\sqsubseteq$-Minimization

<table>
<thead>
<tr>
<th>Tree</th>
<th>Iterations $\cap SP$</th>
<th>Max Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fair, Fixed</td>
<td>Fair, Unfair</td>
</tr>
<tr>
<td>$\text{tree}_2$</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>$\text{tree}_3$</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$\text{tree}_4$</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>$\text{tree}_5$</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>$\text{tree}_6$</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>$\text{tree}_7$</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>$\text{tree}_8$</td>
<td>19</td>
<td>24</td>
</tr>
<tr>
<td>$\text{n}_{tree}$</td>
<td>&amp; $O(n)$</td>
<td></td>
</tr>
<tr>
<td>$\text{expr}$</td>
<td>32–35</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 6.1: “Widening” $\sqsubseteq$-minimization on synthetic benchmarks. The results are the same if minimization is done after upper bound, intersection and widening.

To obtain the benefits of this “widening” but with guaranteed termination $\sqsubseteq$-minimization can be used for upper bound (Algorithm 6.2.3) together with a real widening. See Section 7.3.1 where this is done for Janssens’ widening.

6.2.2 Performance

The “widening” of Definition 6.2.4 was used with the type graph domain. For technical reasons Definition 6.2.4 was also used to provide the upper bound used for disjunctions.

In this section the performance when inferring the recursive types of $\text{tree}_n$ and $\text{expr}$ is investigated. The performance on larger benchmarks is investigated in Section 7.6.

Table 6.1 shows the result for the synthetic benchmarks, analyzed with clauses separately. As can be seen all entries are small and linear in $n$, regardless of scheduling policy. What happens for $\text{tree}_n$ is that the eager analysis of $\text{tree}$ will reach the final success patterns for all but the first recursive clause. LIFO require more iterations since it will needlessly re-analyze clauses 3 to $n$ one extra time.

Example 2 ($\sqsubseteq$-Minimization “Widening” on $\text{tree}_n$) This example illustrates, using $\text{tree}_4$, how the $\sqsubseteq$-Minimization widening will arrive the fi-

---

8 As noted above it is an open question whether this will guarantee termination for the analyzer used in this thesis.

9 Note that this means that the upper bound used in disjunctions is unsymmetric in its arguments. While disjunctions are not used in the synthetic benchmarks it will affect the larger benchmarks analyzed in Section 7.6.
nal type corresponding to all but the first recursive clause during the initial analysis of the clauses.

```
main :- tree(_Any).
   tree(a).
   tree(b(X)) :- tree(X).
   tree(c(X)) :- tree(X).
```

The initial call from `main` will cause all clauses of `tree` to be analyzed (by `ai_goal`, Section 4.1.3.2).

Analyzing the first clause results in a success pattern for the first clause:

```
T_1 \rightarrow a
```

This also becomes (the type component of) the first success pattern of `tree`:

```
T_{sp}^1 \rightarrow a
```

Analyzing the second clause will give

```
T_2 \rightarrow b(T_{sp}^1)
```

Using the widening to combine $T_2$ with the existing success pattern ($T_{sp}^1$) will give the same result as ordinary upper bound, that is:

```
T_{sp}^2 \rightarrow a \mid b(T_{sp}^1)
```

The reason $\sqsubseteq$-Minimization will not affect the upper bound is that the successor of $b$ (that is $T_{sp}^3$) is a functor node (a) whereas the root of the type is an or-node. Recall that Algorithm 6.2.1 only will introduce a backward arc when the two nodes have the same label, that is, if they are both or-nodes or if they are both functor-nodes (with the same label).

Analyzing the third clause will give

```
T_3 \rightarrow c(T_{sp}^2)
```

The upper bound of $T_3$ and the previous success pattern ($T_3 \sqcup T_{sp}^2$) is

```
T_u \rightarrow a \mid b(T_{sp}^1) \mid c(T_{sp}^2)
```

Since $T_{sp}^2 \sqsubseteq T_u \sqcup T_{sp}^2$ and both are or-nodes applying $\sqsubseteq$-minimization will replace the argument of $c(T_{sp}^2)$ with a reference to the root of the new type and the widened result, the new success pattern for `tree`, becomes:

```
T_{sp}^3 \rightarrow a \mid b(T_{sp}^1) \mid c(T_{sp}^3)
```
The same thing will happen for the remaining clause(s) so after the initial eager analysis of all clauses the success pattern for `tree` is:

\[ T_{SP}^4 \rightarrow a | b(T_{SP}^4) | c(T_{SP}^4) | d(T_{SP}^4) \]

Since the success pattern changed all calling clauses will be reanalyzed, that is, all the recursive clauses of `tree` (and `main` but this will be ignored here). Unless LIFO scheduling is used the next analyzed clause will be clause two resulting in

\[ T_2' \rightarrow b(T_{SP}^4) \]

Upper bound of this result and the existing success pattern (\( T_{SP}^4 \)) followed by \( \sqsubseteq \)-Minimization will then give the final success pattern:

\[ T_{SP}^5 \rightarrow a | b(T_{SP}^5) | c(T_{SP}^5) | d(T_{SP}^5) \]

Since the success pattern changed all clauses that call `tree` will be analyzed once again but the success pattern will not change further and a fixpoint has been reached.

\[ \square \]

If clauses are reversed or eager analysis is not done (not shown), then the UNFAIR scheduling policies require quadratic number of iterations. The FAIR policies still require only a linear number of iterations but slightly more than in Table 6.1. For all scheduling policies the number of widenings and the sizes are the same as in Table 6.1.

These promising results show that it may be advantageous to use this “widening”, either by itself, or to decrease the cost of using some widening with known termination properties.

### 6.2.3 Precision

If \( \sqsubseteq \)-minimization is used with upper bound as if it was a widening then the analysis of `list_of_lists` will terminate and the result will be the optimal one. Interestingly the result will be optimal also if this upper bound is used together with Janssens’ widening with \( k = 2 \). What happens is that upper bound followed by \( \sqsubseteq \)-minimization will produce the optimal result and Janssens’ widening will then do nothing since the type only has the list-constructor two deep.

### 6.2.4 A Method by Sağlam & Gallagher

A more precise variant of a method used by Gallagher & de Waal [22] was used by Sağlam & Gallagher [50]. The original method, that will be
investigated in Section 6.5.5, restricts the paths in the (RUL-program\textsuperscript{10} representation of a) grammar such that no two non-terminals on a path can have the same principal labels.

The variant used by Sağlam disallows a path from a non-terminal \( N \) to a non-terminal \( N' \) only if they have the same \textit{prlb-set} and \( N' \subseteq N \).

This method could readily be implemented for grammar graphs and incorporated into the analyzer used in this thesis. However, the example in Section 6.2.1, is also a counterexample against the claim that the method used by Sağlam & Gallagher is a widening in the usual sense.

As noted in Section 2.1.3 it is possible to use a somewhat relaxed requirement on a widening. A widening need only guarantee termination for (increasing) sequences of approximations that can actually occur in the particular framework where it is used.

The method used by Sağlam & Gallagher is available as part of the bottom-up analysis framework by Gallagher [23]. To the authors knowledge this framework is the only publicly available system for type analysis of logic programs and Prolog and it has been used also by other researchers [19, 46].

The question then is if that specific framework prevents the problematic sequences. In fact Drabant [18] argues that the method will guarantee termination in Gallagher's framework. Unfortunately, this is apparently not the case. Using the counterexample presented below, Gallagher was able to construct a program for which his analyzer does not terminate\textsuperscript{11}.

\textbf{Example 3 (Sağlam's Shortening Is Not a Widening)} The example will be presented as a non-normalized term grammar. It can easily be put in normal form or be restated as a RUL-program. The idea is to construct a grammar \( G_n \), with start symbol \( S_n \), that contain a non-terminal \( L_i \) for \( 0 \leq i \leq n \) such that \( L_i \) will recognize lists of length at most \( i \).

For all these grammars the start symbol \( S_n \) is \( L_1 \), that is, the set of empty or one element lists where the only list element is described by \( L_n \).

If the list constructors would all have the same first argument then \( L_i \) would be strictly less precise than all \( L_j, j \leq i \), thus making the shortening introduce cycles.

\textsuperscript{10}Section 3.3.4

\textsuperscript{11}Personal communication, Feb 1999.
Instead the element types are made different and such that the element type of \( L_j \) is not contained in the element type of \( L_i, j \leq i \). In particular it is possible to use the \( L \)'s.

\[
\begin{align*}
S_n & \rightarrow L_1 \\
L_0 & \rightarrow [] \\
L_1 & \rightarrow [] \mid C_1 \\
    & \vdots \\
L_n & \rightarrow [] \mid C_n \\
C_i & \rightarrow n.(L_{i+n-1}, L_{i-1})
\end{align*}
\]

It is easy to see that none of the \( C_i \)'s are comparable and thus none of the \( L_i, i \geq 1 \). The only non-terminal that is comparable to any of the others is \( L_0 \), denoting the empty list.

Since the shortening only combines comparable non-terminals if they have the same principal functors the shortening will not combine \( L_0 \) with any of the others, and since none of the other non-terminals are even comparable the shortening will leave the grammar unaffected.

The sequence \( G_1, \ldots, G_i, \ldots \) with corresponding start symbols \( S_i \) is such that \( \ldots \sqsubseteq S_{i-1} \sqsubseteq S_i \sqsubseteq S_{i+1} \sqsubseteq \ldots \), that is, it is an infinitely increasing sequence. Should an analyzer ever encounter a sequence like this then the shortening will fail to ensure termination.

Adding productions \( L_i \rightarrow q(L_{n-i}) \) for \( 0 \leq i < n \) to \( G_n \) will ensure that not even \( L_0 \) is comparable to any of the others. This means that it will not help if the shortening is modified so that it also transforms the grammar when one prob-set is a subset of the other.

\[ \square \]

### 6.3 Type Jungle

The Type Jungle widening limits the size of a grammar graph by allowing at most one functor node for each function symbol. It is applicable directly on grammar graphs and give minimal grammar graphs as results.

The type-jungle widening was presented in detail in Section 5.2. There it was used with type graphs by converting the result from the type jungle widening from a grammar graph to a type graph. When the type jungle widening was used in this manner it produced exponentially large type graphs.

When used directly on (minimized) grammar graphs the behaviour is very good, see Table 6.2. For \( \text{tree}_n \) the number of iterations, widenings, and the size of the grammar graphs are all linear in \( n \).
The type jungle widening for minimized grammar graphs is evaluated in Section 7.2 on the realistic benchmarks.

6.4 JANSESSENS’ WIDENING REVISITED

Two methods were implemented to let Janssens’ widening take advantage of the compactness of minimized grammar graphs. The first method applies Janssens’ widening directly on minimized grammar graphs and is briefly outlined in the next section. The other idea is to use Janssens’ widening with \( \subseteq \)-minimization, this is described in Section 6.4.2 below and evaluated on the larger benchmarks in Section 7.3.1.

6.4.1 Using Grammar Graphs as Input

Janssens’ widening (in particular Algorithm 5.3.2, page 119) can be applied directly on, not necessarily minimal, grammar graphs. The result will be a type graph but the input need not be.

In Janssens’ widening sets of nodes are compared for equality to determine if a backward arc can be used (Algorithm 5.3.2, line 3). By using a minimized grammar graph instead of a type graph as input there will be fewer nodes and it is therefore more likely that the test for equal sets of nodes will succeed. This in turn makes it more likely that a backward arc is introduced and should therefore lead to smaller type graphs.

As mentioned previously other domain operations, in particular upper bound, is more efficient on (minimized) grammar graphs than if the tree-like shape of type graphs must be maintained.

Unfortunately, using minimized grammar graphs as input to Janssens’ widening does not improve the poor performance observed in Section 5.3.3 when

Table 6.2: Type-Jungle widening on synthetic benchmarks. Abstract domain is minimized grammar graphs.
inferring recursive types. For this reason this approach was not evaluated further.

6.4.2 $\square$-Minimization

Recall the algorithm Type Graph $\square$-Minimization Upper Bound (Algorithm 6.2.3, page 155). It can be used instead of the ordinary upper bound to make the type graphs smaller.

In Section 6.2.2 a “widening” using this technique was shown to give an efficient method for inferring the recursive types for the synthetic benchmarks tree$_n$ and expr.

The same effect is obtained when $\square$-minimization is performed as part of the upper bound operation. Janssens’ widening will therefore never need to transform its argument and analysis of the synthetic benchmarks becomes as efficient as with the $\square$-minimization widening in Section 6.2.2.

The advantage with using Janssens’ widening together with $\square$-minimization upper bound is that, unlike the “widening” in Section 6.2, this combined method will guarantee that the analysis terminates. This combination will be evaluated on the larger benchmarks in Chapter 7.

6.5 SPANNING TREE WIDENING

In the widening of Janssens, the infinite subdomain of type graphs is mapped onto a finite domain. The finite subdomain consists of type graphs with a depth bounded by a program dependent constant. Another way of viewing this is that the widening restricts the depth of a spanning tree of a grammar graph. It just so happens that when the grammar graph is a type graph then the spanning tree is unique.

By generalizing Janssens’ method a widening can be obtained that maps grammar graphs onto a finite domain. The finite domain in this case consists of grammar graphs such that each grammar graph have some spanning tree with a depth limited by a constant. This will ensure that there can only be finitely many such grammar graphs and consequently that the stationarity property (Definition 2.1.3) is established.

The property used to limit the depth can be the number of functor occurrences as in Janssens’ original method. Another possibility is to consider occurrences of (subsets of) principal labels so that the main focus becomes or-nodes instead of functor-nodes. The latter turns out to be more effective at keeping both grammar graph sizes and the number of iterations small.
Algorithm 6.5.1 (Spanning) $k$-restriction based on spanning tree of a grammar graph. Used to build an upper approximation of a set of nodes while at the same time ensuring that there exists a spanning tree of the constructed grammar graph such that no functor occurs more than $k$ times along any path.

spanning returns a pair $(m, M)$ of a node $m$ and a set $M$ representing already visited nodes. ANC is a set that represents the ancestors in the partially constructed spanning tree.

spanning$_d(n) = \text{spanning}(\{n\}, 0, \emptyset, \emptyset) \downarrow 1$

\[
\text{spanning}(S_0, \text{level}, ANC, M) = \\
\begin{align*}
1 & \text{ if any } \in S_0 \text{ then return any}, M \\
2 & S := \text{pnd}(S_0) \\
3 & \text{ if } \exists m (S, m) \in M \text{ then return } m, M \text{ — } m \text{ already part of spanning tree} \\
4 & \text{ if } \text{prlb}(S) = \{f\} \text{ then} \\
5 & \quad \text{ANC}_f := \{(S', l) \in ANC | \text{prlb}(S') = \{f\}\} \\
6 & \quad \text{ if } |\text{ANC}_f| \geq k \text{ then} \\
7 & \quad \quad (k+1)\text{th occurrence of } f \\
8 & \quad \quad \text{ if } \exists (S', l) \in \text{ANC}_f : S \subseteq S' \text{ then} \\
9 & \quad \quad \quad \text{there is an } m \text{ such that } (S', m) \in M \\
10 & \quad \quad \quad \quad \text{return } m, M \cup \{(S, m)\} \\
11 & \quad \quad \text{else} \\
12 & \quad \quad \quad \text{chose a } (S', l) \in \text{ANC}_f \\
13 & \quad \quad \quad \text{ restart the call to spanning at level } l \\
14 & \quad \quad \text{ with } S \cup S' \text{ as first argument} \\
15 & \text{ else} \\
16 & \quad \text{create a new functor-node } m \text{ with } \text{lb}(m) = f \\
17 & \quad \text{ABC}' := ABC \cup \{(S, \text{level})\} \\
18 & \quad \text{level}' := \text{level} + 1 \\
19 & \quad M_0 := M \cup \{(S, m)\} \\
20 & \quad \text{ for each } i \in [1, \text{arity}(f)] \text{ do} \\
21 & \quad \quad m \downarrow i, M_i := \text{spanning}(S \downarrow i, \text{level}', ABC', M_{i-1}) \\
22 & \quad \text{return } m, M_{\text{arity}(f)} \\
23 & \text{ else} \\
24 & \quad \text{create a new or-node } m \\
25 & \quad \text{level}' := \text{level} + 1 \\
26 & \quad M_0 := M \cup \{(S, m)\} \\
27 & \quad \text{ for each } f_i \in \text{prlb}(S) \text{ do} \\
28 & \quad \quad m \downarrow f_i, M_i := \text{spanning}(S_{f_i}, \text{level}', ABC, M_{i-1}) \\
29 & \quad \text{return } m, M_{\text{prlb}(S)} \\
30 & \text{ end if}
\]
6.5.1 Definition

The algorithm will be described as limiting the number of functor occurrences, in the next section the algorithm will be generalized to consider prilb-sets of or-nodes as well. The underlying idea in the algorithm is the same as used by Algorithm 5.3.2 to implement Janssens’ widening. That is, to construct, depth first, an upper bound of a set of grammar graph nodes while ensuring that the constructed type adheres to certain shape constraints.

Given a grammar graph $G$, with root node $n$, Algorithm 6.5.1 will construct a grammar graph $G'$, with root node $n'$, such that $n \subseteq n'$. Furthermore there exists a spanning tree of $G'$ such that no functor $f$ occurs more than $k$ times along a path in the spanning tree.

The algorithm constructs a depth-first spanning tree where the nodes in the tree are sets of nodes from the original grammar graph. The set $M$ represents marked nodes, that is, nodes that are already part of the spanning tree. The set $ANC$ represents the nodes on the path from the root to the present node, it is used to determine if descent is allowed or if the largest number of functors is about to be exceeded.

If descent is not allowed then there two cases,

1. (Line 9), there is an appropriate ancestor in the spanning tree that can safely approximate the node that was about to be added. In this case the implementation will select the deepest such ancestor. An alternative would be to consider not only ancestors but all visited nodes (on $M$), this alternative technique has not been investigated.

2. (Line 13), otherwise, select an ancestor to combine with this node. In this case the implementation will select the ancestor with as large intersection between $S$ and $S'$ as possible, breaking ties by preferring deep nodes. A variant for $k > 1$ would be to also consider merging ancestors on $ANC_f$.

The spanning tree is in general not unique. The algorithm may transform the grammar graph even though there exists another spanning tree that would not have exceeded the limit on functor occurrences. It seems likely that finding the best spanning tree would, in the worst case, have to investigate the exponentially many spanning trees.

Termination is ensured since: the recursion depth is limited by the size of the set $ANC$ which is bounded by the number of subsets of nodes from $G$; each recursive call can only be restarted finitely many times, each restart will involve a strictly larger set of nodes. That the result is a conservative
approximation can be shown in the same way as for Janssens' widening, Algorithm 5.3.2.

### 6.5.2 Variations of the Spanning Tree Widening

The original algorithm disallows descent in the spanning tree if there are already $k$ ancestors with the same function symbol as label. This can be generalized to principal labels and will then apply to or-nodes.

The generalized version would have lines 6–16 duplicated between line 24 and 25. The difference would be that for the new code the test $\text{prlb}(S') = \{f\}$ would become either $\text{prlb}(S') = \text{prlb}(S)$ or, to be even more aggressive, $\text{prlb}(S') \supseteq \text{prlb}(S)$. In addition, at line 29, the set $\text{ANC}$ has to include an entry corresponding to the or-node as well.

To avoid the issue of nested or-nodes the algorithm only combines nodes with the same label, in particular a functor-node is never replaced by an or-node.

If only or-nodes are considered then this operation no longer maps the incoming grammar graph onto a finite sub-domain. The reason is that there may be arbitrarily long paths with only functor-nodes and no or-nodes.

To show that the widening gives stationary sequences it would necessary to prove that the subdomain have the ascending chain property, that is, that any increasing sequence of such widened grammar graphs is stationary. This has not been attempted, instead the occurrences of functor nodes will be used as an additional constraints on the allowable paths to make the subdomain finite.

### 6.5.3 Performance

Several variants of Algorithm 6.5.1 with $k = 1$ will be evaluated in this section when inferring the recursive types of the synthetic benchmarks. The first variant considers only functors; the second one considers only or-nodes with equal $\text{prlb}$-sets; the third variant considers only or-nodes but also stops descent if there is an ancestor with a larger or equal $\text{prlb}$-sets; The last variant uses all these criteria at the same time, thus ensuring that the subdomain is finite.

The types are represented by minimized grammar graphs but size measures indicate the largest grammar graph that occur before minimization, typically this is after the widening or after the upper bound.

#### 6.5.3.1 Functors

The first variant is Algorithm 6.5.1 as it appears in the figure, that is, it only considers ancestors corresponding to functor nodes. This most closely corresponds to Janssens' original widening.
The result with clauses analyzed separately is shown in Table 6.3. The costs are reasonable except for FIXED scheduling where the number of iterations and widenings appears to grow exponentially.

The UNFAIR policies behaves much worse if they are not helped by the initial analysis pass caused by eager analysis. With the base case last the number of iterations and widenings becomes \( \Theta(n^2) \) and \( \Theta(n^3) \) respectively for LIFO. For FIXED both iterations and widenings still grow exponentially.

For all scheduling policies the size of the grammar graphs only grow linearly. This holds also if the clauses are rearranged to bring the base case last.

Recall from Section 5.3.3 that with Janssens’ widening the type graphs grow exponentially, even if only their minimized size is considered and in some cases even if only the number of non-equivalent nodes is considered.

If single clausal form is used then this variant of the spanning tree widening performs very badly (not shown). The number of iterations and widenings becomes quadratic and linear respectively but the size of the grammar graphs grows very quickly.

### 6.5.3.2 Principal Labels

The second variant allows an arbitrary number of functor-nodes on each path in the spanning tree but instead limits the number of or-nodes\(^{12}\) by not allowing two or-nodes on a path of the spanning tree to have the same set of principal labels. The results are shown in Table 6.4.

<table>
<thead>
<tr>
<th>expr</th>
<th>( n )</th>
<th>( \Omega(n) )</th>
<th>( \Theta(n^2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>133</td>
<td>5163</td>
<td>111</td>
<td>1790</td>
</tr>
<tr>
<td>222</td>
<td>(1000)</td>
<td>700</td>
<td></td>
</tr>
</tbody>
</table>

If the initial eager pass is disabled by rearranging clauses then the number of iterations and widenings for the UNFAIR policies increases by a factor \( \Theta(n) \) compared to Table 6.4.

\(^{12}\)Note that the grammar graphs are compact so or-nodes have at least two successors.
### Table 6.4: Spanning1 on synthetic benchmarks, only allows one or-node with a particular `prlb`-set on a path in the spanning tree.

<table>
<thead>
<tr>
<th>Iterations</th>
<th>( \nabla_{SP} )</th>
<th>max size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PAIR</td>
<td>FIXED</td>
</tr>
<tr>
<td>tree_2</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>tree_3</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>tree_4</td>
<td>13</td>
<td>18</td>
</tr>
<tr>
<td>tree_5</td>
<td>16</td>
<td>25</td>
</tr>
<tr>
<td>tree_6</td>
<td>19</td>
<td>33</td>
</tr>
<tr>
<td>tree_7</td>
<td>22</td>
<td>42</td>
</tr>
<tr>
<td>tree_8</td>
<td>25</td>
<td>52</td>
</tr>
<tr>
<td>tree_n</td>
<td>( \Theta(n) )</td>
<td>( \Theta(n^2) )</td>
</tr>
<tr>
<td>expr</td>
<td>46</td>
<td>136</td>
</tr>
</tbody>
</table>

### Table 6.5: Spanning1 on synthetic benchmarks, disallows descendant or-nodes with the same or smaller `prlb`-set on a path in the spanning tree. The result is the same if, in addition, only one occurrence of each function symbol is allowed on each path.

<table>
<thead>
<tr>
<th>Iterations</th>
<th>( \nabla_{SP} )</th>
<th>max size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PAIR</td>
<td>FIXED</td>
</tr>
<tr>
<td>tree_2</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>tree_3</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>tree_4</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>tree_5</td>
<td>13</td>
<td>16</td>
</tr>
<tr>
<td>tree_6</td>
<td>15</td>
<td>19</td>
</tr>
<tr>
<td>tree_7</td>
<td>17</td>
<td>22</td>
</tr>
<tr>
<td>tree_8</td>
<td>19</td>
<td>25</td>
</tr>
<tr>
<td>tree_n</td>
<td>( \Theta(n) )</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>expr</td>
<td>32–33</td>
<td>46</td>
</tr>
</tbody>
</table>
In the third variant descent to an or-node in the spanning tree is also prohibited when there are ancestors with larger \textit{prlb}-set. With this more aggressive variant all worklist scheduling policies behave similarly for the synthetic benchmarks and give a linear number of iterations and widenings, and linear size of encountered grammar graphs, see Table 6.5.

The fourth variant is the same as the third but also disallows multiple occurrences of the same functor along paths in the spanning tree. The results are the same as for the third variant.

Also for the third and fourth variant the UNFAIR policies becomes worse if the initial eager analysis pass is disabled. In this case the number of iterations becomes cubic and the number of widenings becomes quadratic in \( n \).

For variant two to four, that is when \textit{prlb}-sets are used to guide the widening, single clausal form gives good results as well (not shown). The number of iterations becomes linear, corresponding to a constant number (8) of analysis passes over all clauses. The largest size becomes linear as well and the number of widenings becomes constant (4).

In summary, restricting growth using subsets of \textit{prlb}-sets appears to give a robust and efficient method, especially if one of the FAIR scheduling policies is used.

\subsection{6.5.3.3 Increasing \textit{k}} It is necessary to increase \( k \) to enable the analyzer to infer nested types such as the nested lists manipulated by \texttt{list_of_lists}.

To get a rough indication of the cost of increasing \( k \) analysis of \texttt{tree} was performed for increasing values of \( k \) (\( k \in \{1, 2, 3, 5, 6\} \)). This simple experiment indicate that the cost of increasing \( k \) is reasonable.

With the default analysis where clauses are analyzed separately the result vary. For the FAIR policies the performance is good, the number of iterations and widenings and the size increase linearly with \( k \) for all variants of the widening.

With the UNFAIR widenings the performance is worse. For LIFO the number of iterations and widenings and the size increases faster than linearly for all variants. For FIXED the number of iterations and widenings and the size increases faster than linearly for the first variant (that limits the number of occurrences of each function symbol), the increase is linear when FIXED is used with any of the variants that use \textit{prlb}-sets.

If single clausal form is used to analyze \texttt{tree}, then the number of iterations, widenings and the size of the largest type increase linearly with \( k \) for all variants of the widening and, as usual, this is independent of scheduling policy.
6.5.4 Precision

For the synthetic precision benchmark, list_of_lists, this widening gives optimal result with \( k = 2 \) and the FAIR scheduling policies when any of the four variants of the widening is used. With the UNFAIR policies the result is comparable to that obtained with Janssens’ widening.

It is somewhat surprising that this widening can be strictly more precise than Janssens’ widening for the same \( k \).

For the spanning tree widening in this section the spanning tree chosen is rather arbitrary and can affect the precision. Algorithm 6.5.1 uses a left-to-right traversal of the successors of a node, that is, arcs with smaller labels will be followed first. In particular this means that lower numbered arguments of functor nodes will be descended first.

If instead a right-to-left traversal is used then the analysis result, for all scheduling policies, will be as imprecise as when using Janssens’ widening. The analysis result varies somewhat depending on if only function symbols are used to guide the widening or if instead, or in addition, prlb-sets are considered, but the inferred type will be roughly the same as with Janssens’ widening.

Other heuristics are possible for combining nodes along a path in the spanning tree. In particular, for \( k > 1 \), it might be advantageous to consider combining some of the \( k \) ancestors when occurrence \((k + 1)\) is encountered along a path of the spanning tree\(^{13}\). It would be interesting to see if some alternative heuristics could ensure consistently good results for the precision benchmark list_of_lists.

6.5.5 Relation to Gallagher & de Waal

Gallagher and de Waal [22] formulated a widening on regular unary logic programs\(^{14}\) that is similar to the widening presented in this section.

Reformulated on grammar graphs their restriction operator, what they call shortening, will disallow two distinct or-nodes\(^{15}\) with the same principal labels to be connected. It is thus similar to the variant in Section 6.5.2 above but considers all paths in the graph, not just the paths in a spanning tree\(^{16}\).

\(^{13}\)Similar heuristics could be used with Janssens’ widening.

\(^{14}\)See Section 3.3.4.

\(^{15}\)In a normal deterministic but non-compact grammar graph, that is, where or-nodes correspond to non-terminals and are allowed to have one or more successors.

\(^{16}\)It is possible that their method inspired the prlb-set variant of the spanning tree widening, I was aware of their work when the spanning tree widening was implemented.
The method maps the grammar domain onto a finite subdomain and since the subdomain is finite stationarity is ensured and the analysis should therefore terminate. Unfortunately, the method used to perform the mapping from the infinite grammar domain to the finite subdomain may not terminate.

The shortening is based on the following technique. If there is a path from an or-node \( n \) to an or-node \( n' \) such that \( \text{prlb}(n) = \text{prlb}(n') \) then \( n_\cup = n \cup n' \) is computed after which \( n \) is replaced with \( n_\cup \) everywhere in the graph. This is repeated until no such pairs of connected nodes exists.

Contrary to a claim in their paper this is not guaranteed to terminate. It is possible for two new nodes, \( m, m' \) say, to appear in the result of the upper bound \( n_\cup \), such that \( m \) and \( m' \) and their interrelation is structurally identical to the case with \( n \) and \( n' \). In this case the shortening could attempt to combine \( m \) and \( m' \) and would thus not terminate. Example 4 illustrates this using a grammar instead of a grammar graph.

**Example 4 (Shortening May Not Terminate)** Applying a shortening step may produce a new grammar where the same shortening step is still applicable.

\[
\begin{align*}
N & \rightarrow a \mid b(N') \mid c(T_1) \\
N' & \rightarrow a \mid b(a) \mid c(T_2) \\
T_1 & \rightarrow a \mid c(T_1) \\
T_2 & \rightarrow a \mid b(N')
\end{align*}
\]

There is a path from \( N \) to \( N' \) and they both have the same function symbols in their productions. A single step of the shortening would compute \( n_\cup = N \cup N' \)

\[
\begin{align*}
n_\cup & \rightarrow a \mid b(N') \mid c(T_{1\cup 2}) \\
T_{1\cup 2} & \rightarrow a \mid b(N') \mid c(T_1)
\end{align*}
\]

Clearly \( T_{1\cup 2} = T_1 \cup T_2 \) have exactly the same structure as \( N \).

Finally \( N \) would be replaced with \( n_\cup \) everywhere in the original grammar and unreachable productions removed, yielding,

\[
\begin{align*}
n_\cup & \rightarrow a \mid b(N') \mid c(T_{1\cup 2}) \\
T_{1\cup 2} & \rightarrow a \mid b(N') \mid c(T_1) \\
N' & \rightarrow a \mid b(a) \mid c(T_2) \\
T_1 & \rightarrow a \mid c(T_1) \\
T_2 & \rightarrow a \mid b(N')
\end{align*}
\]

The next shortening step could now select \( T_{1\cup 2} \) and \( N' \) leading to non-termination. Thus, in this example, applying a single shortening step to
the original grammar will produce a new grammar containing, in effect, a renamed version of the original grammar, thus enabling an infinite number of applications of the shortening step.

This problem is avoided when using Janssens’ technique, as in Algorithm 6.5.1 by ensuring that each newly created node have an identity taken from the finite set of subsets of nodes from the original grammar graph. The number of new nodes is thus bounded and termination is ensured. Another method would be to use a method that creates an upper bound in a more direct way while ensuring that the size of resulting grammar decreases. Such a method was proposed by Cousot and Cousot [12], where it was, incorrectly \(^{17}\), claimed that the upper bound also is a widening.

### 6.5.6 Summary and Conclusion

A class of widenings were described that operate directly on grammar graphs and therefore can avoid the size problems caused by the tree-like shape of type graphs.

The widenings use a restriction operator, inspired by Janssens’ widening, that constructs a non-unique depth first spanning tree of a grammar graph and restricts the length of the paths in this tree. Several variants of a widening using this restriction operator was implemented for (minimized) grammar graphs.

In the first variant the length of the paths are restricted by disallowing more than \(k\) nodes labeled with the same function symbol on a path. For the second variant, the length of the paths are instead restricted by disallowing more than \(k\) or-nodes with the same \(prlb\)-set on a path. The third variant, limits length of the paths by disallowing the case when an or-node have more than \(k\) ancestors with equal or larger \(prlb\)-set.

The variants where \(prlb\)-sets of or-nodes are used to guide the widening appears to be good at limiting both the number of iterations and the size of the grammar graphs. this is true even if the parameter \(k\) is increased to give more precise results.

With the default settings for the analyser the spanning tree widenings give the optimal type for the nested lists manipulated by \texttt{list_of_list}, this was not achievable with Janssens’ widening.

Analysis of the precision benchmark \texttt{list_of_list} exemplified that the analysis result can depend on implementation details such as the order in which successors of grammar graph nodes are traversed in the widening. High level

\(^{17}\)That their method is not guarantee stationarity was pointed out by Liu [37].
descriptions of widenings, or similar algorithms, that ignore these issues can therefore make it even harder to reproduce experimental results.

The widening in this section has similarities with a method proposed by Gallagher & de Waal [22] but it was demonstrated that their method may not always terminate.

6.6 TOPOLOGICAL CLASH REVISITED

The topological clash widening of Section 5.4 use the tree-shape of its argument type graphs to steer the construction of the resulting type. This means that it is not possible to avoid the sometimes huge differences in size between a type graph representation and a (minimized) grammar graph representation of the same type.

It may still be advantageous to use grammar graphs (minimized or not) as the main representation to speed up the other domain operations and only convert to the type graph representation as part of the topological clash widening.

As with Janssens’ widening it is also possible to use the less precise \( \subseteq \)-minimization upper bound (Section 6.2). In particular it might be possible to avoid some of the problems with the large type graph representation by using this upper bound to create the type graphs necessary as input to the topological clash widening.

In the next section minimized grammar graphs will be used with a novel implementation technique that constructs the result of the topological clash widening in a direct manner as opposed to the transformation based method originally suggested by Henntenryck et al. and implemented in Section 5.4.

6.6.1 Direct Construction of Topological Clash Widening

Recall the definition of the topological clash widening in Section 5.4. It takes as argument an “old” and a “new” type graph and creates an upper bound of these that must satisfy certain structural constraints relative the “old” type graph. The old type graph corresponds to an approximation of a call or success pattern that should be updated to incorporate also the information represented by the new type graph.

The method suggested by Henntenryck et al. [30] was to start with an upper bound \( T' \) of the old \( T \) and new type graphs and then transform \( T' \) until it satisfies the structural constraints vis-a-vis the old type graph \( T \). This approach was described in Section 5.4.

A critical component of the transformation to obtain a suitable type graph is a way to construct the upper bound of two type graph nodes that is not
too large. The method used in Section 5.4 constructs an ordinary upper bound and then attempts to shrink it until it is small enough. The method has two problems: the upper bound tend to be huge and the shrinking sometimes fails to decrease the size in which case an any-node must be used with a resulting loss of precision.

We have already seen two widenings that use a quite different method to obtain a suitable type. The original type graph widening by Janssens (Section 5.3) and the grammar graph spanning tree widening (Section 6.5).

The main component of both these widenings is an algorithm that constructs an upper bound of a set of grammar graph nodes while ensuring that the resulting type has a suitable shape. When, during depth first construction of the upper bound, the constructed type is about to violate the shape constraints the algorithm resolves the problem by introducing an arc to an already constructed and possibly less precise node or by restarting the construction of the upper bound at some level closer to the root using a larger set of nodes as input.

Procedure tc below (Algorithm 6.6.1) implements the same idea for constructing the result of the topological clash widening. The required shape now depends on the shape of the “old” type graph $T$ instead of the path properties considered by Janssens’ widening and the spanning tree widening. The shape constraints for the topological clash widening is that there must be no disallowed topological clashes between the old type $T$ and the constructed type graph (Definition 5.4.6).

The arguments to procedure tc are almost the same as those to the spanning tree widening (Algorithm 6.5.1):

1. $S_0$, the set of nodes for which an upper bound should be created.
2. level, the level (or equivalently depth) of the node that is about to be created. This argument is redundant, it is used as a convenient way to identify this call to tc so that recursive calls to tc can restart this call (with a larger set $S_0$ of nodes).
3. ANC, a set of entries representing each of the (proper) ancestors to the node that is about to be created. Each entry is a triple with the ancestor node $m$, the set of nodes $S'$ for which $m$ is an upper bound, and the level of $m$ ($l$) so that the call to tc can be identified and restarted with some nodes added to $S'$.
4. $n$, the corresponding node from the “old” type graph. If $n'$ is the node about to be constructed then the pair $(n,n')$ corresponds to the pairs of nodes in $C$ (Definition 5.4.1, page 133).
Algorithm 6.6.1 (Topological Clash by Direct Construction)

\[\text{tc}(S_0, \text{level}, \text{ANC}, n) = \]
1. if any \(\in S_0\) then return any
2. \(S := \text{prlb}(S_0)\)
3. \(\text{ANC}_S := \{(S', m, l) \in \text{ANC} \mid \text{prlb}(S) \subseteq \text{prlb}(S'), \ l \leq \text{depth}(n)\}\)
4. if \(\exists m, l : (S, m, l) \in \text{ANC}_S\) then return \(m\) — i.e., a backward arc to \(m\)
5. if \(\text{subset heuristic} = \text{false by default}\)
6. if \(\text{prlb}(S) = \{f\}\) then
7. else
8. if \(\exists (S', l, m) \in \text{ANC}_S, S \subseteq S', \text{prlb}(S') = \{f\}\) then return \(m\)
9. if \(\exists (S', l, m) \in \text{ANC}_S, S \subseteq S'\) then return \(m\)
10. \(\text{ANC}_b := \text{safe_backward}(S, \text{ANC}, \text{level}, \text{depth}(n))\)
11. \(\text{ANC}_r := \text{safe_restart}(S, \text{ANC}, \text{level}, \text{depth}(n))\)
12. if \(\text{depth}(n) < \text{level}\) then — depth-clash, cannot go to next level
13. if \(\text{ANC}_b \neq \emptyset\) then
14. select an \((S', m, l) \in \text{ANC}_b\)
15. return \(m\) — corresponds to arc replacement
16. else — give up. This case has not occurred in practice
17. — Avoid any-node as successor of or-node
18. if parent \((S', m, \text{level} - 1) \in \text{ANC}\) exists and \(|\text{prlb}(S')| > 1\) then
19. return any from level \(\text{level} - 1\)
20. else
21. return any
22. — not a depth-clash, invariant: \(\text{ANC}_b \neq \emptyset \lor \text{ANC}_r \neq \emptyset\)
23. if \(\text{prlb}(n) \subseteq \text{prlb}(S)\) then — functor clash
24. if \(|\text{ANC}_S| > 0\) then
25. if \(\exists (S', m, l) \in \text{ANC}_b\) then
26. return \(m\) — corresponds to arc replacement
27. else — corresponds to node replacement
28. select an \((S', m, l) \in \text{ANC}_r\)
29. restart \(\text{tc}\) at level \(l\) with \(S \cup S'\) as first argument
30. else — no suitable ancestor, grow
31. return \(\text{upp_set}(S, \{(S', m) \mid \exists : (S', l, m) \in \text{ANC}\})\)
32. else — \(\text{prlb}(n) = \text{prlb}(S)\)
33. if \(\text{prlb}(S) = \{f\}\) then
34. create a new functor-node \(m\) with \(\text{lb}(m) = f\)
35. \(\text{ANC}' := \text{ANC} \cup \{(S, m, \text{level})\}\)
36. \(\text{level}' := \text{level} + 1\)
37. for each \(i \in [1, \text{arity}(f)]\) do
38. \(m \downarrow i := \text{tc}(S \downarrow i, \text{level}', \text{ANC}', n \downarrow i)\)
39. return \(m\)
40. else
41. create a new or-node \(m\)
42. \(\text{ANC}' := \text{ANC} \cup \{(S, m, \text{level})\}\)
43. \(\text{level}' := \text{level} + 1\)
44. for each \(f \in \text{prlb}(S)\) do
45. \(m \downarrow f := \text{tc}(S \downarrow f, \text{level}', \text{ANC}', n \downarrow f)\)
46. return \(m\)
Recall that a topological clash between a node $n$ from the old type graph and a node $n'$ in the new type graph is when the two nodes have either different depth or different set of principal labels (Definition 5.4.2). The node $n'$ from the new type graph corresponds to the node $m$ being returned from $tc$.

The following cases apply:

1. The new node $m$ is an any-node. This is always allowed and is detected at line 1 of Algorithm 6.6.1.

2. The new node $m$ is reached through a backward arc and is not deeper than the old node $n$. This is handled at line 4.

   In lines 6–19, if the boolean flag $\text{subset}_\text{heuristic}$ is true then ancestors corresponding to a larger set of nodes is also considered for introducing a backward arc. The effect is similar to the $\subseteq$-minimization in Algorithm 6.2.1. The special case when the set of principal labels is a singleton ensures that (what would become) a functor node is never replaced by an or-node. The implementation will use the deepest suitable ancestor node.

3. The new node $m$ is deeper than $n$. This corresponds to a depth clash (Definition 5.4.6). This case is detected at line 15. In this case a new node is not allowed to be created at the current level. There are two cases:

   (a) If there is some ancestor node that is also an upper bound of $S$ then it can be used provided it is close enough to the root so that it would not be deeper than the old node. This is handled at lines 16–18. This corresponds to the arc replacement used in Section 5.4.2.

   (b) A backward arc cannot be used. This corresponds to the node replacement used in Section 5.4.2.

   An ancestor is selected that can be made an upper bound of both its original set of nodes $S'$ and also the set of nodes of the current level $S$. If such an ancestor $m$ exists then, at lines 19–21, the call to create it is restarted with the union of the two sets of nodes. It is also possible that no such ancestor exists. Then a number of heuristics could be attempted. However, this case has never happened in practice so a simple solution is to simply insert an any-node. This case is handled at lines 24–27 that also takes special care if the parent is an or-node (since or-nodes are not allowed to have any-nodes as successors).
4. The new node has the same depth as the old node but strictly larger set of principal labels (the new node will therefore be an or-node). This is potentially a disallowed functor clash:

   (a) There is an ancestor with at least as large set of principal labels. In this case the new node is not allowed to be used but instead a backward arc or restart (corresponding to arc replacement or node replacement) must be used, lines 32-36.

   (b) There is no ancestor with as large set of principal labels. In this case the new node is allowed and there are no further constraints on the shape of the type graph below it. The call to \textit{upp\_set} at line 38 creates a properly shaped type graph as an upper bound of a set of grammar graph nodes. The boolean flag \textit{subset\_heuristic} is used also in \textit{upp\_seq} (Algorithm 6.6.4) to obtain a smaller type graph.

5. The final possibility is that the old and the (about to be constructed) new node have the same depth (\textit{level}) and the same set of principal labels, that is, they are either both functor nodes or both or-nodes. This is handled as an ordinary upper bound in lines 40-53.

A number of supporting procedures are needed for Algorithm 6.6.1. Procedure \textit{safe\_backward} (Algorithm 6.6.2) extracts the entries on ANC that can be used to introduce a backward arc corresponding to arc replacement. Procedure \textit{safe\_restart} (Algorithm 6.6.3) extracts the entries on ANC that can be restarted with a larger set of nodes, corresponding to node replacement. Finally procedure \textit{upp\_set} (Algorithm 6.6.4) that implements an upper bound for sets of nodes.

The implementation for both \textit{safe\_backward} and \textit{safe\_restart} attempts to select an ancestor entry that is as similar as possible to the current set \( S \) of nodes.

The ancestors that can safely be used instead of creating a new node for the upper bound of the nodes on \( S \) must satisfy:

1. Must be no deeper than \( n \). This is so that using this ancestor does not constitute a depth clash (Definition 5.4.6).

2. Must be a safe approximation. \( S \subseteq S' \) is a sufficient condition to guarantee this, it would also be possible to compare the nodes using \( \sqsubseteq \).

3. Avoid nested or-nodes in the case the replacement is an or-node.
When multiple suitable ancestor entries exists then the implementation will choose the entry with the fewest elements not in $S$, that is, $|S' \setminus S|$ as small as possible, breaking ties by preferring deep entries (with large level).

**Algorithm 6.6.2** safe\_backward($S$, $ANC$, $l$, $l_{\text{max}}$) =

$ANC_0 := \{(S', m, l') \in ANC \mid l' \leq l_{\text{max}}\}$ — close enough to the root

if $\exists (S', m, l - 1) \in ANC : |\text{prlb}(S')| > 1$ then

— parent is an or-node, successor must be a functor-node

$ANC_f := \{(S', m, l') \in ANC_0 \mid \text{prlb}(S') = \text{prlb}(S)\}$

$ANC_1 := \{(S', m, l') \in ANC_f \mid S \subseteq S'\}$ — Safe approximation

else

— parent is a functor-node, successor can be anything

$ANC_1 := \{(S', m, l') \in ANC_0 \mid S \subseteq S'\}$ — Safe approximation

return $ANC_1$

The ancestors that can safely be restarted to provide an upper bound of the union of their original set of nodes ($S'$) and $S$ must satisfy:

1. Must be no deeper than $n$ so it would not constitute a depth clash.
2. Must restart with strictly larger set of nodes to ensure termination.
3. Avoid nested or-nodes in the case that a restart would transform a functor-node to an or-node.

When multiple suitable ancestor entries exist then the implementation will choose the entry with most element in common with $S$, that is, $|S \cap S'|$ as large as possible, breaking ties by preferring deep entries.

**Algorithm 6.6.3**

safe\_restart($S$, $ANC$, $l_{\text{max}}$)

$ANC_0 := \{(S', m, l') \in ANC \mid l' \leq l_{\text{max}}\}$ — close enough to the root

$ANC_1 := \{(S', m, l') \in ANC_0 \mid S \subseteq S'\}$ — restart with more nodes

$ANC_{or} := \{\text{parent is an or-node}\}$

$\{(S', m, l') \in ANC_1 \mid \exists (S'', m', l' - 1) \in ANC : |\text{prlb}(S'')| > 1\}$

$ANC_{fun} := ANC_1 \setminus ANC_{or}$ — parent is a functor-node (if it exists)

— avoid nested or-nodes, do not increase prlb if parent is an or-node

$ANC_2 := \{(S', m, l') \in ANC_{or} \mid \text{prlb}(S) \subseteq \text{prlb}(S')\}$

return $ANC_2 \cup ANC_{fun}$

□
The final procedure used by tc is \texttt{upp\_set} that creates a type graph that is an upper bound of a set of grammar graph nodes, it is a straightforward generalization of the upper bound for type graphs (Algorithm 3.3.7). The boolean flag \texttt{subset\_heuristic} if true implements a simple case of \texttt{$\subseteq$}-minimization, see the description of Algorithm 6.6.1.

\textbf{Algorithm 6.6.4 (Type Graph Set Upper Bound)}

\begin{verbatim}
upp_set(S_0, ANC) =
    if any \in S_0 then return any
    S := prnd(S_0)
    if \exists m : (S, m) \in ANC then return m — i.e., a backward arc to m
    if subset\_heuristic then — false by default
        if prlb(S) = {f} then
            if \exists (S', m) \in ANC, S \subseteq S', prlb(S') = {f} then return m
            else
                if \exists (S', m) \in ANC, S \subseteq S' then return m
        if prlb(S) = {f} then
            create a new functor\_node m with lb(m) = f
            ANC' := ANC \cup \{(S, m)\}
            for each i \in [1, arity(f)] do
                m \downarrow i := upp_set(S \downarrow i, ANC')
            return m
        else
            create a new or\_node m
            ANC' := ANC \cup \{(S, m)\}
            for each f \in prlb(S) do
                m \downarrow f := upp_set(S_f, ANC')
            return m
\end{verbatim}

The direct construction of the topological clash widening creates an upper bound of a set of nodes. As an optimization it is therefore possible to avoid explicitly creating an upper bound of the two argument types\textsuperscript{18}. Furthermore, the argument types need not be type graphs but can be arbitrary\textsuperscript{19} grammar graphs as long as the “old” type is converted to a type graph when used as last argument to tc.

\textsuperscript{18}This was done for Janssens’ widening in Section 5.3.3.3. It require a slight modification of the analyzer.

\textsuperscript{19}As always the grammar graphs are compact and thus normal, deterministic and reduced.
Definition 6.6.5 (Direct Construction Topological Clash Widening)

Topological Clash widening by direct construction. $T_{\text{old}}$ and $T_{\text{new}}$ may be grammar graphs and need not be type graphs. The use of construct (Algorithm 3.3.6) ensures that the last arg to tc is a proper type graph.

$$T_{\text{old}} \vee T_{\text{new}} = \begin{cases} T_{\text{new}} & \text{if } T_{\text{old}} = \bot, \\ T_{\text{old}} & \text{if } T_{\text{new}} \subseteq T_{\text{old}}, \\ \text{tc}(\{T_{\text{old}}, T_{\text{new}}\}, 0, \emptyset, \text{construct}(T_{\text{old}})) & \text{otherwise}. \end{cases}$$

\[\square\]

6.6.2 Discussion

The direct construction method was implemented with minimized grammar graphs as type domain. Unfortunately this does not avoid the need to temporarily create a type graph representation\(^{20}\).

The Topological Clash widening obtained using the direct construction method shares many of the characteristics with the original transformation based formulation in Section 5.4. In particular, due to the need to create a type graph representation temporarily, also the direct construction method has severe performance problems when inferring recursive types such as those in the synthetic benchmark \texttt{tree}.n.

With the default analysis method where each clause is analyzed separately using a FAIR worklist scheduling policy the largest type graph\(^{21}\) appears to grow exponentially with \(n\), the number of function symbols in the recursive type. This is true even if with flag \texttt{subset-heuristic} enabled.

If, however, single-clausal form is used when analysing the programs then the direct construction method performs as well as the transformation based method in terms of the size of the largest encountered type graph size.

The main advantage with the direct construction method is that it does not use the troublesome node replacement procedure (Algorithm 5.4.7). This avoids the performance problems with the sometimes huge initial upper bound used in the replace node procedure. In practice it also avoids the possibility that an any-node need to be used to replace a node in the constructed type graph.

\(^{20}\)The last argument to tc in Definition 6.6.5.

\(^{21}\)The type graph construct-ed as the last argument to tc in Definition 6.6.5.
The topological clash widening using direct construction with minimized grammar graphs as type domain is evaluated on the larger benchmarks in Section 7.4.2. It is the most precise of the proper widenings in this thesis, only the $\sqsubseteq$-minimization "widening" is more precise. In particular it is more precise than the transformation based topological clash widening. This can be explained by the fact that the direct construction methods never resorts to using any-nodes to implement node replacement.

It would be interesting to replace the heuristic $\text{subset heuristic}$ with a technique based on comparing the nodes in the two sets with $\sqsubseteq$ which should result in smaller types than when the simple subset test is used.

Another possibility would be to use $\sqsubseteq$-minimization when converting the "old" grammar graph $T_{old}$ to a type graph\footnote{Care must be taken since stationarity of the widening require that the "old" type is strictly more precise than the constructed type, see Appendix B.}. It would also be possible to relax the requirement that $\text{upps\textsubscript{set}}$ (Algorithm 6.6.4) constructs a proper type graph, it could instead construct a, probably much smaller, grammar graph.
To give some indication of the strengths and weaknesses of the widenings the larger benchmarks were used. The benchmarks are described in Section 4.4.

Two representations of term grammars are used as type-domains: type graphs and minimized grammar graphs.

The default scheduling policy is fair FIFO scheduling of the worklist and clauses analyzed separately. For the topological clash widening other variants are used as well since this proved beneficial in Section 5.4.

The size (the number of nodes) of the largest encountered type is shown in column “max size”. Especially for the type graph domain the size of the intermediate types causes problem. The size of the largest type in the result is also shown (“max result”). These are often much smaller than the types that occur during analysis.

For type graphs the types are kept as small as possible by applying minimization after all operations that create a type graph (upper bound, widening, and intersection). The reason this is done is to give an estimate of the best case performance in terms of the size of the intermediate type graphs. For some widenings the intermediate type graphs are nevertheless impractically large.

Once a method is chosen the question of whether minimization need to be done must be studied separately. It is likely that even if minimization of the type graphs should prove beneficial some heuristic could be used that avoids excessive use of minimization. The same holds for the minimized grammar graphs, it may be advantageous to relax the requirement that the grammar graphs always minimized.

The number of analyzed clause bodies (column “Iter”) and the number of call and success pattern widenings (“\(\nabla_{CP}\)” and “\(\nabla_{SP}\)” respectively) are presented to give an estimate of the domain independent work of the analyzer.
Precision is summarized as the number of non-variable arguments ("Non-var"), detailed precision is presented in Appendix A. Column "Unkn" shows the percentage of unknown arguments, that is, argument position for which the type analysis did not obtain any information and for which the separate analysis that detects uninitialized variables also could not infer any information. The intent is that column "Unkn" should give some indication of the absolute (im-)precision obtained.

For completeness the total time of the analysis (Column "Time") is also included. The time does not include garbage collection but does measure the substantial time used to collect statistics and for code relating to debugging. As was discussed in Section 4.3.1 the time is probably not a good indication of the absolute or relative efficiency of the analyses.

The next section presents the result obtained for the simple-minded functors widening. It also contains measurements of the artificially high precision obtained for the Aquarius benchmarks, as discussed in Section 4.4.

The subsequent sections present results for the more precise widenings. The precision obtained using the various widenings is summarized in Section 7.7, the detailed precision information appear in Appendix A.

7.1 FUNCTORS WIDENING

The simple functors widening, Section 5.1, is used to provide a baseline for the precision that can be obtained. Even when this widening infer the same number of non-variable arguments the types are much less precise.

7.1.1 Comparison with Type Graph domain

If the functors widening is used together with the type graph domain instead of minimized grammar graphs then the size of the largest intermediate type increase substantially. The sizes of the intermediate type graphs and the difference compared to using minimized grammar graphs are shown in Table 7.2. The sizes are larger than for many of the more precise type graph domains, illustrating that a more precise domain may be more efficient in terms of the cost of representing the types.

7.1.2 Impact of Input-Data on Precision

All the Aquarius benchmarks, except chat_big, contain the input data as part of the program, thus making it available to the analyzer with a resulting increase in the obtained precision. As was argued in Section 4.4.2 this is not representative of how real programs, or even the Aquarius benchmarks, would be used. To estimate the impact of this the percentage of non-variable arguments detected for the original benchmarks was compared with the
The results are shown in Table 7.3. Measurements only include predicates common to all three variants, that is, predicates providing input-data or performing input validation are not measured. As can be seen the precision will decrease significantly when the explicit in-data is removed. As could be expected some of this lost precision is regained when the analyzer can get information from analyzing the code that performs validation.

For the original definition of nand the precision, even using this naive widening, is surprisingly good. However, as can be seen from Table 7.3 this precision is to a large extent artificial.

For chat.parser analysis of the original version detects that, with the particular inputs available as part of the program, one predicate will never be called at all. Analysis of any of the variants with unknown input will infer that particular predicate may be reached. This shows that the inputs that are part of chat.parser does not even exercise the whole program. For the more precise widenings additional predicates will be classified as unreachable.
7.1. Functors Widening

<table>
<thead>
<tr>
<th>Program</th>
<th>max size</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>meta-qsort</td>
<td>429</td>
<td>11</td>
</tr>
<tr>
<td>prover</td>
<td>508</td>
<td>9</td>
</tr>
<tr>
<td>browse</td>
<td>91</td>
<td>5</td>
</tr>
<tr>
<td>unify</td>
<td>360</td>
<td>10</td>
</tr>
<tr>
<td>flatten</td>
<td>82</td>
<td>4</td>
</tr>
<tr>
<td>sdda</td>
<td>115</td>
<td>3</td>
</tr>
<tr>
<td>reducer</td>
<td>2213</td>
<td>50</td>
</tr>
<tr>
<td>boyer</td>
<td>14434</td>
<td>118</td>
</tr>
<tr>
<td>simple-analyzer</td>
<td>556</td>
<td>9</td>
</tr>
<tr>
<td>nand</td>
<td>376</td>
<td>8</td>
</tr>
<tr>
<td>chat-parser</td>
<td>149713</td>
<td>569</td>
</tr>
<tr>
<td>chat-big</td>
<td>122290</td>
<td>172</td>
</tr>
<tr>
<td>aquarius_compiler</td>
<td>3172</td>
<td>18</td>
</tr>
<tr>
<td>kish-andi</td>
<td>181545</td>
<td>500</td>
</tr>
<tr>
<td>symbolic1</td>
<td>224096</td>
<td>357</td>
</tr>
<tr>
<td>tricia</td>
<td>17405</td>
<td>185</td>
</tr>
<tr>
<td>reader</td>
<td>5790</td>
<td>67</td>
</tr>
<tr>
<td>bamspec</td>
<td>10544</td>
<td>87</td>
</tr>
</tbody>
</table>

Table 7.2: Functors widening using type graph domain. Size of largest intermediate type graph shown. Minimization done after upper bound and intersection.

<table>
<thead>
<tr>
<th>Program</th>
<th>Nonvariable Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Orig</td>
</tr>
<tr>
<td>meta-qsort</td>
<td>10%</td>
</tr>
<tr>
<td>prover</td>
<td>47%</td>
</tr>
<tr>
<td>browse</td>
<td>57%</td>
</tr>
<tr>
<td>unify</td>
<td>38%</td>
</tr>
<tr>
<td>flatten</td>
<td>33%</td>
</tr>
<tr>
<td>sdda</td>
<td>37%</td>
</tr>
<tr>
<td>reducer</td>
<td>24%</td>
</tr>
<tr>
<td>boyer</td>
<td>10%</td>
</tr>
<tr>
<td><strong>simple-analyzer</strong></td>
<td>41%</td>
</tr>
<tr>
<td>nand</td>
<td>70%</td>
</tr>
<tr>
<td>chat-parser</td>
<td>36%</td>
</tr>
<tr>
<td>kish-andi</td>
<td>32%</td>
</tr>
</tbody>
</table>

Table 7.3: Impact on precision of explicit input in program text. Original Aquarius Benchmarks compared to versions that read input and read+validate input. No entry in column Validate means that it seems unreasonable to validate the input data and that the value in column Read should be used. The unintuitive result for **simple-analyzer** is explained in the text.
The table also illustrates that the absolute precision obtained on benchmarks with input data available can make a widening appear much better than what could more realistically be expected.

The unmodified benchmarks are nevertheless used for all measurements in this thesis. The reason is that the main focus is on comparing the precision obtained for various widenings and not on estimating the absolute precision.

7.1.3 An Unintuitive Result

For the benchmark simple_analyzer the precision is higher when the input is unknown (column Read of Table 7.3). This counterintuitive result is explained by the fact that the functors widening is not monotone\(^1\). While a monotone function \( f \) would have \( T \subseteq T' \implies f(T) \subseteq f(T') \) this is not true for the functors widening (nor for the other grammar widenings considered in this thesis). This means that the restriction of a more precise input \( T \) is not guaranteed to be more precise than the restriction of a less precise type \( T' \).

Concretely what happens in simple_analyzer is the following. There is a doubly recursive predicate \( p \)\(^2\) with a structure like:

\[
p(\emptyset, \ldots ).
p([X | Xs], \ldots ) :-
X = s(Y, Xs'),
\ldots ,
p(Xs', \ldots ),
p(Xs, \ldots ).
\]

The predicate \( p \) is originally reached via a call from another predicate \( q \). The argument when called from \( q \) is a list of binary structures \( s(\ldots ) \) where the second argument of \( s \) is also such a list. That is, the call pattern for the call from \( q \) is \( p(T) \) where \( T \) is,

\[
T \rightarrow (S, T) \mid []
S \rightarrow s(E, T)
\]

The difference between the original version with known input and the one with unknown input is that for the original version the analyzer will infer that \( E \), the first argument of \( s(\ldots ) \), will be a list cell with unknown argu-

\(^1\)In particular the restriction function functors, Algorithm 5.1.2, is nonmonotone.
\(^2\)Actually traverse_trees/5.
ments, that is $E \rightarrow (\text{any,any})$. So the grammar for the call from $q$ will be:

$$
T \rightarrow (S,T) \mid [] \\
S \rightarrow s(E,T) \\
E \rightarrow (\text{any,any})
$$

For the version with unknown input the analyzer will not be able to infer any information about the first argument of $s(\ldots)$, that is, with unknown input to simple_analyzer the call from $q$ will be described by:

$$
T' \rightarrow (S',T') \mid [] \\
S' \rightarrow s(E',T') \\
E' \rightarrow \text{any}
$$

$T$ is strictly more precise than $T'$, as would be expected. However, consider now what happens when the functors-widening is applied to $T$ and $T'$. The widened result of $T'$ ($T'_v$) will be:

$$
T'_v \rightarrow (T'_v,T'_v) \mid s(\text{any},T'_v) \mid []
$$

For the more precise $T$ the list constructor in $E$ will “taint” the list constructor corresponding to the top-level list, the widened result becomes:

$$
T_v \rightarrow (\text{any,any}) \mid s(T_v,T_v) \mid []
$$

Finally when the analyzer reaches the last recursive call in $p$ it will call $p$ with the type corresponding to the tail of the argument list. For the original version of the benchmark, where more precision was available for the call to $p$, this will be the tail of the list constructor in $T_v$, that is, $\text{any}$. This will destroy all type-information and the final call pattern for $p$ will be $\text{any}$.

For the modified version of simple_analyzer, where the information about the call to $p$ was less precise, the tail of the argument list of $p$ is the tail of the list constructor in $T'_v$, that is, $T'_v$ itself. A fixpoint have been reached and the final call pattern $T'_v$ is much more precise even though the original call from $q$ was strictly less precise.

### 7.2 TYPE-JUNGLE WIDENING

Table 7.4 shows the result when using the type jungle widening (Section 5.2) with minimized grammar graphs, FIFO worklist scheduling and each clause analyzed separately. This analysis is identified with “Jungle” in the detailed precision tables in Appendix A.
<table>
<thead>
<tr>
<th>Program</th>
<th>Iter</th>
<th>$\nabla_{CP}$</th>
<th>$\nabla_{SP}$</th>
<th>max size</th>
<th>max result</th>
<th>Nonvar</th>
<th>Unkn</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt</td>
<td>51</td>
<td>12</td>
<td>15</td>
<td>30</td>
<td>5</td>
<td>72%</td>
<td>28%</td>
<td>2s</td>
</tr>
<tr>
<td>meta_sort</td>
<td>149</td>
<td>31</td>
<td>35</td>
<td>45</td>
<td>16</td>
<td>10%</td>
<td>60%</td>
<td>3s</td>
</tr>
<tr>
<td>prover</td>
<td>179</td>
<td>44</td>
<td>69</td>
<td>62</td>
<td>22</td>
<td>41%</td>
<td>32%</td>
<td>7s</td>
</tr>
<tr>
<td>browse</td>
<td>57</td>
<td>19</td>
<td>31</td>
<td>79</td>
<td>9</td>
<td>57%</td>
<td>19%</td>
<td>4s</td>
</tr>
<tr>
<td>unify</td>
<td>159</td>
<td>58</td>
<td>72</td>
<td>29</td>
<td>11</td>
<td>38%</td>
<td>44%</td>
<td>5s</td>
</tr>
<tr>
<td>flatten</td>
<td>138</td>
<td>45</td>
<td>70</td>
<td>16</td>
<td>12</td>
<td>33%</td>
<td>41%</td>
<td>3s</td>
</tr>
<tr>
<td>sddda</td>
<td>145</td>
<td>44</td>
<td>61</td>
<td>26</td>
<td>16</td>
<td>35%</td>
<td>45%</td>
<td>3s</td>
</tr>
<tr>
<td>reduce</td>
<td>156</td>
<td>37</td>
<td>94</td>
<td>43</td>
<td>33</td>
<td>24%</td>
<td>58%</td>
<td>4s</td>
</tr>
<tr>
<td>boyer</td>
<td>177</td>
<td>36</td>
<td>137</td>
<td>122</td>
<td>116</td>
<td>10%</td>
<td>62%</td>
<td>13s</td>
</tr>
<tr>
<td>simple analyzer</td>
<td>318</td>
<td>98</td>
<td>169</td>
<td>89</td>
<td>16</td>
<td>43%</td>
<td>33%</td>
<td>20s</td>
</tr>
<tr>
<td>nand</td>
<td>660</td>
<td>118</td>
<td>159</td>
<td>47</td>
<td>17</td>
<td>69%</td>
<td>5%</td>
<td>23s</td>
</tr>
<tr>
<td>chat parser</td>
<td>5302</td>
<td>1437</td>
<td>1236</td>
<td>329</td>
<td>191</td>
<td>40%</td>
<td>16%</td>
<td>596s</td>
</tr>
<tr>
<td>chat Jig</td>
<td>11233</td>
<td>1467</td>
<td>3482</td>
<td>719</td>
<td>635</td>
<td>23%</td>
<td>44%</td>
<td>2000s</td>
</tr>
<tr>
<td>aquarius compiler</td>
<td>11356</td>
<td>2535</td>
<td>4373</td>
<td>180</td>
<td>66</td>
<td>24%</td>
<td>50%</td>
<td>407s</td>
</tr>
<tr>
<td>kish andi</td>
<td>1861</td>
<td>471</td>
<td>734</td>
<td>356</td>
<td>93</td>
<td>39%</td>
<td>39%</td>
<td>165s</td>
</tr>
<tr>
<td>symbolic1</td>
<td>2238</td>
<td>103</td>
<td>711</td>
<td>620</td>
<td>596</td>
<td>54%</td>
<td>27%</td>
<td>326s</td>
</tr>
<tr>
<td>tricia</td>
<td>1495</td>
<td>399</td>
<td>808</td>
<td>121</td>
<td>35</td>
<td>20%</td>
<td>60%</td>
<td>52s</td>
</tr>
<tr>
<td>reader</td>
<td>641</td>
<td>112</td>
<td>166</td>
<td>70</td>
<td>27</td>
<td>42%</td>
<td>11%</td>
<td>16s</td>
</tr>
<tr>
<td>bamspec</td>
<td>1143</td>
<td>289</td>
<td>481</td>
<td>157</td>
<td>115</td>
<td>37%</td>
<td>31%</td>
<td>49s</td>
</tr>
</tbody>
</table>

*Some debugging code was disabled so the time is unfairly low compared to the other benchmarks.

Table 7.4: Type-Jungle widening using minimized grammar graphs.
Clauses analyzed separately, FIFO scheduling policy.
It should be noted that the type-jungle widening can be implemented more efficiently than what was used for Table 7.4, in particular the widening can be done as part of the upper bound operation\(^3\) which probably would reduce the size of the already reasonably sized grammar graphs.

The precision of the type-jungle widening compares favorably with using Janssens’ widening with \(k = 1\) but without any of the performance problems.

The only widening that systematically gives lower precision is the functors widening of the previous section.

### 7.3 JANSSENS’ WIDENING

Several variations were used to evaluate Janssens’ type graph widening (Section 5.3). All measurements are done with clauses analyzed separately using the fair FIFO scheduling policy.

The widening is optimized by first checking for the common case that no forward path has a function symbols that occurs more than \(k\) times. If no such path exists then the widening would be the identity function and need not be called. This optimization therefore does not affect the result, only the time.

To give an estimate of the best case, in terms of the sizes of the intermediate type graphs, minimization was performed after all operations (intersection, upper bound, and widening). This will keep down the sizes at the type graphs at the expense of time.

Janssens’ variant of type graphs allows or-nodes to be successors of or-nodes, that is, non-normal (but still deterministic) type graphs.

For this reason such nested or-nodes are allowed by the analyzer when using Janssens’ widening (but not for any other widening). A consequence of this is that the grammar graph version of the upper bound operation could not be used, instead upper bound for type graphs was computed using the same method as was used by Janssens’, that is, by using (the full version of) Algorithm 5.3.2 without a limit on the number of occurrences of function symbols.

The upper bound operation attempts to avoid creating a new result by first checking if either argument is less precise than the other. There are also checks for some other, less common, special cases that will avoid the full upper bound operation. Only if these special cases does not apply is the full version based on Algorithm 5.3.2 needed. The upper bound is therefore

\(^3\)See Section 5.3.3.3 where this was done for Janssens’ widening.
<table>
<thead>
<tr>
<th>Program</th>
<th>Iter</th>
<th>$\nabla CP$</th>
<th>$\nabla SP$</th>
<th>max size</th>
<th>max result</th>
<th>Nonvar</th>
<th>Unkn</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt</td>
<td>51</td>
<td>12</td>
<td>15</td>
<td>12</td>
<td>4</td>
<td>72%</td>
<td>28%</td>
<td>2s</td>
</tr>
<tr>
<td>meta reports</td>
<td>185</td>
<td>37</td>
<td>39</td>
<td>104</td>
<td>58</td>
<td>10%</td>
<td>60%</td>
<td>4s</td>
</tr>
<tr>
<td>prover</td>
<td>219</td>
<td>59</td>
<td>90</td>
<td>494</td>
<td>44</td>
<td>41%</td>
<td>32%</td>
<td>20s</td>
</tr>
<tr>
<td>browse</td>
<td>57</td>
<td>19</td>
<td>31</td>
<td>91</td>
<td>9</td>
<td>57%</td>
<td>19%</td>
<td>2s</td>
</tr>
<tr>
<td>unify</td>
<td>159</td>
<td>58</td>
<td>72</td>
<td>20</td>
<td>5</td>
<td>38%</td>
<td>44%</td>
<td>5s</td>
</tr>
<tr>
<td>flatten</td>
<td>138</td>
<td>45</td>
<td>70</td>
<td>21</td>
<td>19</td>
<td>33%</td>
<td>41%</td>
<td>3s</td>
</tr>
<tr>
<td>sdda</td>
<td>146</td>
<td>44</td>
<td>63</td>
<td>22</td>
<td>13</td>
<td>35%</td>
<td>45%</td>
<td>3s</td>
</tr>
<tr>
<td>reducer</td>
<td>157</td>
<td>38</td>
<td>95</td>
<td>69</td>
<td>54</td>
<td>24%</td>
<td>58%</td>
<td>5s</td>
</tr>
<tr>
<td>prover</td>
<td>177</td>
<td>36</td>
<td>137</td>
<td>310</td>
<td>235</td>
<td>10%</td>
<td>62%</td>
<td>17s</td>
</tr>
<tr>
<td>simple analyzer</td>
<td>322</td>
<td>99</td>
<td>170</td>
<td>226</td>
<td>30</td>
<td>43%</td>
<td>33%</td>
<td>14s</td>
</tr>
<tr>
<td>nand</td>
<td>660</td>
<td>118</td>
<td>159</td>
<td>275</td>
<td>39</td>
<td>69%</td>
<td>5%</td>
<td>34s</td>
</tr>
<tr>
<td>chat big</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>27022</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>aquarium compiler</td>
<td>11615</td>
<td>2387</td>
<td>4452</td>
<td>11119</td>
<td>1471</td>
<td>24%</td>
<td>50%</td>
<td>484s</td>
</tr>
<tr>
<td>kish andi</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>618219</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>symbolic1</td>
<td>2238</td>
<td>103</td>
<td>711</td>
<td>2759</td>
<td>486</td>
<td>54%</td>
<td>27%</td>
<td>33s</td>
</tr>
<tr>
<td>tricia</td>
<td>1513</td>
<td>403</td>
<td>819</td>
<td>518</td>
<td>173</td>
<td>20%</td>
<td>60%</td>
<td>58s</td>
</tr>
<tr>
<td>reader</td>
<td>641</td>
<td>112</td>
<td>168</td>
<td>354</td>
<td>63</td>
<td>42%</td>
<td>11%</td>
<td>23s</td>
</tr>
<tr>
<td>bamsp ec</td>
<td>1144</td>
<td>289</td>
<td>479</td>
<td>1063</td>
<td>1062</td>
<td>37%</td>
<td>31%</td>
<td>53s</td>
</tr>
</tbody>
</table>

Table 7.5: Janssens' widening, $k = 1$ using type graphs. Clauses analyzed separately, FIFO scheduling policy, minimization performed after intersect, upper bound and after widening.

more optimized than the version used by Janssens' where a new type graph is always created for the upper bound.

The disadvantage compared to the evaluation of the other widenings on type graphs is that the result from upper bound before minimization typically becomes larger.

The first two variants of Janssens' widenings use $k = 1$ (Table 7.5) and $k = 2$ (Table 7.6). In the detailed precision evaluation in Appendix A these analyses are denoted "J1" and "J2" respectively.

For some benchmarks the analyses was aborted by the time-limit imposed on all domain operations\(^4\). In these cases the size of the largest type graph before the interruption is shown.

Recall from Section 5.3.3 that using Janssens' widening to infer the type of a recursively defined data structure will produce excessively large type graphs during analysis.

\(^4\)The limit is two CPU minutes.
Several of the benchmarks manipulate such data structures. However, in many cases the information about the recursive data structure is lost due to lack of precision and the analysis proceeds without difficulty.

For the benchmarks where the information about recursive data structures is not lost due to imprecision Janssens' widening produces impractically large type graphs. This is particularly apparent for benchmark *kish andi*, where the problems stem from the analysis of a predicate for simplifying arithmetic expressions, a predicate with a structure similar to the synthetic benchmark *expr*.

A precision comparison between all widenings is presented in Section 7.7. In summary there are other methods that are more precise than Janssens' widening and also methods that are as precise but more efficient.

<table>
<thead>
<tr>
<th>Program</th>
<th>Iter</th>
<th>$\vee_{CP}$</th>
<th>$\vee_{SP}$</th>
<th>max size</th>
<th>max result</th>
<th>Nonvar</th>
<th>Unkn</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt</td>
<td>61</td>
<td>14</td>
<td>17</td>
<td>13</td>
<td>6</td>
<td>72%</td>
<td>2%</td>
<td>2s</td>
</tr>
<tr>
<td>meta-qsort</td>
<td>266</td>
<td>48</td>
<td>45</td>
<td>121</td>
<td>96</td>
<td>10%</td>
<td>60%</td>
<td>7s</td>
</tr>
<tr>
<td>prover</td>
<td>322</td>
<td>82</td>
<td>131</td>
<td>11679</td>
<td>47</td>
<td>41%</td>
<td>32%</td>
<td>90s</td>
</tr>
<tr>
<td>browse</td>
<td>79</td>
<td>25</td>
<td>41</td>
<td>91</td>
<td>22</td>
<td>57%</td>
<td>19%</td>
<td>3s</td>
</tr>
<tr>
<td>unify</td>
<td>166</td>
<td>58</td>
<td>88</td>
<td>67</td>
<td>34</td>
<td>38%</td>
<td>44%</td>
<td>6s</td>
</tr>
<tr>
<td>flatten</td>
<td>167</td>
<td>51</td>
<td>86</td>
<td>31</td>
<td>19</td>
<td>33%</td>
<td>41%</td>
<td>4s</td>
</tr>
<tr>
<td>sdda</td>
<td>177</td>
<td>54</td>
<td>75</td>
<td>31</td>
<td>16</td>
<td>35%</td>
<td>45%</td>
<td>5s</td>
</tr>
<tr>
<td>reducer</td>
<td>186</td>
<td>47</td>
<td>112</td>
<td>69</td>
<td>69</td>
<td>31%</td>
<td>52%</td>
<td>6s</td>
</tr>
<tr>
<td>boyer</td>
<td>185</td>
<td>41</td>
<td>141</td>
<td>291</td>
<td>291</td>
<td>10%</td>
<td>62%</td>
<td>19s</td>
</tr>
<tr>
<td>simple-analyzer</td>
<td>428</td>
<td>135</td>
<td>224</td>
<td>547</td>
<td>45</td>
<td>43%</td>
<td>33%</td>
<td>24s</td>
</tr>
<tr>
<td>nand</td>
<td>743</td>
<td>145</td>
<td>202</td>
<td>1229</td>
<td>47</td>
<td>69%</td>
<td>5%</td>
<td>48s</td>
</tr>
<tr>
<td>chat-parser</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>149600</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>chat-jig</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>667420</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>aquarius-compiler</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>230000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>kish-angi</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>658210</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>symbolic-1</td>
<td>2476</td>
<td>113</td>
<td>724</td>
<td>2571</td>
<td>486</td>
<td>54%</td>
<td>27%</td>
<td>380s</td>
</tr>
<tr>
<td>tricia</td>
<td>1796</td>
<td>461</td>
<td>981</td>
<td>7550</td>
<td>228</td>
<td>23%</td>
<td>38%</td>
<td>98s</td>
</tr>
<tr>
<td>reader</td>
<td>693</td>
<td>127</td>
<td>211</td>
<td>155</td>
<td>122</td>
<td>42%</td>
<td>11%</td>
<td>29s</td>
</tr>
<tr>
<td>bamspiec</td>
<td>1192</td>
<td>301</td>
<td>503</td>
<td>1069</td>
<td>1062</td>
<td>37%</td>
<td>31%</td>
<td>57s</td>
</tr>
</tbody>
</table>

Table 7.6: Janssens' widening, $k = 2$ using type graphs. Clauses analyzed separately, FIFO scheduling policy, minimization performed after intersect, upper bound and after widening.
### 7.3.1 Janssens’ widening with $\sqsubseteq$-Minimization

The last two variants of using Janssens’ widening is performed as in the previous section but use an upper bound followed by $\sqsubseteq$-minimization to perform all upper bound operations, Algorithm 6.2.3 Section 6.4.2. The results for $k = 1$ and $k = 2$ are shown in Table 7.7 and Table 7.8 respectively. In Appendix A these analyses are denoted “J1L” for $k = 1$ and “J2L” for $k = 2$.

The reason it might be advantageous to use $\sqsubseteq$-minimization is threefold: Firstly, by first transforming the type graphs to minimized grammar graphs and then performing the upper bound on the resulting grammar graphs it avoids creating large (non-minimal) type graphs as a result of upper bound. Second, when transforming the resulting upper bound into a proper type graph a backward arc is introduced not only if there is an equivalent ancestor, but also if there is a less precise ancestor, resulting in a smaller type graph. Finally, $\sqsubseteq$-minimization efficiently infers the type of recursively defined data structures, something that is prohibitively expensive using the unmodified version of Janssens’ widening.

<table>
<thead>
<tr>
<th>Program</th>
<th>Iter</th>
<th>$\nabla CP$</th>
<th>$\nabla SP$</th>
<th>max size</th>
<th>max result</th>
<th>Nonvar</th>
<th>Unkn</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt</td>
<td>51</td>
<td>12</td>
<td>15</td>
<td>12</td>
<td>4</td>
<td>72%</td>
<td>28%</td>
<td>2s</td>
</tr>
<tr>
<td>meta_report</td>
<td>168</td>
<td>34</td>
<td>38</td>
<td>117</td>
<td>58</td>
<td>10%</td>
<td>60%</td>
<td>5s</td>
</tr>
<tr>
<td>prover</td>
<td>215</td>
<td>56</td>
<td>70</td>
<td>281</td>
<td>39</td>
<td>41%</td>
<td>32%</td>
<td>17s</td>
</tr>
<tr>
<td>browse</td>
<td>57</td>
<td>19</td>
<td>31</td>
<td>91</td>
<td>9</td>
<td>57%</td>
<td>19%</td>
<td>2s</td>
</tr>
<tr>
<td>unify</td>
<td>130</td>
<td>58</td>
<td>72</td>
<td>24</td>
<td>5</td>
<td>38%</td>
<td>44%</td>
<td>5s</td>
</tr>
<tr>
<td>flatten</td>
<td>140</td>
<td>47</td>
<td>72</td>
<td>20</td>
<td>15</td>
<td>33%</td>
<td>41%</td>
<td>3s</td>
</tr>
<tr>
<td>sdda</td>
<td>146</td>
<td>44</td>
<td>63</td>
<td>17</td>
<td>13</td>
<td>35%</td>
<td>45%</td>
<td>4s</td>
</tr>
<tr>
<td>reducer</td>
<td>157</td>
<td>38</td>
<td>95</td>
<td>60</td>
<td>54</td>
<td>24%</td>
<td>58%</td>
<td>5s</td>
</tr>
<tr>
<td>boyer</td>
<td>177</td>
<td>36</td>
<td>137</td>
<td>1784</td>
<td>832</td>
<td>10%</td>
<td>62%</td>
<td>29s</td>
</tr>
<tr>
<td>simple_analyzer</td>
<td>322</td>
<td>99</td>
<td>170</td>
<td>226</td>
<td>20</td>
<td>43%</td>
<td>33%</td>
<td>14s</td>
</tr>
<tr>
<td>nand</td>
<td>752</td>
<td>153</td>
<td>199</td>
<td>409</td>
<td>14</td>
<td>51%</td>
<td>22%</td>
<td>41s</td>
</tr>
<tr>
<td>chat_parser</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chat_big</td>
<td>11333</td>
<td>1939</td>
<td>3649</td>
<td>10919</td>
<td>5463</td>
<td>44%</td>
<td>44%</td>
<td>11698s</td>
</tr>
<tr>
<td>aquarius_compiler</td>
<td>11358</td>
<td>2575</td>
<td>4373</td>
<td>679</td>
<td>203</td>
<td>24%</td>
<td>50%</td>
<td>428s</td>
</tr>
<tr>
<td>kish_andi</td>
<td>1862</td>
<td>479</td>
<td>730</td>
<td>3900</td>
<td>98</td>
<td>39%</td>
<td>38%</td>
<td>165s</td>
</tr>
<tr>
<td>symbolic1</td>
<td>2238</td>
<td>103</td>
<td>711</td>
<td>2080</td>
<td>486</td>
<td>54%</td>
<td>27%</td>
<td>453s</td>
</tr>
<tr>
<td>tricia</td>
<td>1489</td>
<td>403</td>
<td>807</td>
<td>173</td>
<td>173</td>
<td>20%</td>
<td>60%</td>
<td>57s</td>
</tr>
<tr>
<td>reader</td>
<td>643</td>
<td>112</td>
<td>169</td>
<td>164</td>
<td>75</td>
<td>42%</td>
<td>11%</td>
<td>26s</td>
</tr>
<tr>
<td>bam_spec</td>
<td>1150</td>
<td>292</td>
<td>483</td>
<td>833</td>
<td>832</td>
<td>37%</td>
<td>31%</td>
<td>69s</td>
</tr>
</tbody>
</table>

Table 7.7: Janssens’ widening, $k = 1$ using type graphs. Upper bound performed using $\sqsubseteq$-minimization, clauses analyzed separately, FIFO scheduling policy, minimization performed after intersect and after widening.
In Section 7.6 \( \subseteq \)-minimization is used as the sole widening\(^5\) which gives better precision than any other widening. As discussed in Section 6.2.3 using \( \subseteq \)-minimization in combination with Janssens’ widening can give more precise results than when using only Janssens’ widening. For the larger benchmarks the precision turns out to be the same except for benchmark \texttt{nand} where the use of \( \subseteq \)-minimization gives much lower precision.

The results from using Janssens’ widening with \( \subseteq \)-minimization are interesting. Not only is the precision as good\(^6\), the intermediate type graphs are much smaller and only the two chat-parser benchmarks cause serious problems. Also interesting is that the largest type graphs are sometimes smaller with \( k = 2 \) than with \( k = 1 \), presumably because Janssens’ widening interferes less with the work of the \( \subseteq \)-minimization when \( k \) is larger.

\(^5\)As discussed in Section 6.2.1 it is an open question whether \( \subseteq \)-minimization guarantees termination of the analysis. The combination with Janssens’ widening can therefore be regarded as a modification of the \( \subseteq \)-minimization “widening” to guarantee termination.

\(^6\)Except for \texttt{nand}.

<table>
<thead>
<tr>
<th>Program</th>
<th>Iter</th>
<th>( \bigtriangleup_{CP} )</th>
<th>( \bigtriangleup_{SP} )</th>
<th>max</th>
<th>max</th>
<th>Nonvar</th>
<th>Unkn</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>size</td>
<td>result</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crypt</td>
<td>55</td>
<td>14</td>
<td>14</td>
<td>12</td>
<td>6</td>
<td>72%</td>
<td>28%</td>
<td>2s</td>
</tr>
<tr>
<td>meta_report</td>
<td>256</td>
<td>48</td>
<td>39</td>
<td>152</td>
<td>45</td>
<td>10%</td>
<td>60%</td>
<td>7s</td>
</tr>
<tr>
<td>prover</td>
<td>200</td>
<td>52</td>
<td>68</td>
<td>110</td>
<td>47</td>
<td>41%</td>
<td>32%</td>
<td>15s</td>
</tr>
<tr>
<td>browse</td>
<td>73</td>
<td>24</td>
<td>35</td>
<td>91</td>
<td>22</td>
<td>57%</td>
<td>19%</td>
<td>3s</td>
</tr>
<tr>
<td>unify</td>
<td>160</td>
<td>58</td>
<td>83</td>
<td>67</td>
<td>34</td>
<td>38%</td>
<td>44%</td>
<td>6s</td>
</tr>
<tr>
<td>flatten</td>
<td>140</td>
<td>47</td>
<td>72</td>
<td>20</td>
<td>15</td>
<td>33%</td>
<td>41%</td>
<td>3s</td>
</tr>
<tr>
<td>sdda</td>
<td>165</td>
<td>52</td>
<td>71</td>
<td>18</td>
<td>16</td>
<td>35%</td>
<td>45%</td>
<td>5s</td>
</tr>
<tr>
<td>reducer</td>
<td>185</td>
<td>50</td>
<td>112</td>
<td>69</td>
<td>69</td>
<td>31%</td>
<td>52%</td>
<td>6s</td>
</tr>
<tr>
<td>boyer</td>
<td>177</td>
<td>36</td>
<td>137</td>
<td>513</td>
<td>513</td>
<td>10%</td>
<td>62%</td>
<td>24s</td>
</tr>
<tr>
<td>simple_analyzer</td>
<td>356</td>
<td>123</td>
<td>184</td>
<td>226</td>
<td>43</td>
<td>43%</td>
<td>33%</td>
<td>15s</td>
</tr>
<tr>
<td>nand</td>
<td>853</td>
<td>185</td>
<td>231</td>
<td>409</td>
<td>28</td>
<td>51%</td>
<td>22%</td>
<td>54s</td>
</tr>
<tr>
<td>chat_parser</td>
<td>-</td>
<td>-</td>
<td>28485</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>( \infty )</td>
</tr>
<tr>
<td>chat_big</td>
<td>-</td>
<td>-</td>
<td>29324</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>( \infty )</td>
</tr>
<tr>
<td>aquarius_compiler</td>
<td>11495</td>
<td>2634</td>
<td>4461</td>
<td>693</td>
<td>203</td>
<td>24%</td>
<td>49%</td>
<td>433s</td>
</tr>
<tr>
<td>kishandi</td>
<td>2633</td>
<td>767</td>
<td>1019</td>
<td>1572</td>
<td>184</td>
<td>40%</td>
<td>37%</td>
<td>293s</td>
</tr>
<tr>
<td>symbolic1</td>
<td>2468</td>
<td>111</td>
<td>714</td>
<td>1912</td>
<td>486</td>
<td>54%</td>
<td>27%</td>
<td>441s</td>
</tr>
<tr>
<td>tricia</td>
<td>1564</td>
<td>425</td>
<td>834</td>
<td>228</td>
<td>228</td>
<td>23%</td>
<td>58%</td>
<td>60s</td>
</tr>
<tr>
<td>reader</td>
<td>710</td>
<td>133</td>
<td>202</td>
<td>149</td>
<td>75</td>
<td>42%</td>
<td>11%</td>
<td>32s</td>
</tr>
<tr>
<td>lam.spec</td>
<td>1154</td>
<td>294</td>
<td>485</td>
<td>833</td>
<td>832</td>
<td>37%</td>
<td>31%</td>
<td>67s</td>
</tr>
</tbody>
</table>

Table 7.8: Janssens’ widening, \( k = 2 \) using type graphs. Upper bound performed using \( \subseteq \)-minimization, Clauses analyzed separately, FIFO scheduling policy, minimization performed after intersect, and after widening.
7.4 TOPOLOGICAL CLASH WIDENING

This section presents results for several variants of the topological clash widening described in Section 5.4.

Recall from Section 5.4.2 that the result from applying the topological clash widening is a type graph that obeys certain shape constraints in relation to a previously obtained type graph.

The original formulation, also followed in Section 5.4.2, obtains a suitably shaped type graph by starting with an upper bound of the arguments of the widening and then repeatedly transforming the type graph until the shape requirements are satisfied. To this end the main heuristic is the method used to replace an internal node \( n \) in a type graph with an upper bound of \( n \) and one of its descendants while ensuring that the resulting type graph has strictly fewer nodes. The method used in this thesis creates a sometimes huge upper bound that is then shrunk. If a small enough type graph cannot be obtained through shrinkage then, as an imprecise fallback, an any-node is used. The results of using the original formulation of the topological clash widening are presented in the next section.

An alternative and novel formulation of the topological clash widening, described in Section 6.6.1, constructs a suitable type graph directly using a similar method as that used by Janssens (but with a completely different shape requirement). The advantage with this method is both efficiency, since no (huge) upper bounds of internal nodes need to be created, and precision, since the widening never needs to fall back to using an any-node to ensure termination. Further efficiency is obtained by avoiding the creation of the initial upper bound. The results of using the direct construction method is presented in Section 7.4.2.

7.4.1 Original Topological Clash Widening

This section presents the results with topological clash widening as originally formulated, using repeated transformations to obtain a resulting type graph of suitable shape.

As discussed above a crucial part of the widening is the method that can compute the upper bound of two internal nodes of a type graph such that the number of nodes in the upper bound is below a certain limit.

No such method was suggested in the articles by Hentenryck et al. The method used in the evaluation in this thesis first creates the upper bound and then try to shrink the resulting type graph using various heuristics as detailed in Section 5.4.3.2.

This heuristic has two main shortcomings: Firstly the upper bound of the two internal nodes may, as is always the case with type graphs, become very
large before the shrinking heuristics can be applied. Secondly, it may prove impossible to shrink the resulting upper bound sufficiently in which case an any-node has to be used to obtain a very imprecise upper bound but with fewer nodes.

The necessity to sometimes introduce any-nodes means that there is no graceful degradation of the obtained precision. Even though this widening in principle is more precise than any of the other widenings this is not the case in practice. This is at least partly due to the precision loss caused by the any-nodes. As an example the precision is sometimes worse than the theoretically much less precise type-jungle and functors widenings.

A further consequence of the brittleness is that the obtained precision varies non-systematically with factors such as the scheduling policy used.

To give an estimate of the best case, in terms of the sizes of the intermediate type graphs, minimization was performed after intersection and widening. This is similar to how Janssens’ widening was evaluated.

---

<table>
<thead>
<tr>
<th>Program</th>
<th>Iter</th>
<th>$\nabla_{CP}$</th>
<th>$\nabla_{SP}$</th>
<th>max size</th>
<th>max result</th>
<th>Nonvar</th>
<th>Unkn</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt</td>
<td>71</td>
<td>19</td>
<td>16</td>
<td>21</td>
<td>17</td>
<td>72%</td>
<td>28%</td>
<td>3s</td>
</tr>
<tr>
<td>meta_qsort</td>
<td>202</td>
<td>40</td>
<td>41</td>
<td>617</td>
<td>140</td>
<td>10%</td>
<td>60%</td>
<td>65s</td>
</tr>
<tr>
<td>prover</td>
<td>117</td>
<td>28</td>
<td>50</td>
<td>161</td>
<td>26</td>
<td>23%</td>
<td>50%</td>
<td>4s</td>
</tr>
<tr>
<td>browse</td>
<td>110</td>
<td>34</td>
<td>36</td>
<td>143</td>
<td>143</td>
<td>57%</td>
<td>19%</td>
<td>49s</td>
</tr>
<tr>
<td>unify</td>
<td>160</td>
<td>58</td>
<td>84</td>
<td>64</td>
<td>31</td>
<td>38%</td>
<td>44%</td>
<td>6s</td>
</tr>
<tr>
<td>flatten</td>
<td>138</td>
<td>45</td>
<td>70</td>
<td>21</td>
<td>19</td>
<td>33%</td>
<td>41%</td>
<td>3s</td>
</tr>
<tr>
<td>sdda</td>
<td>198</td>
<td>61</td>
<td>81</td>
<td>48</td>
<td>30</td>
<td>36%</td>
<td>44%</td>
<td>7s</td>
</tr>
<tr>
<td>reducer</td>
<td>197</td>
<td>52</td>
<td>114</td>
<td>69</td>
<td>69</td>
<td>40%</td>
<td>42%</td>
<td>8s</td>
</tr>
<tr>
<td>boyer</td>
<td>177</td>
<td>36</td>
<td>137</td>
<td>252</td>
<td>252</td>
<td>10%</td>
<td>62%</td>
<td>12s</td>
</tr>
<tr>
<td>simple_analyzer</td>
<td>352</td>
<td>123</td>
<td>180</td>
<td>423</td>
<td>226</td>
<td>41%</td>
<td>35%</td>
<td>64s</td>
</tr>
<tr>
<td>nand</td>
<td>715</td>
<td>136</td>
<td>185</td>
<td>612</td>
<td>127</td>
<td>69%</td>
<td>5%</td>
<td>48s</td>
</tr>
<tr>
<td>chat_parser</td>
<td>7727</td>
<td>2074</td>
<td>1756</td>
<td>25827</td>
<td>9507</td>
<td>40%</td>
<td>16%</td>
<td>4241s</td>
</tr>
<tr>
<td>chat_big</td>
<td>11680</td>
<td>2003</td>
<td>3744</td>
<td>198941</td>
<td>11328</td>
<td>23%</td>
<td>44%</td>
<td>5487s</td>
</tr>
<tr>
<td>aquarius_compiler</td>
<td>11776</td>
<td>2698</td>
<td>4585</td>
<td>7471</td>
<td>185</td>
<td>24%</td>
<td>49%</td>
<td>511s</td>
</tr>
<tr>
<td>kish_andi</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3364011</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>symbolic1</td>
<td>6042</td>
<td>344</td>
<td>862</td>
<td>5927</td>
<td>1326</td>
<td>54%</td>
<td>27%</td>
<td>1653s</td>
</tr>
<tr>
<td>tricia</td>
<td>1601</td>
<td>428</td>
<td>855</td>
<td>291</td>
<td>228</td>
<td>23%</td>
<td>58%</td>
<td>66s</td>
</tr>
<tr>
<td>reader</td>
<td>632</td>
<td>127</td>
<td>183</td>
<td>122</td>
<td>122</td>
<td>42%</td>
<td>11%</td>
<td>30s</td>
</tr>
<tr>
<td>bamspec</td>
<td>1153</td>
<td>294</td>
<td>482</td>
<td>1063</td>
<td>1062</td>
<td>37%</td>
<td>31%</td>
<td>55s</td>
</tr>
</tbody>
</table>

**Table 7.9: Topological Clash widening using type graphs.** Clauses analyzed separately, FIFO scheduling policy. Minimization done after upper bound, intersection and widening. Upper bound uses minimized grammar graphs to avoid huge intermediate type graphs.
Upper bound was implemented by first comparing the arguments and directly returning the less precise if either is less precise than the other. In the remaining cases both type graphs are converted to minimized grammar graphs and upper bound is performed to obtain a minimized grammar graph. The minimized grammar graph can then be converted directly to a minimized type graph. The advantage is that no nonminimal type graph needs to be constructed. Note that the use of minimized grammar graphs was not possible in the evaluation of Janssens’ widening since that widening uses a slightly relaxed definition of type graphs.

Minimized grammar graphs were not used for the upper bounds performed on internal nodes of a type graph as part of the widening. These upper bounds therefore tend to create substantially larger type graphs. Since these upper bounds are internal to the widening and an artifact of the chosen heuristic their size is not included in the tables in the column for largest encountered type graph.

7.4.1.1 Clauses Analyzed Separately - Fair Scheduling

The first variant uses FIFO scheduling and clauses analyzed separately. This variant is denoted “TC” in the detailed precision tables of Appendix A. The results are shown in Table 7.9.

When used for inferring the type of recursive data structures this variant of the topological clash widening leads to excessively sized type graphs. This was systematically studied in Section 5.4.4. The same effect can be seen in Table 7.9, especially for benchmark kish_andi.

7.4.1.2 Clauses Analyzed Separately - Unfair Scheduling

In Section 5.4.4.1 it was observed that using the unfair worklist scheduling policy LIFO could lead to significantly smaller intermediate type graphs, at the expense of more iterations and widenings. There the scheduling policy was given more control over when clauses are analyzed by moving the non-recursive clause last in the synthetic benchmark tree, effectively disabling the initial eager analysis of the call from the entry point.

A more systematic method is to disable all eager analysis of calls, see ai_cali Section 4.1.3.2. This gives the scheduling policy total control over when clauses are analyzed.

Table 7.10 shows the result of using LIFO scheduling and non-eager analysis of calls. In Appendix A this analysis is referenced to as “CT”\(^7\).

The number of analyzed clauses (column Iter) and the number of widenings are significantly larger than for the FIFO case in Table 7.9.

\(^7\)For “TC” in the other direction …
The troublesome benchmark kishandi can now be analyzed and the largest type graph encountered is much smaller than for the FIFO case. For the other benchmarks the result is not so clear.

The reason analysis of benchmark symbolic1 is aborted can be found in the upper bound used as part of node replacement as discussed earlier. During analysis of symbolic1 type graphs larger than twelve million nodes are created in the widening operation. Eventually this causes a call to the widening to exceed the time limit and analysis is aborted.

### 7.4.1.3 Clauses Analyzed Together

The topological clash widening is more likely to introduce recursiveness in the type if the nodes close to the root have large sets of principal labels. One way to obtain this effect is to postpone widening until the result of all clauses of a predicate have been combined. As discussed in Section 4.2.1 this is achieved in the analyzer by transforming each predicate into single clausal form. Analysis of a predicate in single clausal form will obtain the result from the single clause,
<table>
<thead>
<tr>
<th>Program</th>
<th>Iter</th>
<th>$\nabla_{CP}$</th>
<th>$\nabla_{SP}$</th>
<th>max size</th>
<th>max result</th>
<th>Nonvar</th>
<th>Unkn</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt</td>
<td>28</td>
<td>19</td>
<td>15</td>
<td>21</td>
<td>17</td>
<td>72%</td>
<td>28%</td>
<td>3s</td>
</tr>
<tr>
<td>meta_qsort</td>
<td>53</td>
<td>46</td>
<td>25</td>
<td>497</td>
<td>140</td>
<td>10%</td>
<td>60%</td>
<td>93s</td>
</tr>
<tr>
<td>prove</td>
<td>57</td>
<td>46</td>
<td>38</td>
<td>190</td>
<td>47</td>
<td>41%</td>
<td>32%</td>
<td>17s</td>
</tr>
<tr>
<td>browse</td>
<td>62</td>
<td>36</td>
<td>32</td>
<td>144</td>
<td>139</td>
<td>57%</td>
<td>19%</td>
<td>54s</td>
</tr>
<tr>
<td>unify</td>
<td>86</td>
<td>61</td>
<td>59</td>
<td>61</td>
<td>22</td>
<td>38%</td>
<td>44%</td>
<td>10s</td>
</tr>
<tr>
<td>flatten</td>
<td>107</td>
<td>51</td>
<td>60</td>
<td>21</td>
<td>19</td>
<td>33%</td>
<td>41%</td>
<td>6s</td>
</tr>
<tr>
<td>sdda</td>
<td>92</td>
<td>58</td>
<td>50</td>
<td>48</td>
<td>30</td>
<td>36%</td>
<td>44%</td>
<td>9s</td>
</tr>
<tr>
<td>reducer</td>
<td>128</td>
<td>58</td>
<td>70</td>
<td>69</td>
<td>69</td>
<td>40%</td>
<td>42%</td>
<td>17s</td>
</tr>
<tr>
<td>boyer</td>
<td>45</td>
<td>36</td>
<td>31</td>
<td>301</td>
<td>295</td>
<td>10%</td>
<td>62%</td>
<td>30s</td>
</tr>
<tr>
<td>simple analyzer</td>
<td>249</td>
<td>129</td>
<td>155</td>
<td>423</td>
<td>226</td>
<td>41%</td>
<td>35%</td>
<td>91s</td>
</tr>
<tr>
<td>nand</td>
<td>179</td>
<td>141</td>
<td>119</td>
<td>663</td>
<td>635</td>
<td>69%</td>
<td>5%</td>
<td>103s</td>
</tr>
<tr>
<td>chat_parser</td>
<td>2771</td>
<td>2636</td>
<td>1835</td>
<td>11420</td>
<td>5605</td>
<td>40%</td>
<td>16%</td>
<td>4245s</td>
</tr>
</tbody>
</table>

Table 7.11: Topological Clash widening using type graphs. Predicates in single clausal form. FIFO scheduling policy. Minimization done after upper bound, intersection and widening. Upper bound uses minimized grammar graphs to avoid huge intermediate type graphs.

For predicates with a simple structure, such as the synthetic benchmarks `tree`, `expr`, this lead to a tremendous efficiency improvement, as was shown in Section 5.4.4.2.

All the benchmarks can now be analyzed. The recursive data structure manipulated in `kishandi` is no longer a problem. Both versions of the `chat parser` still cause very large type graphs to appear.

Table 7.11 shows the performance when predicates are analyzed in single clausal form. In Appendix A this analysis is denoted “TCSC”. Note that each analyzed clause now corresponds to analysis of all the original clauses so the value in column “Iter” is not comparable to the analyses where single clausal form is not used.
Table 7.12: Topological Clash widening using direct construction with subset heuristic as described in the text. Minimized grammar graph domain. Clauses analyzed separately, FIFO scheduling policy.

<table>
<thead>
<tr>
<th>Program</th>
<th>Iter</th>
<th>$\Delta CP$</th>
<th>$\Delta SP$</th>
<th>max result</th>
<th>Nonvar</th>
<th>Unkn</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt</td>
<td>71</td>
<td>19</td>
<td>13</td>
<td>45</td>
<td>11</td>
<td>72%</td>
<td>28%</td>
</tr>
<tr>
<td>meta_qsort</td>
<td>202</td>
<td>40</td>
<td>41</td>
<td>430</td>
<td>71</td>
<td>10%</td>
<td>60%</td>
</tr>
<tr>
<td>prover</td>
<td>147</td>
<td>39</td>
<td>65</td>
<td>156</td>
<td>20</td>
<td>41%</td>
<td>32%</td>
</tr>
<tr>
<td>browse</td>
<td>110</td>
<td>34</td>
<td>36</td>
<td>164</td>
<td>80</td>
<td>57%</td>
<td>19%</td>
</tr>
<tr>
<td>unify</td>
<td>160</td>
<td>58</td>
<td>83</td>
<td>80</td>
<td>20</td>
<td>38%</td>
<td>44%</td>
</tr>
<tr>
<td>flatten</td>
<td>138</td>
<td>45</td>
<td>70</td>
<td>29</td>
<td>12</td>
<td>33%</td>
<td>41%</td>
</tr>
<tr>
<td>slda</td>
<td>189</td>
<td>58</td>
<td>78</td>
<td>59</td>
<td>23</td>
<td>36%</td>
<td>44%</td>
</tr>
<tr>
<td>reducer</td>
<td>172</td>
<td>44</td>
<td>105</td>
<td>75</td>
<td>41</td>
<td>40%</td>
<td>42%</td>
</tr>
<tr>
<td>boyer</td>
<td>177</td>
<td>36</td>
<td>137</td>
<td>1376</td>
<td>167</td>
<td>10%</td>
<td>62%</td>
</tr>
<tr>
<td>simple_analyzer</td>
<td>359</td>
<td>121</td>
<td>180</td>
<td>228</td>
<td>79</td>
<td>50%</td>
<td>25%</td>
</tr>
<tr>
<td>nand</td>
<td>710</td>
<td>136</td>
<td>185</td>
<td>435</td>
<td>23</td>
<td>69%</td>
<td>5%</td>
</tr>
<tr>
<td>chat_parser</td>
<td>8420</td>
<td>2547</td>
<td>2182</td>
<td>9115</td>
<td>204</td>
<td>40%</td>
<td>16%</td>
</tr>
<tr>
<td>chat_jig</td>
<td>11685</td>
<td>2006</td>
<td>3756</td>
<td>64880</td>
<td>648</td>
<td>24%</td>
<td>44%</td>
</tr>
<tr>
<td>aquarium_compiler</td>
<td>11709</td>
<td>2675</td>
<td>4555</td>
<td>1650</td>
<td>71</td>
<td>24%</td>
<td>49%</td>
</tr>
<tr>
<td>kish_andi</td>
<td>2331</td>
<td>651</td>
<td>881</td>
<td>652819</td>
<td>119</td>
<td>39%</td>
<td>38%</td>
</tr>
<tr>
<td>symbolic</td>
<td>5761</td>
<td>309</td>
<td>787</td>
<td>5624</td>
<td>620</td>
<td>54%</td>
<td>27%</td>
</tr>
<tr>
<td>tricia</td>
<td>1585</td>
<td>428</td>
<td>844</td>
<td>202</td>
<td>68</td>
<td>23%</td>
<td>58%</td>
</tr>
<tr>
<td>reader</td>
<td>640</td>
<td>119</td>
<td>179</td>
<td>117</td>
<td>50</td>
<td>42%</td>
<td>11%</td>
</tr>
<tr>
<td>bangspec</td>
<td>1151</td>
<td>293</td>
<td>481</td>
<td>1063</td>
<td>126</td>
<td>37%</td>
<td>31%</td>
</tr>
</tbody>
</table>

7.4.2 Direct Construction

An alternative method to implement the topological clash widening was presented in Section 6.6.1. The idea is to construct a type graph of the required shape using a method similar to the method used by Janssens'. The details are different since the topological clash widening has a different shape requirement than that used in Janssens' widening. The widening uses the shape of the old type graph to impose shape requirement on the constructed type graph.

One advantage with this method is that no upper bound operations are needed as part of the widening internals, thus avoiding the sometimes huge type graphs that occur using the original method based on successive transformations of a type graph.

Another advantage is that by directly constructing the type the widening will never\(^8\) have to resort to using an any-node to ensure termination. This makes the direct construction method more precise than the transformation based implementation used in the previous section, it also makes the method

---

\(^8\)At least not for any of the benchmarks on which the widening was used.
more robust, precision appears to be less affected by, for example, scheduling policy.

The implementation used in the evaluation use minimized grammar graphs to represent the types. For this reason the argument used to steer the widening is converted to a proper type graph. The resulting type graph is then transformed to a minimized grammar graph.

In the tables the size of the largest encountered type accounts for both the grammar graphs used as the type-domain and the type graphs used temporarily as input to and output from the direct construction algorithm.

The procedure that constructs a type graph of the required shape obtains a set of nodes for which it should construct an upper bound while adhering to the shape requirements. By default a backward arc is used if there is an ancestor that has already been created to represent the same set of nodes. An alternative that will result in smaller types uses the subset heuristic described in Section 6.6.1, a kind of “poor man’s $\subseteq$-minimization”. This heuristic introduce a backward arc also if there is an ancestor node representing the upper bound of a strictly larger set of nodes.
The analyzer is modified slightly from the description in Section 4.1 so that upper bound of the old and new type is not performed by the analyzer framework. Since the direct construction method creates an upper bound of a set of nodes it can simply be called on a set containing the nodes corresponding to the roots of the two types.

The first variant (Table 7.12) use the subset heuristic to obtain smaller types. It uses FIFO scheduling and analyze clauses separately. Even though the subset heuristic is used it suffers from the problems inherent in the topological clash widening, that is, excessive intermediate sizes when inferring recursive types. This is clearly seen in the entry for kishandi.

The second variant (Table 7.13) use single clausal form as well as the subset heuristic.

The size of the largest intermediate type is now more reasonable. The result without the subset heuristic (not shown) is similar except when analyzing chat.parser for which a huge type appears.

The precision for all three variants are exactly the same so only the version in Table 7.13, denoted “TCXLSC”, is presented in Appendix A. Only the $\sqcup$-Minimization “widening” is consistently as precise as this widening.
7.5 SPANNING TREE WIDENING

To fully utilize the compactness of minimized grammar graphs requires a widening that operate directly on that representation without having to create a type graph even as an intermediate step during widening.

The spanning tree widening described in Section 6.5 is inspired by Janssens’ widening but restricts the shape on paths on one of the many spanning trees of a grammar graph instead of the single spanning tree of a type graph.

The spanning tree widening restricts the shape of paths of a spanning tree by limiting the number of occurrences of “similar” nodes along a path. For Janssens’ widening two nodes along a path are considered “similar” if they are functor nodes with the same function symbol. This notion of similarity are also used for the spanning tree widening. In addition two or-nodes along a path of a spanning tree are deemed “similar” in this sense if the principal labels of the ancestor is the same or a superset of the principal labels of the descendan or-node. As for Janssens’ widening the number of “similar” nodes along a path is limited by a parameter \(k\).

<table>
<thead>
<tr>
<th>Program</th>
<th>Iter</th>
<th>(\nabla_{CP})</th>
<th>(\nabla_{SP})</th>
<th>max size</th>
<th>max result</th>
<th>Nonvar</th>
<th>Unkn</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt</td>
<td>51</td>
<td>12</td>
<td>15</td>
<td>30</td>
<td>5</td>
<td>72%</td>
<td>28%</td>
<td>2s</td>
</tr>
<tr>
<td>meta_qsort</td>
<td>146</td>
<td>31</td>
<td>35</td>
<td>58</td>
<td>16</td>
<td>10%</td>
<td>60%</td>
<td>3s</td>
</tr>
<tr>
<td>prover</td>
<td>145</td>
<td>38</td>
<td>62</td>
<td>83</td>
<td>20</td>
<td>41%</td>
<td>32%</td>
<td>5s</td>
</tr>
<tr>
<td>browse</td>
<td>57</td>
<td>19</td>
<td>31</td>
<td>79</td>
<td>9</td>
<td>57%</td>
<td>19%</td>
<td>4s</td>
</tr>
<tr>
<td>unify</td>
<td>139</td>
<td>58</td>
<td>72</td>
<td>56</td>
<td>11</td>
<td>38%</td>
<td>44%</td>
<td>5s</td>
</tr>
<tr>
<td>flatten</td>
<td>138</td>
<td>45</td>
<td>70</td>
<td>30</td>
<td>12</td>
<td>33%</td>
<td>41%</td>
<td>3s</td>
</tr>
<tr>
<td>sdelta</td>
<td>146</td>
<td>44</td>
<td>63</td>
<td>41</td>
<td>16</td>
<td>35%</td>
<td>45%</td>
<td>3s</td>
</tr>
<tr>
<td>reducer</td>
<td>157</td>
<td>38</td>
<td>95</td>
<td>61</td>
<td>35</td>
<td>24%</td>
<td>58%</td>
<td>4s</td>
</tr>
<tr>
<td>boyer</td>
<td>177</td>
<td>36</td>
<td>137</td>
<td>149</td>
<td>129</td>
<td>10%</td>
<td>62%</td>
<td>12s</td>
</tr>
<tr>
<td>simple_analyzer</td>
<td>322</td>
<td>99</td>
<td>170</td>
<td>116</td>
<td>16</td>
<td>43%</td>
<td>33%</td>
<td>18s</td>
</tr>
<tr>
<td>nand</td>
<td>660</td>
<td>118</td>
<td>159</td>
<td>59</td>
<td>17</td>
<td>69%</td>
<td>5%</td>
<td>20s</td>
</tr>
<tr>
<td>chat_parser</td>
<td>8221</td>
<td>2187</td>
<td>1827</td>
<td>671</td>
<td>251</td>
<td>40%</td>
<td>16%</td>
<td>467s</td>
</tr>
</tbody>
</table>

Table 7.14: Spanning Tree widening, restrict both functor-nodes and or-nodes, \(k = 1\). Minimized grammar graph domain. Clauses analyzed separately. FIFO scheduling policy.
Two variants are evaluated. The first variant considers both these criteria of similarity, that is for each functor only \( k \) functor-nodes with that functor are allowed along a path in the spanning tree used by the widening. In addition an or-node is only allowed to have \( k - 1 \) ancestors with the same or larger set of principal labels. The first variant was evaluated with \( k = 1 \) (Table 7.14) and \( k = 2 \) (Table 7.15). In the precision tables in Appendix A these are denoted “SPA1”\(^9\) and “SPA2” respectively.

The second variant only limits the occurrences of or-nodes along a path. Since any number of function symbols may occur this makes the domain infinite. As discussed in Section 6.5.2 it is an open question if the domain has infinite ascending chains or not, that is, if this variant of the method ensures stationarity and thus termination of the analysis. The second variant was evaluated with \( k = 1 \) (Table 7.16) and \( k = 2 \) (Table 7.17). In Appendix A these are denoted “SPS1”\(^10\) and “SPS2” respectively. Note that this variant does terminate when the first variant does.

\(^9\)For Spanning tree All criteria.
\(^10\)For Spanning tree Subset criteria.
For both variants the size of the largest encountered grammar graph is quite small, especially compared to the type graph widenings earlier in this chapter.

For \( k = 2 \) however both variants fail to analyze benchmark `chat.parser`. The reason is not that enormous types are encountered, instead it is a problem with the complexity of the widening itself. The central idea of this widening is the same as used by Janssens' widening and the direct construction variant of the topological clash widening. That is, a procedure that creates an upper bound of a set of nodes while at the same time ensuring that some (widening specific) shape criteria is satisfied by the created type\textsuperscript{11}.

The procedure works by creating the nodes of the new type depth first, where each node is the upper bound of a set of nodes from the original type. If the shape requirements cannot be satisfied when creating the upper bound for a set of nodes then an earlier level is restarted with a larger set of nodes and the depth first construction resumes closer to the root. However,

\textsuperscript{11}A consequence of this is that the explicit creation of an upper bound as input to the widening can be avoided as was done in Section 7.4.2 for the topological clash widening. This has not been evaluated for the spanning tree widening.
each such restart will discard all the work performed below the restarted level.

What seems to happen when analyzing chat parser with \( k = 2 \) is that the creation of the type proceeds extremely slowly because a lot of work is repeatedly performed and discarded due to restarts. A number of heuristics seems possible to alleviate this problem but none have been evaluated.

This problem is also inherent in the other methods that use the same basic technique to construct a suitable type, that is, Janssens’ widening in Section 6.3 and the spanning tree widening in Section 6.5.

### 7.6 \( \sqsubseteq \)-Minimization WIDENING

The final method evaluated is not a widening at all, at least not in the strict sense. This method is obtained by using type graphs but performing all upper bound operations by converting the arguments to minimized grammar graphs, performing the upper bound to obtain a minimized grammar graph that is then converted back to a type graph using \( \sqsubseteq \)-minimization. See


<table>
<thead>
<tr>
<th>Program</th>
<th>Iter</th>
<th>$\nabla_{CP}$</th>
<th>$\nabla_{SP}$</th>
<th>max size</th>
<th>max result</th>
<th>Nonvar</th>
<th>Unkn</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt</td>
<td>71</td>
<td>19</td>
<td>15</td>
<td>12</td>
<td>10</td>
<td>72%</td>
<td>28%</td>
<td>2 s</td>
</tr>
<tr>
<td>meta_qsort</td>
<td>247</td>
<td>47</td>
<td>43</td>
<td>778</td>
<td>140</td>
<td>10%</td>
<td>60%</td>
<td>102 s</td>
</tr>
<tr>
<td>prover</td>
<td>188</td>
<td>30</td>
<td>67</td>
<td>74</td>
<td>47</td>
<td>41%</td>
<td>32%</td>
<td>5 s</td>
</tr>
<tr>
<td>browse</td>
<td>110</td>
<td>34</td>
<td>36</td>
<td>173</td>
<td>173</td>
<td>57%</td>
<td>19%</td>
<td>46 s</td>
</tr>
<tr>
<td>unify</td>
<td>160</td>
<td>58</td>
<td>83</td>
<td>64</td>
<td>31</td>
<td>38%</td>
<td>44%</td>
<td>4 s</td>
</tr>
<tr>
<td>flatten</td>
<td>138</td>
<td>45</td>
<td>70</td>
<td>20</td>
<td>19</td>
<td>33%</td>
<td>41%</td>
<td>2 s</td>
</tr>
<tr>
<td>sdda</td>
<td>196</td>
<td>60</td>
<td>80</td>
<td>46</td>
<td>30</td>
<td>36%</td>
<td>44%</td>
<td>4 s</td>
</tr>
<tr>
<td>reducer</td>
<td>199</td>
<td>57</td>
<td>119</td>
<td>69</td>
<td>69</td>
<td>31%</td>
<td>52%</td>
<td>6 s</td>
</tr>
<tr>
<td>boyer</td>
<td>177</td>
<td>36</td>
<td>137</td>
<td>340</td>
<td>280</td>
<td>10%</td>
<td>62%</td>
<td>17 s</td>
</tr>
<tr>
<td>simple_analyzer</td>
<td>627</td>
<td>280</td>
<td>302</td>
<td>424</td>
<td>226</td>
<td>50%</td>
<td>25%</td>
<td>79 s</td>
</tr>
<tr>
<td>nand</td>
<td>650</td>
<td>118</td>
<td>164</td>
<td>105</td>
<td>76</td>
<td>69%</td>
<td>5%</td>
<td>25 s</td>
</tr>
<tr>
<td>chat_big</td>
<td>11805</td>
<td>2054</td>
<td>3842</td>
<td>10825</td>
<td>5416</td>
<td>24%</td>
<td>44%</td>
<td>1233 s</td>
</tr>
<tr>
<td>aquarius_compiler</td>
<td>11593</td>
<td>2680</td>
<td>4504</td>
<td>693</td>
<td>203</td>
<td>24%</td>
<td>49%</td>
<td>320 s</td>
</tr>
<tr>
<td>kishandi</td>
<td>2472</td>
<td>707</td>
<td>906</td>
<td>603</td>
<td>388</td>
<td>44%</td>
<td>34%</td>
<td>254 s</td>
</tr>
<tr>
<td>symbolic1</td>
<td>6301</td>
<td>319</td>
<td>787</td>
<td>1916</td>
<td>759</td>
<td>54%</td>
<td>27%</td>
<td>1217 s</td>
</tr>
<tr>
<td>tricia</td>
<td>1575</td>
<td>427</td>
<td>839</td>
<td>228</td>
<td>228</td>
<td>23%</td>
<td>58%</td>
<td>46 s</td>
</tr>
<tr>
<td>reader</td>
<td>892</td>
<td>171</td>
<td>238</td>
<td>222</td>
<td>75</td>
<td>42%</td>
<td>11%</td>
<td>32 s</td>
</tr>
<tr>
<td>baSpec</td>
<td>1156</td>
<td>295</td>
<td>485</td>
<td>833</td>
<td>832</td>
<td>37%</td>
<td>31%</td>
<td>45 s</td>
</tr>
</tbody>
</table>

Table 7.18: Upper bound using $\subseteq$-Minimization as sole type graph “widening”. Clauses analyzed separately, FIFO scheduling policy. Note that the analyzer exhausts memory for benchmark chat.parser.

Section 6.2 for details, including why this is not a widening in the strict sense and thus might not guarantee termination of the analysis.

The results for this “widening” (Table 7.18) are surprisingly good. The encountered type graphs are rather small, approaching the small size of using minimized grammar graphs. Furthermore this is the most precise “widening”, only the topological clash widening using direct construction is consistently of comparable precision.

There are two drawbacks with this method. Firstly, it is not a widening since it does not guarantee stationarity. It is therefore not clear if the analyzer framework used in this thesis will always terminate with this method. Secondly, when analyzing benchmark chat.parser the analyzer eventually exhausts memory, losing all statistics in the process.

7.7 PRECISION COMPARISON

The tables in Appendix A show the precision obtained with the methods presented in this chapter. The precision is broken down into “tags”, as discussed in Section 4.3.2.3. The precision of the methods are presented
roughly in order of decreasing precision. The mnemonics from Appendix A are used to identify the widenings and their variants.

**LEQ** \([-]\)-minimization as part of upper bound, Section 7.6.

This is the most precise widening. It fails to analyze benchmark `chat-parser`.

**TCXLSC** Topological Clash widening by direct construction, Section 7.4.2.

The precision is close to “LEQ”, it is always less than one percent from “LEQ”. For some benchmarks (`aquarius_compiler` and `kish.andi`) other widenings are more precise but only “LEQ” is consistently as good or better.

None of the transformation based variants of the topological clash widening (“TC”, “CT”, “TCSC”) are consistently more precise and they are often much less precise.

This widening manages to analyze all benchmarks.

**SPA** Spanning Tree widening on minimized grammar graphs restricting both or-nodes and functor-nodes. For the same parameter \(k\) this widening is always at least as precise as Janssens’ widening (when Janssens’ widening is not aborted due to excessive resource usage).

“SPA2” does not terminate for benchmark `chat-parser`.

**SPS** Spanning Tree widening on minimized grammar graphs restricting only or-nodes. With \(k = 1\) (“SPS1”) it gives better precision than “SPA1”. With \(k = 2\) (“SPS2”) it gives better precision than “SPA2” only for benchmark `reducer`.

“SPS2” does not terminate for benchmark `chat-parser`.

**J2** Janssens’ widening \(k = 2\). Never more precise than “SPA2”. Infeasibly expensive for several benchmarks.

**J1L,J2L** Janssens’ widening with \([-]\)-minimization. About as precise as “Jk” for the same \(k\). Does not succeed on any benchmarks for which “LEQ” does not succeed, furthermore “LEQ” is more precise.

**Jungle** Less precise than “J1” only on `aquarius_compiler` but without any performance problems.

**Funct** Least precise of all widenings, especially if the tags are considered. Does not have performance problems.
In this thesis I evaluated abstract interpretation using abstract domains based on term grammars and accompanying widenings as a method to infer recursive types for the arguments with which predicates are called.

Two classes of benchmarks were used to study the characteristics of the widenings and the representations used for the underlying term grammar. Traditionally such studies uses “realistic” benchmarks in order to get an indication of the performance that could be expected in practice. This approach was also used in this thesis and two sets of benchmarks were considered that have been used by a number of researchers to evaluate analyses. However, as was shown in Section 4.4, these benchmarks are likely to give a quite unrealistic estimate of the absolute precision that could be expected from an analysis. The reason is that each benchmark program comes equipped with input data that is actually present in the analyzed program and therefore available also to the analyzer. For the Aquarius benchmark this was shown to have a substantial impact on the analysis results (Section 7.1.2). While these benchmark may be used to compare relative precision using the Aquarius benchmark will make a method look much better in absolute terms than what could be expected for more typical programs. The other set of benchmarks, the GAIA benchmarks, were shown to have these and other problems to the extent that it must be regarded as completely unsuitable for benchmarking purposes.

To study the efficiency and the relative precision of the methods a number of large benchmarks were used to complement the Aquarius benchmarks. Many of these benchmarks manipulate recursive data structures with a non-trivial number of constructors. However, in many cases no information was obtained by the analysis for the recursive data structures. These benchmarks will therefore not expose methods that are only efficient when they give little usable information.

The synthetic benchmark $\text{tree}_n$ corresponds to a simple program that traverses a recursive data structure. The number of constructors $n$ used in
the data structure can be increased to give an estimate of how well an analysis method performs in the ideal case when precision about the recursive data structure is not lost. The synthetic benchmarks were then used to show that several methods require intermediate grammar representations that are exponential in \(n\).

There are many parameters that can effect the efficiency and precision of an analysis using a particular abstract domain and widening. In order to keep the number of analyzer specific parameters to a minimum a very simple mono-variant analyzer framework was used. The impact of the scheduling policy was investigated, that is, the order clauses are picked off the worklist for analysis. For some widenings the scheduling policy was shown to have sometimes surprising impact on the efficiency of the analysis. Another aspect that was investigated was when the widening should be applied. The default method applies the widening after the result from each clause becomes available, this propagates change as soon as possible to the other parts of the analyzed program. Another method was also used where the result of the analysis of all clauses are combined before widening is applied, for some methods this was critical in order to obtain reasonable performance. Finally, a number of methods specific to each widening was devised and investigated.

In Chapter 5 the type graph representation of term grammars was investigated. Type graphs were proposed by Janssens and Bruynooghe [32] and are a directed graph representation of a term grammar such that the graph has an unique spanning tree. It is easy to see that the tree-like shape imposed on type graphs can make a type graph exponentially larger than the grammar it represents. What is not so obvious is if this causes problems in practice but the measurements revealed that this is indeed the case. Another aspect of the shape requirement on type graphs is that the worst case complexity of the upper bound operation is super-polynomial, whereas the corresponding operation on term grammar has quadratic complexity. The poor worst case complexity also affects the containment operation used by Janssens and Bruynooghe but a method with quadratic complexity can be used instead (Section 3.3.3.1).

The type graph widening used by Janssens and Bruynooghe was described in Section 5.3, where also a fix for a termination problem in the original presentation of their algorithm were proposed. The performance of this widening when inferring recursive types were investigated using the synthetic benchmarks. A number of heuristics were tried including the analyzer specific parameters outlined above. The results were, however, discouraging. In all cases it seems that inferring the recursive type of \(n\) constructors manipulated by \texttt{tree}_n will produce type graphs that have a size that is exponential in \(n\).
Evaluation on the larger benchmarks show that analysis using the method by Janssens and Bruynooghe will encounter infeasibly large type graphs. Furthermore, the precision comparison in Chapter 7 show that there are other methods that are more precise and also methods that are as precise but that only require, in comparison, small grammar representations.

Hentenryck, Cortesi, and Le Charlier proposed another widening on type graphs 5.4. Their widening specifies a relation between the shape of an old and a new type graph. They also outlined a method that will repeatedly transform the new type graph until it has the required shape relative the old type graph. A critical component of their transformation based method is the technique used to obtain an upper bound of two type graph nodes with a limited size. However, no such method was suggested. In Section 5.4 one such method was proposed that obtains an upper bound of the two nodes and then shrinks the result using a number of heuristics. This method suffers from two problems, the upper bound can become very large and it is sometimes impossible to shrink the result sufficiently in which case the imprecise any will have to be used which results in poor precision for some benchmarks. However, a type graph with the required shape can also be constructed directly using a fundamentally different technique, presented in Section 6.6.1, instead of through repeated transformations. The direct construction method avoids the performance and precision problems of the transformation based method. The precision comparison in Chapter 7 show that the direct construction implementation of the topological clash widening is the most precise of all the widenings investigated.

Another critical aspect of the topological clash widening that was not treated by Hentenryck et al. is the tremendous impact of when the widening is applied. Using the default method in the analyzer where the widening is applied as soon as the result from one clause is available will make this widening infeasible. In particular, for the synthetic benchmark tree\(n\) inferring the type of a recursive data structure with \(n\) constructors appears to require type graphs of size exponential in \(n\). The problem can be seen also for the larger benchmarks. The performance is completely different if widening is instead postponed until the result is available from the analysis of all clauses of a predicate. If widening is postponed in this manner then the topological clash widening will approximate the type graphs in a more aggressive way. For the synthetic benchmark tree\(n\) analysis now require only linear sized type graphs and a constant number of widenings. With widening postponed and the novel direct construction implementation all benchmarks can be analyzed and, as noted above, the precision is better than for any other widening.

Even more precise, however, is a method that is not a proper widening. The \(\sqsubseteq\)-minimization method in Section 6.2 constructs a type graph by replacing nodes with backward arcs to less precise ancestors if possible. In
Section 6.2.1 it was shown that this may not in general guarantee termination of the analysis. The reason is that this method, in principle, makes it possible for the analysis result of a predicate to be updated infinitely many times. However, the performance in practice is surprisingly good, even though the troublesome type graphs were used as the underlying representations of term grammars.

Grammar graphs were introduced in Section 3.3.2 to overcome the problems inherent in the type graph representation. A grammar graph is like a type graph but does not have the tree-shape requirement, a grammar graph therefore have the same, or somewhat smaller, size as the term grammar it represents.

In Chapter 6 widenings and other techniques were investigated that use minimized grammar graphs, that is, where no two nodes in the graph are equivalent. This representation was used also with the direct construction method for the topological clash widening outlined above. However, the tree-shape of type graphs is a critical component of the topological clash widening. This means that even if the minimized grammar graphs are used to provide a compact representations of term grammars the grammar graph must be converted to a type graph as part of the topological clash widening. The tree-shape of the resulting type graph may, however, make it exponentially larger then the grammar graph. To avoid the troublesome type graphs and fully utilize the compactness of the grammar graph representation requires widenings that can operate directly on grammar graphs.

The Spanning Tree widening (Section 6.5) was inspired by the type graph widening by Janssens and Bruynooghe. Their method restricts the paths in the unique spanning tree of a type graph by the requirement that only \( k \) occurrences of the same function symbols is allowed along any such path, for some fixed \( k \). The spanning tree widening generalizes this in two ways. First, it restricts the paths in one of the many possible spanning trees of a grammar graph. Second, the spanning tree widening also restricts occurrences of the or-nodes, corresponding to disjunctive types, along a path. By only considering a spanning tree this method need only consider a tree linear in the size of the grammar graph and by considering also or-nodes a more aggressively approximating method is obtained. The evaluation both on the synthetic benchmarks and the larger benchmarks show that the spanning tree widening does not have the problems with large representations of the term grammars that the type graph based methods have. The spanning tree method is as precise as the widening by Janssens and Bruynooghe but it is not as precise as the direct construction implementation of the topological clash widening.

The second widening that operates directly on grammar graph is the type jungle widening, Section 5.2. It is even more aggressive than the spanning
tree widening and permits only one functor node for each function symbol in a type. It can be implemented efficiently and is perhaps the only truly scalable method investigated in this thesis. It is however not as precise as the spanning tree widenings. Interestingly it is as precise as the widening of Janssens and Bruynooghe with the least precise setting \( k = 1 \) of their method.

To actually implement, or at least consider implementing, a method from the literature necessitates a fairly close look at the details of the method. This in turn sometimes brings up mistakes and errors in the proposed methods. As mentioned above the algorithm originally presented for the widening by Janssens and Bruynooghe [31] will not always terminate. Two methods that have similarities with some of the widenings in this thesis were also found to contain errors. The first method, by Gallagher and de Waal [22], use a similar criteria as the spanning tree widening to limit the length of paths in a type. However, as was shown in Section 6.5.5, the technique they propose for enforcing this path property is not guaranteed to terminate. The second method, by Sağlam and Gallagher [50], use a method similar to \( \subseteq \)-minimization, unfortunately, neither of these methods guarantees that the analysis terminates (Section 6.2.4). Finally, the termination related proofs presented by Hentenryck et al. [30] for their topological clash widening are incorrect. An adapted and corrected proof appears in Appendix B.

Returning to the original objective, to provide a method for type analysis suitable for inclusion in a compiler, what have we learned? First of all, the type graph representation of term grammars is inherently brittle due to its tendency to grow exponentially larger than what it is intended to represent. Furthermore, using type graphs as the term grammar representation makes also the domain operations asymptotically and unnecessarily expensive. Relaxing the tree-shape requirement therefore seems a much more viable alternative. The grammar graph representation is one possibility, if it is used it should be investigated whether keeping it minimized at all times, as was done in the evaluations in this thesis, is really worthwhile.

Unfortunately, the most precise methods, the direct construction implementation of the topological clash widening and the \( \subseteq \)-minimization method, require that a type graph is present at least temporarily. However, since the type graph is created from a small minimized grammar graphs it would be easy to enforce some, possibly ad-hoc, limit on the growth that occur during the creation of the type graph.

Another route would be to try to increase the precision when using the spanning tree widening or even the type-jungle widening. Clearly it can be assumed that precision could be substantially improved using, for example, polyvariance and better handling of variables and builtins. It is even pos-
sible that the less precise widening methods would benefit more from these improvements than what more costly and precise methods would.

Another interesting question is raised by the $\subseteq$-minimization method. This method is, in theory, not guaranteed to terminate but, in practice, seems likely to terminate ahead of most other methods, with more precise result. As discussed in Section 6.2 one explanation for the aggressive but still precise node replacement performed by the $\subseteq$-minimization, lies in a fundamental fact about the intermediate approximate types produced during analysis: Nodes further from the root of a type tend to correspond to earlier and thus more precise approximations of the same type. This is utilized by the $\subseteq$-minimization by replacing references to such more precise nodes by references back to the less precise root node. Especially for the synthetic benchmark $\text{tree}_a$, a natural question is then if a more direct method could infer the correct type immediately. The idea of using a single representative for the type of procedure argument or a program variable is used in set-based analysis [27, 44]. Cousot and Cousot [12] show that set based analysis can be regarded as an abstract interpretation using techniques similar to those considered in this thesis. They also outline a grammar based method that obtains a finite domain by, roughly, restricting the number of non-terminals in the grammar to one for each program variable. The behavior of the $\subseteq$-minimization method indicate that a method based on these ideas may give a both precise and efficient analysis.


Appendix A


Precision on Larger Benchmarks

A.1 Overview of Tables

This appendix lists the precision results for the widenings in Chapter 7. For brevity benchmarks with less than about one hundred argument positions are omitted from the tables.

Each table presents the analysis result of all widenings for one benchmark. The number of reachable arguments are shown as well as the number of reached arguments that the separate analysis of uninitialized variables determines are always a variable or sometimes (C) a variable. Column ⊥ shows the number of arguments of the reachable predicates that type analysis determines will not be reached. The percentage of uninitialized variables refers to the reached arguments only.

The analysis precision is summarized as tags or “top-level” types (Definition 4.3.1, Page 89), described again below for convenience. Note that all tags are mutually exclusive, an argument is only counted in the column corresponding to the most precise tag that describes the type of the argument.

The last two columns list the total number of widenings and the number of analyzed clauses (“iterations”) to give some indication of the domain independent work performed by the analyzer. Other performance measures appear in the tables for each widening in Chapter 7.

The detailed column headings are:

**Kind** This column contains mnemonics for the analyzer parameters used. These are described on the next page.

**NI** The number of arguments that are [] (also known as nil).

---

1The number of reached arguments is taken to be the largest value in column ⊥ subtracted from the number of reachable arguments.
CO The number of arguments that are lists cells (also known as cons).

LI The number of arguments whose type is a disjunction of NI and CO.
Note that this not necessarily corresponds to proper lists.

ST Compound structures including list cells.

DI Atoms (also known as symbols, non-numerical constants) including the empty list.

HY An argument that is either an atom or a compound structure, that is, terms that are neither numbers nor variables.

NUM Numbers.

NV Any kind of non-variable term.

Unreached arguments. The sum of the arities of predicates that are reachable from the main entry-point but that the analyzer can determine will never be called.

$\nabla_{CP+SP}$ The sum of the number of widenings for call patterns and success patterns. Note that for the analyses performed using single clausal form the number of widenings often are significantly fewer since there are only one clause for each predicate.

Iter The number of analyzed clause bodies. This is the number of calls to procedure ai.clause in the analyzer (Section 4.1.3). Note that the clauses are fewer but larger for analysis with predicates in single clausal form. This column is therefore not comparable between the analyses using single clausal form and those that do not.

The widenings and their variants are denoted:

Funct The simple functors widening on minimized grammar graphs described in Section 5.1. The performance measurements appear in Table 7.1, page 184.

Jungle The type-jungle widening on minimized grammar graphs described in Section 5.2. The performance measurements appear in Table 7.4, page 188.

J1 Janssens’ type graph widening with $k = 1$ described in Section 5.3. The performance measurements appear in Table 7.5, page 190.

J2 Janssens’ widening with $k = 2$. The performance measurements appear in Table 7.6, page 191.
**J1L** Janssens’ type graph widening with \( k = 1 \) but with \( \sqsubseteq \)-minimization applied as part of upper bound. The performance measurements appear in Table 7.7, page 192.

**J2L** As J1L but with \( k = 2 \). The performance measurements appear in Table 7.8, page 193.

**TC** Topological Clash widening of Hentenryck et al. for type graphs. Described in Section 5.4. This version uses FIFO scheduling of the worklist and analyze each clause separately (that is, not in single-clausal form). The performance measurements appear in Table 7.9, page 195.

**CT** As TC but uses the unfair LIFO scheduling policy for the worklist. The performance measurements appear in Table 7.10, page 197.

**TCSC** As TC but predicates analyzed in single-clausal form. The performance measurements appear in Table 7.11, page 198.

**TCXLSC** Topological Clash widening for type graphs using the direct construction and the subset heuristic to obtain smaller type graphs. Described in Section 6.6.1. Predicates analyzed in single-clausal form. The performance measurements appear in Table 7.13, page 200.

**SPA1** Spanning tree widening for minimized grammar graphs with \( k = 1 \). This widening and its variants are described in Section 6.5. This is the most aggressive version, where both function symbols and \( prlb \) subsets are used to limit the length of the paths of the spanning tree of the grammar graph. The performance measurements appear in Table 7.14, page 202.

**SPA2** As SPA1 but with \( k = 2 \). The performance measurements appear in Table 7.15, page 203.

**SPS1** As SPA1 but only restricting the occurrences of or-nodes, that is, \( prlb \) subsets are used to restrict length of paths in the spanning tree. The performance measurements appear in Table 7.16, page 204.

**SPS2** As SPS1 but \( k = 2 \). The performance measurements appear in Table 7.17, page 205.

**LEQ** \( \sqsubseteq \)-Minimization used as a “widening” for type graphs. All upper bound operations are performed using \( \sqsubseteq \)-minimization, no other widening is performed. \( \sqsubseteq \)-minimization is described in Section 6.2. The performance measurements appear in Table 7.18, page 206.
## A.1. Overview of Tables

### Table A.1: simple analyzer precision summary.

<table>
<thead>
<tr>
<th>Kind</th>
<th>NI</th>
<th>CO</th>
<th>LI</th>
<th>ST</th>
<th>DI</th>
<th>HY</th>
<th>NUM</th>
<th>NV</th>
<th>Total</th>
<th>$\n CP+SP$</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jungle</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>15</td>
<td>8</td>
<td>104</td>
<td>305</td>
<td>145</td>
<td>345</td>
</tr>
<tr>
<td>J1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>14</td>
<td>15</td>
<td>8</td>
<td>20</td>
<td>108</td>
<td>269</td>
<td>322</td>
</tr>
<tr>
<td>J2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>14</td>
<td>15</td>
<td>8</td>
<td>20</td>
<td>108</td>
<td>359</td>
<td>428</td>
</tr>
<tr>
<td>J1L</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>14</td>
<td>15</td>
<td>8</td>
<td>20</td>
<td>108</td>
<td>269</td>
<td>322</td>
</tr>
<tr>
<td>J2L</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>14</td>
<td>15</td>
<td>8</td>
<td>20</td>
<td>108</td>
<td>307</td>
<td>356</td>
</tr>
<tr>
<td>TC</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>14</td>
<td>15</td>
<td>5</td>
<td>20</td>
<td>104</td>
<td>303</td>
<td>352</td>
</tr>
<tr>
<td>CT</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>14</td>
<td>15</td>
<td>5</td>
<td>20</td>
<td>104</td>
<td>402</td>
<td>700</td>
</tr>
<tr>
<td>TCSC</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>14</td>
<td>15</td>
<td>5</td>
<td>20</td>
<td>104</td>
<td>251</td>
<td></td>
</tr>
<tr>
<td>TCXLSC</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>28</td>
<td>15</td>
<td>13</td>
<td>20</td>
<td>128</td>
<td>582</td>
<td>627</td>
</tr>
</tbody>
</table>

| SPA1 | 3  | 3  | 4  | 5  | 14 | 15 | 8   | 20 | 108   | 269    | 322  |
| SPA2 | 3  | 3  | 4  | 5  | 14 | 15 | 8   | 20 | 108   | 361    | 426  |
| SPS1 | 3  | 3  | 4  | 4  | 14 | 15 | 5   | 20 | 104   | 303    | 352  |
| SPS2 | 3  | 3  | 4  | 4  | 14 | 15 | 5   | 20 | 104   | 600    | 654  |
| LEQ  | 3  | 3  | 4  | 6  | 28 | 15 | 13  | 20 | 128   | 582    | 627  |

### Table A.2: unify precision summary.

<table>
<thead>
<tr>
<th>Kind</th>
<th>NI</th>
<th>CO</th>
<th>LI</th>
<th>ST</th>
<th>DI</th>
<th>HY</th>
<th>NUM</th>
<th>NV</th>
<th>Total</th>
<th>$\n CP+SP$</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jungle</td>
<td>6</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>3</td>
<td>54</td>
<td>110</td>
<td>137</td>
</tr>
<tr>
<td>J1</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>54</td>
<td>146</td>
<td>159</td>
</tr>
<tr>
<td>J2</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>54</td>
<td>146</td>
<td>159</td>
</tr>
<tr>
<td>J1L</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>54</td>
<td>141</td>
<td>160</td>
</tr>
<tr>
<td>J2L</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>54</td>
<td>142</td>
<td>160</td>
</tr>
<tr>
<td>TC</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>54</td>
<td>192</td>
<td>284</td>
</tr>
<tr>
<td>CT</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>54</td>
<td>120</td>
<td>86</td>
</tr>
<tr>
<td>TCSC</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>54</td>
<td>114</td>
<td>85</td>
</tr>
<tr>
<td>TCXLSC</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>54</td>
<td>114</td>
<td>85</td>
</tr>
<tr>
<td>SPA1</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>54</td>
<td>141</td>
<td>160</td>
</tr>
<tr>
<td>SPA2</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>54</td>
<td>141</td>
<td>160</td>
</tr>
<tr>
<td>SPS1</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>54</td>
<td>152</td>
<td>169</td>
</tr>
<tr>
<td>SPS2</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>54</td>
<td>141</td>
<td>160</td>
</tr>
<tr>
<td>LEQ</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>54</td>
<td>141</td>
<td>160</td>
</tr>
</tbody>
</table>
### Program reducer, 95 reachable args, 17 reached uninit (18%), TC

<table>
<thead>
<tr>
<th>Kind</th>
<th>NI</th>
<th>CO</th>
<th>LI</th>
<th>ST</th>
<th>DI</th>
<th>HY</th>
<th>NUM</th>
<th>NV</th>
<th>↓</th>
<th>Tot</th>
<th>(\sqrt{CP+SP})</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Funct</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>23</td>
<td>125</td>
<td>149</td>
</tr>
<tr>
<td>Jungle</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>23</td>
<td>131</td>
<td>156</td>
</tr>
<tr>
<td>J1</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>23</td>
<td>133</td>
<td>157</td>
</tr>
<tr>
<td>J2</td>
<td>0</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>29</td>
<td>159</td>
<td>186</td>
</tr>
<tr>
<td>J1L</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>23</td>
<td>133</td>
<td>157</td>
</tr>
<tr>
<td>J2L</td>
<td>0</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>29</td>
<td>162</td>
<td>185</td>
</tr>
<tr>
<td>TC</td>
<td>0</td>
<td>6</td>
<td>17</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>38</td>
<td>166</td>
<td>197</td>
</tr>
<tr>
<td>CT</td>
<td>0</td>
<td>6</td>
<td>17</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>38</td>
<td>187</td>
<td>361</td>
</tr>
<tr>
<td>TCSC</td>
<td>0</td>
<td>6</td>
<td>17</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>38</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>TCXLSC</td>
<td>0</td>
<td>6</td>
<td>17</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>38</td>
<td>118</td>
<td>118</td>
</tr>
<tr>
<td>SPA1</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>23</td>
<td>133</td>
<td>157</td>
</tr>
<tr>
<td>SPA2</td>
<td>0</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>29</td>
<td>159</td>
<td>186</td>
</tr>
<tr>
<td>SPS1</td>
<td>0</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>29</td>
<td>161</td>
<td>185</td>
</tr>
<tr>
<td>SPS2</td>
<td>0</td>
<td>6</td>
<td>17</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>38</td>
<td>179</td>
<td>213</td>
</tr>
<tr>
<td>LEQ</td>
<td>0</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>29</td>
<td>176</td>
<td>199</td>
</tr>
</tbody>
</table>

Table A.3: reducer precision summary.

### Program nand, 174 reachable args, 46 reached uninit (26%), 8C

<table>
<thead>
<tr>
<th>Kind</th>
<th>NI</th>
<th>CO</th>
<th>LI</th>
<th>ST</th>
<th>DI</th>
<th>HY</th>
<th>NUM</th>
<th>NV</th>
<th>↓</th>
<th>Tot</th>
<th>(\sqrt{CP+SP})</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Funct</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>0</td>
<td>37</td>
<td>68</td>
<td>0</td>
<td>0</td>
<td>120</td>
<td>286</td>
<td>656</td>
</tr>
<tr>
<td>Jungle</td>
<td>1</td>
<td>6</td>
<td>32</td>
<td>14</td>
<td>14</td>
<td>5</td>
<td>38</td>
<td>13</td>
<td>0</td>
<td>120</td>
<td>277</td>
<td>660</td>
</tr>
<tr>
<td>J1</td>
<td>1</td>
<td>6</td>
<td>32</td>
<td>14</td>
<td>14</td>
<td>5</td>
<td>38</td>
<td>13</td>
<td>0</td>
<td>120</td>
<td>277</td>
<td>660</td>
</tr>
<tr>
<td>J2</td>
<td>1</td>
<td>7</td>
<td>31</td>
<td>14</td>
<td>14</td>
<td>5</td>
<td>38</td>
<td>13</td>
<td>0</td>
<td>120</td>
<td>347</td>
<td>743</td>
</tr>
<tr>
<td>J1L</td>
<td>1</td>
<td>6</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>5</td>
<td>38</td>
<td>0</td>
<td>0</td>
<td>89</td>
<td>352</td>
<td>752</td>
</tr>
<tr>
<td>J2L</td>
<td>1</td>
<td>7</td>
<td>13</td>
<td>14</td>
<td>14</td>
<td>5</td>
<td>38</td>
<td>0</td>
<td>0</td>
<td>89</td>
<td>416</td>
<td>856</td>
</tr>
<tr>
<td>TC</td>
<td>1</td>
<td>5</td>
<td>33</td>
<td>11</td>
<td>14</td>
<td>5</td>
<td>38</td>
<td>13</td>
<td>0</td>
<td>120</td>
<td>321</td>
<td>715</td>
</tr>
<tr>
<td>CT</td>
<td>1</td>
<td>5</td>
<td>33</td>
<td>14</td>
<td>14</td>
<td>5</td>
<td>38</td>
<td>13</td>
<td>0</td>
<td>120</td>
<td>460</td>
<td>1112</td>
</tr>
<tr>
<td>TCSC</td>
<td>1</td>
<td>5</td>
<td>33</td>
<td>14</td>
<td>14</td>
<td>5</td>
<td>38</td>
<td>13</td>
<td>0</td>
<td>120</td>
<td>260</td>
<td>179</td>
</tr>
<tr>
<td>TCXLSC</td>
<td>1</td>
<td>7</td>
<td>31</td>
<td>14</td>
<td>14</td>
<td>5</td>
<td>38</td>
<td>13</td>
<td>0</td>
<td>120</td>
<td>272</td>
<td>182</td>
</tr>
<tr>
<td>SPA1</td>
<td>1</td>
<td>6</td>
<td>32</td>
<td>11</td>
<td>14</td>
<td>5</td>
<td>38</td>
<td>13</td>
<td>0</td>
<td>120</td>
<td>277</td>
<td>660</td>
</tr>
<tr>
<td>SPA2</td>
<td>1</td>
<td>7</td>
<td>31</td>
<td>11</td>
<td>14</td>
<td>5</td>
<td>38</td>
<td>13</td>
<td>0</td>
<td>120</td>
<td>326</td>
<td>724</td>
</tr>
<tr>
<td>SPS1</td>
<td>1</td>
<td>7</td>
<td>31</td>
<td>11</td>
<td>14</td>
<td>5</td>
<td>38</td>
<td>13</td>
<td>0</td>
<td>120</td>
<td>322</td>
<td>732</td>
</tr>
<tr>
<td>SPS2</td>
<td>1</td>
<td>7</td>
<td>31</td>
<td>11</td>
<td>14</td>
<td>5</td>
<td>38</td>
<td>13</td>
<td>0</td>
<td>120</td>
<td>443</td>
<td>911</td>
</tr>
<tr>
<td>LEQ</td>
<td>1</td>
<td>7</td>
<td>31</td>
<td>11</td>
<td>14</td>
<td>6</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>120</td>
<td>282</td>
<td>660</td>
</tr>
</tbody>
</table>

Table A.4: nand precision summary.
# A.1. Overview of Tables

**Table A.5: chat.parser precision summary.**

<table>
<thead>
<tr>
<th>Kind</th>
<th>NI</th>
<th>CO</th>
<th>LI</th>
<th>ST</th>
<th>DI</th>
<th>HY</th>
<th>NUM</th>
<th>NV</th>
<th>⊥</th>
<th>Tot</th>
<th>(\nabla_{CP+SP})</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jungle</td>
<td>10</td>
<td>8</td>
<td>88</td>
<td>50</td>
<td>19</td>
<td>114</td>
<td>2</td>
<td>0</td>
<td>10</td>
<td>301</td>
<td>3830</td>
<td>7727</td>
</tr>
<tr>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J1L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J2L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TC</td>
<td>10</td>
<td>8</td>
<td>88</td>
<td>50</td>
<td>19</td>
<td>114</td>
<td>2</td>
<td>0</td>
<td>10</td>
<td>301</td>
<td>3830</td>
<td>7727</td>
</tr>
<tr>
<td>CT</td>
<td>10</td>
<td>8</td>
<td>88</td>
<td>50</td>
<td>19</td>
<td>114</td>
<td>2</td>
<td>0</td>
<td>10</td>
<td>301</td>
<td>20186</td>
<td>39767</td>
</tr>
<tr>
<td>TCSC</td>
<td>10</td>
<td>8</td>
<td>88</td>
<td>48</td>
<td>19</td>
<td>116</td>
<td>2</td>
<td>0</td>
<td>10</td>
<td>301</td>
<td>4471</td>
<td>2771</td>
</tr>
<tr>
<td>TCXLSC</td>
<td>10</td>
<td>8</td>
<td>88</td>
<td>50</td>
<td>19</td>
<td>116</td>
<td>5</td>
<td>0</td>
<td>10</td>
<td>306</td>
<td>5325</td>
<td>3223</td>
</tr>
<tr>
<td>SPA1</td>
<td>10</td>
<td>8</td>
<td>88</td>
<td>50</td>
<td>19</td>
<td>116</td>
<td>5</td>
<td>0</td>
<td>10</td>
<td>306</td>
<td>4014</td>
<td>8221</td>
</tr>
<tr>
<td>SPA2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPS1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table A.6: chat_big precision summary.**

<table>
<thead>
<tr>
<th>Kind</th>
<th>NI</th>
<th>CO</th>
<th>LI</th>
<th>ST</th>
<th>DI</th>
<th>HY</th>
<th>NUM</th>
<th>NV</th>
<th>⊥</th>
<th>Tot</th>
<th>(\nabla_{CP+SP})</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jungle</td>
<td>12</td>
<td>9</td>
<td>115</td>
<td>76</td>
<td>35</td>
<td>108</td>
<td>13</td>
<td>5</td>
<td>0</td>
<td>373</td>
<td>5248</td>
<td>11233</td>
</tr>
<tr>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J1L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J2L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TC</td>
<td>12</td>
<td>9</td>
<td>115</td>
<td>72</td>
<td>35</td>
<td>108</td>
<td>13</td>
<td>9</td>
<td>0</td>
<td>373</td>
<td>5747</td>
<td>11680</td>
</tr>
<tr>
<td>CT</td>
<td>12</td>
<td>9</td>
<td>115</td>
<td>75</td>
<td>35</td>
<td>108</td>
<td>13</td>
<td>6</td>
<td>0</td>
<td>373</td>
<td>19867</td>
<td>32414</td>
</tr>
<tr>
<td>TCSC</td>
<td>12</td>
<td>9</td>
<td>115</td>
<td>72</td>
<td>35</td>
<td>108</td>
<td>13</td>
<td>9</td>
<td>0</td>
<td>373</td>
<td>4197</td>
<td>2584</td>
</tr>
<tr>
<td>TCXLSC</td>
<td>12</td>
<td>9</td>
<td>115</td>
<td>66</td>
<td>35</td>
<td>110</td>
<td>16</td>
<td>15</td>
<td>0</td>
<td>378</td>
<td>4238</td>
<td>2605</td>
</tr>
<tr>
<td>SPA1</td>
<td>12</td>
<td>9</td>
<td>115</td>
<td>76</td>
<td>35</td>
<td>110</td>
<td>16</td>
<td>5</td>
<td>0</td>
<td>378</td>
<td>5350</td>
<td>11406</td>
</tr>
<tr>
<td>SPA2</td>
<td>12</td>
<td>9</td>
<td>115</td>
<td>76</td>
<td>35</td>
<td>110</td>
<td>16</td>
<td>5</td>
<td>0</td>
<td>378</td>
<td>6508</td>
<td>12355</td>
</tr>
<tr>
<td>SPS1</td>
<td>12</td>
<td>9</td>
<td>115</td>
<td>76</td>
<td>35</td>
<td>110</td>
<td>16</td>
<td>5</td>
<td>0</td>
<td>378</td>
<td>5663</td>
<td>11595</td>
</tr>
<tr>
<td>SPS2</td>
<td>12</td>
<td>9</td>
<td>115</td>
<td>76</td>
<td>35</td>
<td>110</td>
<td>16</td>
<td>5</td>
<td>0</td>
<td>378</td>
<td>6705</td>
<td>12604</td>
</tr>
<tr>
<td>LEQ</td>
<td>12</td>
<td>9</td>
<td>115</td>
<td>76</td>
<td>35</td>
<td>110</td>
<td>16</td>
<td>5</td>
<td>0</td>
<td>378</td>
<td>5896</td>
<td>11805</td>
</tr>
</tbody>
</table>
### Table A.7: aquarius Compiler precision summary.

<table>
<thead>
<tr>
<th>Kind</th>
<th>NI</th>
<th>CO</th>
<th>LI</th>
<th>ST</th>
<th>DI</th>
<th>HY</th>
<th>NUM</th>
<th>NV</th>
<th>Total</th>
<th>$\nabla_{CP+SP}$</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Funct</td>
<td>57</td>
<td>8</td>
<td>283</td>
<td>50</td>
<td>188</td>
<td>209</td>
<td>157</td>
<td>260</td>
<td>11</td>
<td>1223</td>
<td>6317</td>
</tr>
<tr>
<td>Jungle</td>
<td>57</td>
<td>53</td>
<td>504</td>
<td>177</td>
<td>189</td>
<td>106</td>
<td>158</td>
<td>4</td>
<td>11</td>
<td>1259</td>
<td>6908</td>
</tr>
<tr>
<td>J1</td>
<td>57</td>
<td>53</td>
<td>504</td>
<td>177</td>
<td>190</td>
<td>110</td>
<td>158</td>
<td>4</td>
<td>11</td>
<td>1264</td>
<td>7039</td>
</tr>
<tr>
<td>J2</td>
<td>57</td>
<td>53</td>
<td>504</td>
<td>177</td>
<td>190</td>
<td>111</td>
<td>158</td>
<td>4</td>
<td>11</td>
<td>1265</td>
<td>6948</td>
</tr>
<tr>
<td>J1L</td>
<td>57</td>
<td>55</td>
<td>516</td>
<td>181</td>
<td>190</td>
<td>111</td>
<td>158</td>
<td>4</td>
<td>11</td>
<td>1283</td>
<td>7095</td>
</tr>
<tr>
<td>J2L</td>
<td>57</td>
<td>55</td>
<td>516</td>
<td>181</td>
<td>190</td>
<td>111</td>
<td>158</td>
<td>4</td>
<td>11</td>
<td>1282</td>
<td>7283</td>
</tr>
<tr>
<td>TC</td>
<td>57</td>
<td>55</td>
<td>515</td>
<td>181</td>
<td>190</td>
<td>111</td>
<td>158</td>
<td>4</td>
<td>11</td>
<td>1275</td>
<td>11027</td>
</tr>
<tr>
<td>CT</td>
<td>57</td>
<td>53</td>
<td>511</td>
<td>180</td>
<td>190</td>
<td>111</td>
<td>158</td>
<td>4</td>
<td>11</td>
<td>1274</td>
<td>6056</td>
</tr>
<tr>
<td>TCSC</td>
<td>57</td>
<td>55</td>
<td>510</td>
<td>178</td>
<td>190</td>
<td>111</td>
<td>158</td>
<td>4</td>
<td>11</td>
<td>1274</td>
<td>6035</td>
</tr>
<tr>
<td>SPA1</td>
<td>57</td>
<td>53</td>
<td>504</td>
<td>177</td>
<td>190</td>
<td>111</td>
<td>158</td>
<td>4</td>
<td>11</td>
<td>1265</td>
<td>6952</td>
</tr>
<tr>
<td>SPA2</td>
<td>57</td>
<td>55</td>
<td>510</td>
<td>170</td>
<td>190</td>
<td>107</td>
<td>158</td>
<td>4</td>
<td>11</td>
<td>1262</td>
<td>7707</td>
</tr>
<tr>
<td>SPS1</td>
<td>57</td>
<td>53</td>
<td>504</td>
<td>178</td>
<td>190</td>
<td>111</td>
<td>158</td>
<td>4</td>
<td>11</td>
<td>1266</td>
<td>7214</td>
</tr>
<tr>
<td>SPS2</td>
<td>57</td>
<td>55</td>
<td>510</td>
<td>170</td>
<td>190</td>
<td>107</td>
<td>158</td>
<td>4</td>
<td>11</td>
<td>1262</td>
<td>7869</td>
</tr>
<tr>
<td>LEQ</td>
<td>57</td>
<td>55</td>
<td>516</td>
<td>181</td>
<td>190</td>
<td>111</td>
<td>158</td>
<td>4</td>
<td>11</td>
<td>1283</td>
<td>7184</td>
</tr>
</tbody>
</table>

### Table A.8: kishandi precision summary.

<table>
<thead>
<tr>
<th>Kind</th>
<th>NI</th>
<th>CO</th>
<th>LI</th>
<th>ST</th>
<th>DI</th>
<th>HY</th>
<th>NUM</th>
<th>NV</th>
<th>Total</th>
<th>$\nabla_{CP+SP}$</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Funct</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>11</td>
<td>35</td>
<td>58</td>
<td>95</td>
<td>95</td>
<td>38</td>
<td>336</td>
<td>1428</td>
</tr>
<tr>
<td>Jungle</td>
<td>0</td>
<td>9</td>
<td>95</td>
<td>94</td>
<td>41</td>
<td>14</td>
<td>110</td>
<td>0</td>
<td>38</td>
<td>401</td>
<td>1205</td>
</tr>
<tr>
<td>J1</td>
<td>0</td>
<td>9</td>
<td>93</td>
<td>94</td>
<td>40</td>
<td>18</td>
<td>113</td>
<td>0</td>
<td>38</td>
<td>405</td>
<td>1209</td>
</tr>
<tr>
<td>J2</td>
<td>0</td>
<td>9</td>
<td>94</td>
<td>99</td>
<td>43</td>
<td>18</td>
<td>114</td>
<td>0</td>
<td>38</td>
<td>415</td>
<td>1786</td>
</tr>
<tr>
<td>TC</td>
<td>0</td>
<td>9</td>
<td>97</td>
<td>94</td>
<td>40</td>
<td>15</td>
<td>118</td>
<td>0</td>
<td>38</td>
<td>411</td>
<td>2591</td>
</tr>
<tr>
<td>CT</td>
<td>0</td>
<td>9</td>
<td>71</td>
<td>94</td>
<td>41</td>
<td>14</td>
<td>113</td>
<td>0</td>
<td>38</td>
<td>380</td>
<td>1340</td>
</tr>
<tr>
<td>TCSC</td>
<td>0</td>
<td>9</td>
<td>97</td>
<td>94</td>
<td>41</td>
<td>14</td>
<td>118</td>
<td>0</td>
<td>38</td>
<td>411</td>
<td>1285</td>
</tr>
<tr>
<td>TCXLSC</td>
<td>0</td>
<td>9</td>
<td>97</td>
<td>94</td>
<td>41</td>
<td>14</td>
<td>118</td>
<td>0</td>
<td>38</td>
<td>411</td>
<td>1285</td>
</tr>
<tr>
<td>SPA1</td>
<td>0</td>
<td>9</td>
<td>96</td>
<td>94</td>
<td>41</td>
<td>14</td>
<td>113</td>
<td>0</td>
<td>38</td>
<td>405</td>
<td>1210</td>
</tr>
<tr>
<td>SPA2</td>
<td>0</td>
<td>9</td>
<td>97</td>
<td>94</td>
<td>41</td>
<td>14</td>
<td>113</td>
<td>0</td>
<td>38</td>
<td>406</td>
<td>2041</td>
</tr>
<tr>
<td>SPS1</td>
<td>0</td>
<td>9</td>
<td>94</td>
<td>94</td>
<td>41</td>
<td>17</td>
<td>113</td>
<td>0</td>
<td>38</td>
<td>406</td>
<td>1750</td>
</tr>
<tr>
<td>SPS2</td>
<td>0</td>
<td>9</td>
<td>97</td>
<td>94</td>
<td>41</td>
<td>14</td>
<td>113</td>
<td>0</td>
<td>38</td>
<td>406</td>
<td>2242</td>
</tr>
<tr>
<td>LEQ</td>
<td>0</td>
<td>11</td>
<td>94</td>
<td>99</td>
<td>51</td>
<td>30</td>
<td>126</td>
<td>0</td>
<td>38</td>
<td>449</td>
<td>1613</td>
</tr>
</tbody>
</table>
### Program tricia, 767 reachable args, 148 reached uninit (19%), 10C

<table>
<thead>
<tr>
<th>Kind</th>
<th>NI</th>
<th>CO</th>
<th>LI</th>
<th>ST</th>
<th>D1</th>
<th>HY</th>
<th>NUM</th>
<th>NV</th>
<th>⊥</th>
<th>Tot</th>
<th>( \nabla_{CP+SP} )</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Funct</td>
<td>17</td>
<td>2</td>
<td>6</td>
<td>14</td>
<td>3</td>
<td>21</td>
<td>44</td>
<td>49</td>
<td>0</td>
<td>156</td>
<td>1055</td>
<td>1273</td>
</tr>
<tr>
<td>Jungle</td>
<td>17</td>
<td>8</td>
<td>38</td>
<td>35</td>
<td>3</td>
<td>12</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>157</td>
<td>1207</td>
<td>1495</td>
</tr>
<tr>
<td>J1</td>
<td>17</td>
<td>8</td>
<td>38</td>
<td>35</td>
<td>3</td>
<td>9</td>
<td>44</td>
<td>3</td>
<td>0</td>
<td>157</td>
<td>1222</td>
<td>1513</td>
</tr>
<tr>
<td>J2</td>
<td>17</td>
<td>8</td>
<td>34</td>
<td>35</td>
<td>3</td>
<td>11</td>
<td>44</td>
<td>1</td>
<td>0</td>
<td>173</td>
<td>1442</td>
<td>1796</td>
</tr>
<tr>
<td>JIL</td>
<td>17</td>
<td>8</td>
<td>37</td>
<td>35</td>
<td>3</td>
<td>13</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>157</td>
<td>1210</td>
<td>1489</td>
</tr>
<tr>
<td>J2L</td>
<td>17</td>
<td>8</td>
<td>53</td>
<td>35</td>
<td>3</td>
<td>15</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>175</td>
<td>1259</td>
<td>1564</td>
</tr>
<tr>
<td>TC</td>
<td>17</td>
<td>8</td>
<td>53</td>
<td>35</td>
<td>3</td>
<td>15</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>175</td>
<td>1283</td>
<td>1601</td>
</tr>
<tr>
<td>CT</td>
<td>17</td>
<td>8</td>
<td>54</td>
<td>35</td>
<td>3</td>
<td>12</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>173</td>
<td>1430</td>
<td>2600</td>
</tr>
<tr>
<td>TCSC</td>
<td>17</td>
<td>8</td>
<td>52</td>
<td>35</td>
<td>3</td>
<td>16</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>175</td>
<td>1097</td>
<td>931</td>
</tr>
<tr>
<td>TCXLSC</td>
<td>17</td>
<td>8</td>
<td>53</td>
<td>35</td>
<td>3</td>
<td>15</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>175</td>
<td>1085</td>
<td>920</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kind</th>
<th>NI</th>
<th>CO</th>
<th>LI</th>
<th>ST</th>
<th>D1</th>
<th>HY</th>
<th>NUM</th>
<th>NV</th>
<th>⊥</th>
<th>Tot</th>
<th>( \nabla_{CP+SP} )</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPA1</td>
<td>17</td>
<td>8</td>
<td>37</td>
<td>35</td>
<td>3</td>
<td>13</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>157</td>
<td>1204</td>
<td>1485</td>
</tr>
<tr>
<td>SPA2</td>
<td>17</td>
<td>8</td>
<td>55</td>
<td>35</td>
<td>3</td>
<td>13</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>173</td>
<td>1406</td>
<td>1760</td>
</tr>
<tr>
<td>SPS1</td>
<td>17</td>
<td>8</td>
<td>53</td>
<td>35</td>
<td>3</td>
<td>13</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>173</td>
<td>1250</td>
<td>1560</td>
</tr>
<tr>
<td>SPS2</td>
<td>17</td>
<td>8</td>
<td>55</td>
<td>35</td>
<td>3</td>
<td>13</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>175</td>
<td>1423</td>
<td>1771</td>
</tr>
<tr>
<td>LEQ</td>
<td>17</td>
<td>8</td>
<td>53</td>
<td>35</td>
<td>3</td>
<td>15</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>175</td>
<td>1266</td>
<td>1575</td>
</tr>
</tbody>
</table>

Table A.9: tricia precision summary.

### Program symbollicl, 98 reachable args, 19 reached uninit (19%), 4C

<table>
<thead>
<tr>
<th>Kind</th>
<th>NI</th>
<th>CO</th>
<th>LI</th>
<th>ST</th>
<th>D1</th>
<th>HY</th>
<th>NUM</th>
<th>NV</th>
<th>⊥</th>
<th>Tot</th>
<th>( \nabla_{CP+SP} )</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Funct</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>23</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>0</td>
<td>53</td>
<td>704</td>
<td>2833</td>
</tr>
<tr>
<td>Jungle</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>814</td>
<td>2238</td>
</tr>
<tr>
<td>J1</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>814</td>
<td>2238</td>
</tr>
<tr>
<td>J2</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>837</td>
<td>2476</td>
</tr>
<tr>
<td>JIL</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>814</td>
<td>2238</td>
</tr>
<tr>
<td>J2L</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>825</td>
<td>2468</td>
</tr>
<tr>
<td>TC</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>1206</td>
<td>6042</td>
</tr>
<tr>
<td>CT</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>533</td>
<td>392</td>
</tr>
<tr>
<td>TCSC</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>457</td>
<td>359</td>
</tr>
<tr>
<td>TCXLSC</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>814</td>
<td>2238</td>
</tr>
<tr>
<td>SPA1</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>814</td>
<td>2238</td>
</tr>
<tr>
<td>SPA2</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>837</td>
<td>2476</td>
</tr>
<tr>
<td>SPS1</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>1100</td>
<td>5799</td>
</tr>
<tr>
<td>SPS2</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>1112</td>
<td>5953</td>
</tr>
<tr>
<td>LEQ</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>1106</td>
<td>6501</td>
</tr>
</tbody>
</table>

Table A.10: symbollicl precision summary.
### Program reader, 115 reachable args, 54 reached uninit (47%), 5C

<table>
<thead>
<tr>
<th>Kind</th>
<th>NI</th>
<th>CO</th>
<th>LI</th>
<th>ST</th>
<th>DI</th>
<th>HY</th>
<th>NUM</th>
<th>NV</th>
<th>⊥</th>
<th>Tot</th>
<th>( \nabla_{CP+SP} )</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Funct</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>22</td>
<td>24</td>
<td>0</td>
<td>48</td>
<td>228</td>
<td>540</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jungle</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>3</td>
<td>48</td>
<td>278</td>
<td></td>
<td>641</td>
</tr>
<tr>
<td>J1</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>3</td>
<td>48</td>
<td>280</td>
<td></td>
<td>641</td>
</tr>
<tr>
<td>J2</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>3</td>
<td>48</td>
<td>338</td>
<td></td>
<td>693</td>
</tr>
<tr>
<td>JIL</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>3</td>
<td>48</td>
<td>281</td>
<td></td>
<td>643</td>
</tr>
<tr>
<td>J2L</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>3</td>
<td>48</td>
<td>335</td>
<td></td>
<td>710</td>
</tr>
<tr>
<td>TC</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>3</td>
<td>48</td>
<td>310</td>
<td></td>
<td>632</td>
</tr>
<tr>
<td>CT</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>28</td>
<td>0</td>
<td>48</td>
<td>625</td>
<td></td>
<td>1912</td>
</tr>
<tr>
<td>TCSC</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>3</td>
<td>48</td>
<td>213</td>
<td></td>
<td>180</td>
</tr>
<tr>
<td>TCXLSC</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>3</td>
<td>48</td>
<td>196</td>
<td></td>
<td>167</td>
</tr>
<tr>
<td>SPA1</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>3</td>
<td>48</td>
<td>280</td>
<td></td>
<td>641</td>
</tr>
<tr>
<td>SPA2</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>3</td>
<td>48</td>
<td>344</td>
<td></td>
<td>722</td>
</tr>
<tr>
<td>SPS1</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>3</td>
<td>48</td>
<td>297</td>
<td></td>
<td>640</td>
</tr>
<tr>
<td>SPS2</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>3</td>
<td>48</td>
<td>349</td>
<td></td>
<td>729</td>
</tr>
<tr>
<td>LEQ</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>28</td>
<td>0</td>
<td>48</td>
<td>409</td>
<td></td>
<td>892</td>
</tr>
</tbody>
</table>

Table A.11: reader precision summary.

### Program bamspec, 512 reachable args, 165 reached uninit (32%), 28C

<table>
<thead>
<tr>
<th>Kind</th>
<th>NI</th>
<th>CO</th>
<th>LI</th>
<th>ST</th>
<th>DI</th>
<th>HY</th>
<th>NUM</th>
<th>NV</th>
<th>⊥</th>
<th>Tot</th>
<th>( \nabla_{CP+SP} )</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Funct</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>28</td>
<td>5</td>
<td>17</td>
<td>137</td>
<td>0</td>
<td>187</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jungle</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>187</td>
<td></td>
<td>720</td>
</tr>
<tr>
<td>J1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>187</td>
<td></td>
<td>768</td>
</tr>
<tr>
<td>J2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>187</td>
<td></td>
<td>804</td>
</tr>
<tr>
<td>JIL</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>187</td>
<td></td>
<td>775</td>
</tr>
<tr>
<td>J2L</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>187</td>
<td></td>
<td>779</td>
</tr>
<tr>
<td>TC</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>187</td>
<td></td>
<td>776</td>
</tr>
<tr>
<td>CT</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>186</td>
<td></td>
<td>1127</td>
</tr>
<tr>
<td>TCSC</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>186</td>
<td></td>
<td>628</td>
</tr>
<tr>
<td>TCXLSC</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>187</td>
<td></td>
<td>629</td>
</tr>
<tr>
<td>SPA1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>187</td>
<td></td>
<td>768</td>
</tr>
<tr>
<td>SPA2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>187</td>
<td></td>
<td>802</td>
</tr>
<tr>
<td>SPS1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>187</td>
<td></td>
<td>780</td>
</tr>
<tr>
<td>SPS2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>187</td>
<td></td>
<td>811</td>
</tr>
<tr>
<td>LEQ</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>28</td>
<td>86</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>187</td>
<td></td>
<td>780</td>
</tr>
</tbody>
</table>

Table A.12: bamspec precision summary.
Correctness of the Topological Clash Widening

In this appendix the correctness proofs in [30] is adapted to fit the definitions of Section 5.4. The correctness requirements for widenings are defined in Definition 2.1.3, page 18.

Termination of the widening operation itself was discussed in Section 5.4.

Soundness, i.e., that $T_{old} \nabla T_{new} \equiv T_{old} \cup T_{new}$, is trivial since widen$_{TC}$ applies the transformations starting with $T_{old} \cup T_{new}$ and each transformation will increase the concretization (or leave it unaffected).

B.1 STATIONARITY

The main complexity lies in showing the stationarity property. The proof is done in a way that is quite close to that used in [30]. The proof in [30], however, appears to be incorrect and the differences needed to correct it will be pointed out below.

In what follows tuples are compared lexicographically, that is,

\[ \langle x_0, \ldots , x_k, \ldots \rangle < \langle y_0, \ldots , y_k, \ldots \rangle \]

if and only if there exists a $k$ such that $x_i = y_i$ for all $i < k$ and $x_k < y_k$, given some underlying comparison operator $\cdot < \cdot$ on the tuple elements.
Theorem B.1.1 (Stationarity of Topological Clash Widening) For any sequence $G_0^*, \ldots, G_i^*, \ldots$ of type graphs the sequence of results from the Topological Clash widening is stationary, that is, the sequence

$G_0 = G_0^*$

$G_{i+1} = G_i \triangledown G_i^* (i \geq 0)$

is not strictly increasing for $\sqsubseteq$. □

This will be proved by proving the following lemma:

Lemma B.1.2

$G_i \sqsubseteq G_{i+1} \implies M(G_i) > M(G_{i+1})$

where $M(G)$ is a well founded measure on type graphs, defined below. □

Since $M(G)$ is well founded its value can only decrease finitely many times. So eventually for some $k$, $M(G_k) > M(G_{k+1})$ will be false and $G_k \sqsubseteq G_{k+1}$ must be false as well by Lemma B.1.2. The soundness of the widening imply that $G_k \sqsubseteq G_{k+1}$ so for $i \geq k$ all $G_i$ will be equivalent, that is the sequence is no longer strictly increasing.

The well founded measure used to prove Lemma B.1.2 is the tuple

$M(G) = \langle P(G), E(G) \rangle$

The components of the tuple are closely tied to the shape requirements enforced by $\text{CR}(G,G')$ in Section 5.4.2. Component $P(G)$ correspond to the condition on the available prlb-sets among the ancestors and component $E(G)$ correspond to the requirement on the depth of the targets of the introducing arcs.

Intuitively, $G_{i+1}$ grows compared to $G_i$ only at nodes where the node in $G_{i+1}$ has a larger prlb-set than the corresponding node in $G_i$, furthermore $G_{i+1}$ is only allowed to grow in this manner if the node in $G_{i+1}$ has no ancestor with larger prlb-set.

The main component of $M(G)$ is $P(G)$ that, intuitively, measures the number of nodes for which it is hard to find such an ancestor. We show that this measure will never increase and that for the cases where it does not strictly decrease the type graphs will have fewer and fewer backward arcs.
targeting nodes far from the root. The latter property is captured by the measure \( E(G) \), formally defined later.

The fundamental property used to construct \( P(G) \) is the potential of a node, denoted \( p(n) \). The intuition is that a larger potential means that it is easier to find an ancestor with a large enough prlb-set to trigger one of the replacement rules. The potential of an any-node is defined to be larger than that of any other kind of node by referring to the size of the, finite set of functors, \( \mathcal{F} \).\(^1\)

\[
p(n) = \begin{cases} 2^{2^{|F|}} + 1 & \text{if } n \text{ is any} \\ \left| \{ s \mid s \subseteq \text{prlb}(n'), n' \in (\{n\} \cup \text{anc}(n)) \} \right| & \text{otherwise} \end{cases}
\]  

(B.1)

Note that the potential of a node is at least as large as that of any of its ancestors, that is, \( p(n) \geq p(n_{\text{anc}}), n_{\text{anc}} \in \text{anc}(n) \).

Define \( P[S,i] \) as the number of nodes \( n \in S \) such that \( p(n) = i \). We define \( P(S) \) as the tuple,

\[
P(S) = \langle P[S,2], \ldots, P[S,2^{|F|}] \rangle
\]  

(B.2)

This means that nodes with small potential leads to large values of \( P(S) \).

Note that this definition of \( P(S) \) does not depend on the number of any-nodes. In [30] the least significant, i.e., rightmost, part of the tuple was \( P[S,2^{|F|} + 1] \), corresponding to the any-nodes, but this will sometimes make \( P(G_i) \) strictly less than \( P(G_{i+1}) \) thus invalidating their proof that the sequence of type graphs is stationary.

We will now show that \( P(G_i) \) will never increase. This will be done by considering various subsets of the nodes in \( G_i \) and \( G_{i+1} \).

There are two main classes of nodes, those that were “matched” against a node from the other graph and those that had no corresponding node in the other graph.

1. \( CC = C(G_i,G_{i+1}) \downarrow 1 \), this is the pairs \((n,n')\) of nodes from the two graphs with a corresponding node in the other graph.

   (a) \( CI = CC \downarrow 1 \), this is the nodes from \( G_i \) that have a corresponding node in \( G_{i+1} \).

\( ^1\)This formulation is to facilitate comparison with [30], for our purposes the value of \( p(\text{any}) \) is irrelevant.
(b) \( C2 = CC \downarrow 2 \), this is the nodes from \( G_{i+1} \) that have a corresponding node in \( G_i \).

2. \( TC = TC(G_{i}, G_{i+1}) \downarrow 1 \), this is the pairs of nodes that make up the topological clashes.

3. \( CO = CC \setminus TC \), this is the pairs of “similar” nodes, that is, the pairs that are not topological clashes.

4. \( NE = \{ n' \in G_{i+1} | \neg \exists n (n, n') \in CC \} \), this is the nodes from \( G_{i+1} \) that have no corresponding node in \( G_i \). It could also be defined as \( G_{i+1} \setminus C2 \).

5. \( OE = \{ n \in G_i | \neg \exists n (n, n') \in CC \} \), this is the symmetric case, that is, the nodes in \( G_i \) that have no corresponding node in \( G_{i+1} \). It could also be defined as \( G_i \setminus CI \).

Note that for \( (n, n'), (m, m') \in CO \) we have \( n = m \iff n' = m' \) whereas this is not true for \( CC \) or \( TC \).

The nodes in \( G_i \) are \( CI \cup OE \), that is \( CO \downarrow 1 \cup TC \downarrow 1 \cup OE \). Similarly \( G_{i+1} \) is \( CO \downarrow 2 \cup TC \downarrow 2 \cup NE \). The property \( P(G_i) \geq P(G_{i+1}) \) will be shown separately for each of these (partially overlapping) sets.

The set \( CO \) consists of the “similar” parts of the two graphs. Pairs \((n, n')\) in \( CO \) have the same \( \mathcal{p} \)-set and depth and so do their corresponding ancestors, thus their potentials will be the same and \( P(CO \downarrow 1) = P(CO \downarrow 2) \).

Pairs \((n, n')\) in \( TC \) can only be of one of the following kinds:

1. Different depth
   
   (a) \( \text{depth}(n) > \text{depth}(n') \)
   
   In this case \( n' \) is in \( CO \downarrow 2 \) and will already have been accounted for in \( P(CO \downarrow 2) \).
   
   (b) \( \text{depth}(n) < \text{depth}(n') \)
   
   By the definition of the widening this implies that \( n' \) is \textbf{any} and it will thus not contribute to \( P(G_{i+1}) \).

In neither of these two cases will \( n' \) contribute further to \( P(G_{i+1}) \).

In both cases \( n \) is either not in \( CO \downarrow 1 \) and thus contributes further to \( P(G_i) \) or \( n \) is in \( CO \downarrow 1 \) and was already accounted for in \( P(CO \downarrow 1) \).

In the latter case, however, for any pair of arcs \((e, e')\) introducing \((n, n')\) it will be the case that \( e \) is a backward arc and if \( e' \) is too then \( n \) will have a larger depth than \( n' \). This motivates the definition of the function \( E(S) \) below.
2. Different prlb-set (and the same depth).
Neither $n$ nor $n'$ will be in $CO \downarrow 1$ or $CO \downarrow 2$ respectively. Thus their potential will not already be accounted for in $CO$.

(a) $n'$ is an any-node.
   It will thus not contribute to $P(G_{i+1})$, however $n$ will not be an any-node and will contribute to $P(G_i)$.

(b) $n'$ is not an any-node. (And, since $n \subseteq n'$, neither is $n$.)
   By the definition of the widening it follows that $n'$ is an or-node\(^2\) and that it does not have an ancestor $n'_{anc}$ such that prlb($n'$) $\subseteq$ prlb($n'_{anc}$). From this also follows that neither does $n$, i.e., $n$ does not have an ancestor $n_{anc}$ such that prlb($n'$) $\subseteq$ prlb($n_{anc}$). Thus $p(n) < p(n')$ and $n$ will contribute more to $P(G_i)$ than what $n'$ does to $P(G_{i+1})$.

This accounts for the nodes in $CO$ and $TC$, that is $C1$ and $C2$ and shows that $P(C1) \geq P(C2)$. The nodes not yet accounted for are all in the sets $NE$ and $OE$. If there is at least one pair $(n, n') \in TC$ such that $n$ and $n'$ have the same depth (and thus different prlb-sets), then by Case 2 above, $P(C1) > P(C2)$ will hold and there will exist a smallest $k$ such that $P[C1, k] > P[C2, k]$.

By definition of the widening any node $n''$ in $NE$ will have an ancestor $n'$ such that $(n, n') \in TC$ where $n$ and $n'$ will have the same depth and thus, by Case 2b above, $k \leq p(n) < p(n') \leq p(n'')$. This implies that every node in $NE$ will have a potential larger than $k$ so,

\[
\begin{align*}
P[C1, k] &> P[C2 \cup NE, k] \\
P[C1, j] &= P[C2 \cup NE, j], j < k
\end{align*}
\]

Since $G_i = C1 \cup OE$ we have $P(G_i) = P(C1 \cup OE) \geq P(C1) > P(C2 \cup NE) = P(G_{i+1})$.

The remaining case to consider is when there is no pair $(n, n') \in TC$ such that $n$ and $n'$ have the same depth. In this case $NE$ is empty and $G_{i+1} = C2$ so that $P(G_i) = P(C1 \cup OE) \geq P(C1) \geq P(C2) = P(G_{i+1})$.

If $P(G_i) = P(G_{i+1})$ then $OE$ too is empty. By the assumption $G_i \subseteq G_{i+1}$ we know that $TC$ is non-empty. Any pair $(n, n') \in TC$ will belong to Case 1 above, $n$ will belong to $CO \downarrow 1$ and would have been introduced via a backward arc $e$. If $n'$ is any the introducing arc $e'$ was not a backward arc and would not contribute to $E(G_{i+1})$ whereas $e$, being a

\(^2\)Unlike the presentation in [30] all our or-nodes have at least two successors.
backward arc, would contribute to $E(G_i)$. If $n'$ is not any then $n, n'$
was introduced by one or more pairs of backward arcs $(e, e')$ and, since
in this case $\text{depth}(n) > \text{depth}(n')$, each such $e$ would contribute more to
$E(G_i)$ than what $e'$ contributes to $E(G_{i+1})$. So, if $P(G_i) = P(G_{i+1})$ then
$E(G_i) > E(G_{i+1})$.

Thus follows that $G_i \sqsubset G_{i+1} \implies M(G_i) > M(G_{i+1})$ and concludes the
proof of Lemma B.1.2 and the stationarity Theorem B.1.1. □

Note that if minimization or $\sqsubseteq$-minimization is applied then $M(G_i)$ will
decrease so if these transformations are done on the result from the widening
then the stationarity property still holds.
BIBLIOGRAPHY

The numbers inside braces indicate on which pages each citation occurred.


32. G. Janssens, M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. Journal of


35. Thomas Lindgren. The Precision of a Simple Structure Analyzer. Technical report 140, Computing Science Department, Uppsala University, April 1997. {69}


44. Leszek Pacholski, Andreas Podelski. Set Constraints — a Pearl in Research on Constraints. invited tutorial; Third International Conference on Constraint Programming, Springer LNCS 1330.


50. H. Sağlam, J. Gallagher. Constrained Regular Approximation of Logic Programs. LOPSTR’97, LNCS 1463, pp. 282-299, 1998 {60, 159, 212}


<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract intersection</td>
<td>11</td>
</tr>
<tr>
<td>abstraction function</td>
<td>9</td>
</tr>
<tr>
<td>anc</td>
<td>52</td>
</tr>
<tr>
<td>anc⁺</td>
<td>52</td>
</tr>
<tr>
<td>ancestor</td>
<td>52</td>
</tr>
<tr>
<td>any-node</td>
<td>54</td>
</tr>
<tr>
<td>args</td>
<td>53</td>
</tr>
<tr>
<td>argument</td>
<td>53</td>
</tr>
<tr>
<td>argument variables</td>
<td>21</td>
</tr>
<tr>
<td>arity</td>
<td>41</td>
</tr>
<tr>
<td>ascending</td>
<td>14</td>
</tr>
<tr>
<td>ascending chain condition</td>
<td>14</td>
</tr>
<tr>
<td>backward arcs</td>
<td>57</td>
</tr>
<tr>
<td>binding</td>
<td>22</td>
</tr>
<tr>
<td>body</td>
<td>21</td>
</tr>
<tr>
<td>bottom</td>
<td>11</td>
</tr>
<tr>
<td>call pattern</td>
<td>65</td>
</tr>
<tr>
<td>chain</td>
<td>14</td>
</tr>
<tr>
<td>child</td>
<td>52</td>
</tr>
<tr>
<td>collecting semantics</td>
<td>12</td>
</tr>
<tr>
<td>compact</td>
<td>55</td>
</tr>
<tr>
<td>comparable</td>
<td>19</td>
</tr>
<tr>
<td>concretization function</td>
<td>9</td>
</tr>
<tr>
<td>conditionally uninitialized</td>
<td>87</td>
</tr>
<tr>
<td>conservative</td>
<td>7</td>
</tr>
<tr>
<td>containment</td>
<td>9</td>
</tr>
<tr>
<td>DAG</td>
<td>52</td>
</tr>
<tr>
<td>definite alias</td>
<td>26</td>
</tr>
<tr>
<td>depth</td>
<td>52</td>
</tr>
<tr>
<td>desc</td>
<td>52</td>
</tr>
<tr>
<td>descendant</td>
<td>52</td>
</tr>
<tr>
<td>descriptive</td>
<td>5</td>
</tr>
<tr>
<td>determinate</td>
<td>30</td>
</tr>
<tr>
<td>deterministic</td>
<td>43</td>
</tr>
<tr>
<td>directed acyclic graph</td>
<td>52</td>
</tr>
<tr>
<td>domain</td>
<td>22</td>
</tr>
<tr>
<td>environment</td>
<td>73</td>
</tr>
<tr>
<td>equivalence</td>
<td>9</td>
</tr>
<tr>
<td>FAIR</td>
<td>79</td>
</tr>
<tr>
<td>FIFO</td>
<td>79</td>
</tr>
<tr>
<td>finite height</td>
<td>14</td>
</tr>
<tr>
<td>FIXED</td>
<td>80</td>
</tr>
<tr>
<td>forward arcs</td>
<td>57</td>
</tr>
<tr>
<td>forward path</td>
<td>57</td>
</tr>
<tr>
<td>function symbols</td>
<td>41</td>
</tr>
<tr>
<td>functor-node</td>
<td>54</td>
</tr>
<tr>
<td>functors</td>
<td>41</td>
</tr>
<tr>
<td>grammar graph</td>
<td>53</td>
</tr>
<tr>
<td>greatest lower bound</td>
<td>11</td>
</tr>
<tr>
<td>head</td>
<td>21</td>
</tr>
<tr>
<td>height</td>
<td>82</td>
</tr>
<tr>
<td>implied</td>
<td>36</td>
</tr>
<tr>
<td>incoming arc</td>
<td>52</td>
</tr>
<tr>
<td>integrated types</td>
<td>51</td>
</tr>
<tr>
<td>introduce</td>
<td>133</td>
</tr>
<tr>
<td>introducing arc</td>
<td>133</td>
</tr>
<tr>
<td>label</td>
<td>54</td>
</tr>
<tr>
<td>LIFO</td>
<td>80</td>
</tr>
<tr>
<td>literals</td>
<td>21</td>
</tr>
<tr>
<td>locally consistent</td>
<td>10</td>
</tr>
</tbody>
</table>

235
LRF, 79
may alias, 26
minimal, 45
minimized, 56
mode, 88
mono-variant, 66
multiple specialization, 66
non-determinate, 30
normal form, 21, 42
occur-check, 63
or-node, 54
outgoing arc, 52
overlap, 30
parent, 52
path, 41, 52
poly-variant, 66
precision order, 9
predecessor, 52
prescriptive, 5
principal labels, 47, 55
principal nodes, 54
productive, 44
program points, 23
program variables, 21
proper ancestor, 52
ranked, 41
reachable, 44
reduced, 44
regular term grammar, 41
regular unary logic program, 60
restriction operator, 17
restrictive, 31
rigid, 51
root, 52
rooted, 140
rooted graph, 52
RUL program, 60
same value component, 31
scheduling policy, 79
selector, 30
set constraints, 6
shortening, 170
simple node, 53
single clausal form, 81
size, 44
sound, 7
spanning tree, 52
start symbol, 42
stationarity, 18
stationary sequence, 18, 19
strictly more precise, 9
strictly more restrictive, 31
substitution, 22
substitution closed, 27
subterm replacement, 30
success pattern, 65
successor, 52
tag, 88
target, 52
top, 9
tuple distributive, 41
TWOPHASE, 80
type, 27
type graph, 57
Type-Jungle, 109
UNFAIR, 80
uninitialized variable, 87
value, 22
widening, 16
worklist, 69