RUC

A Logic Programming Based Approach to Applying Abstract Interpretation to Embedded Software

Henriksen, Kim Steen

A Logic Programming Based Approach to Applying Abstract Interpretation to Embedded Software

Kim S. Henriksen

Computer Science
Roskilde University
P. O. Box 260
DK–4000 Roskilde
Denmark

Telephone:	+45 4674 3839
Telefax:	+45 4674 3072
Internet:	http://www.ruc.dk/dat/
E-mail:	datalogi@ruc.dk

ISSN 0109–9779

Research reports are available electronically from:

http://www.ruc.dk/dat/

A Logic Programming Based Approach to Applying Abstract Interpretation to Embedded Software

Kim S. Henriksen

October 15, 2007

Abstract

Abstract interpretation is a general framework for static program analysis. In recent years this framework has been used outside academia for verification of embedded and real-time systems. Airbus and the European Space Agency are examples of organisations that have successfully adapted this analysis framework for verification of critical components.

Logic programming is a programming paradigm with a sound mathematical foundation. One of its characteristics is the separation of logic (the meaning of a program) and control (how it is executed); hence logic programming, and in particular its extension with constraints, is a language comparatively well suited for program analysis.

In this thesis logic programming is used to analyse software developed for embedded systems. The particular embedded system is modeled as an emulator written as a constraint logic program. The emulator is specialised with respect to some object program in order to obtain a constraint logic program isomorphic to this object program. Applying abstract interpretation based analysers to the specialised emulator will provide analysis results that can directly be related back to the object program due to the isomorphism maintained between the object program and the specialised emulator.

Two abstract interpretation based analysers for logic programs have been developed. The first is a convex polyhedron analyser for constraint logic programs implementing a set of widening techniques for improved precision of the analysis. The second analyser is a type analysis tool for logic programs that automatically derives a pre-interpretation from a regular type definition.

Additionallly, a framework for using a restricted form of logic programming, namely Datalog, to express and check program properties is described.

At the end of the thesis it is shown how instrumenting the semantics of the emulator can be used to obtain, for instance, a fully automatic Worst Case Execution Time analysis by applying the convex polyhedron analyser to the instrumented and specialised emulator.

The tools developed in this thesis have all been made available online for demonstration.

Resume (in danish)

Abstract interpretation er en overordnet referenceramme for statisk programanalyse. I de seneste år har abstract interpretation vundet udbredelse uden for det akademiske miljø, hvor det er blevet anvendt til verifikation af indlejrede og tidskritiske systemer.

Logikprogrammering er et programmeringsparadigme der har et solidt matematisk fundament. Et af dets karakteristika er adskillelsen af logik (betydningen af et program) og kontrol (hvordan programmet udføres), hvilket gør logikprogrammering til et meget anvendeligt sprog hvad angår programanalyse.

I denne afhandling bliver logikprogrammering anvendt til at analysere programmer udviklet til indlejrede systemer. Et givet indlejret system modelleres som en emulator skrevet i en variant af logikprogrammering kaldet *constraint logic programming* (CLP). Emulatoren specialiseres med hensyn til et givet program, hvilket resulterer i et nyt program i skrevet i sproget CLP der samtidig er isomorft med programmet skrevet til det indlejrede system. Anvendes abstract interpretation baserede analysatorer på det specialiserede program, kan resultater fra denne analyse direkte overføres til den indlejrede program, da dette program og den specialiserede emulator holdes isomorft.

To abstract interpretation baserede analysatorer for logikprogrammering er udviklet i denne afhandling. Den første er en convex polyhedron analysator for CLP programmer, der implementerer et sæt af *widening*-teknikker der giver forbedret præcision af analysen. Den anden analysator er en type-analysator for logikprogrammering, der automatisk udleder en *pre-interpretation* fra et sæt af regulære type-definitioner.

Sidst i afhandlingen vises det hvorledes en udvidelse af emulatorens semantik kan benyttes til at opnå, for eksempel, en fuldautomatisk *Worst Case Execution Time* analyse, ved at anvende det convex polyhedron baserede analyseværktøj på den udvidede og specialiserede emulator.

Alle analyseværktøjer udviklet i forbindelse med denne afhandling er gjort tilgængelige online.

Acknowledgements

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Bugs in software can turn out to be costly. A well known example of this is the bug in the control software used in Ariane 5's first test flight in 1996[1]. The Ariane 5 rocket is one of the European Space Agency's launch vehicles. On its first flight a software bug cause the rocket to explode just 40 seconds after lift-off. The value of the rocket and its cargo was estimated to be around $500 million. An investigation of the incident led to the discovery of a bug in a piece of code where a floating point number was converted to a 16 bit signed integer. The code did not ensure that the floating point number could be correctly represented as a 16 bit integer. This incident lead to an increased support for research in the field of ensuring reliability of critical systems.

1.1 Program analysis

The overall goal of this thesis is to present an approach to applying program analyses to software developed for small low level hardware such as microcontrollers used in pervasive computing. These devices are becoming embedded in more and more products. Traditional software development generally tends to produce general purpose and bloated software built from pre-written libraries. Software developers for pervasive software face different challenges. Typically these devices are resource constrained; this could for example be limited memory capacity. Battery lifetime of the device on which the program will be executed is an example of a resource constraint that would typically not be considered in traditional software development. Additionally, factors such as reliability, development cost and development time can be more constrained for pervasive software. If such a piece of software will be included in an everyday product such as a clothing item, reliability can become an important factor. Updating faulty software might not be possible,

---

[1] http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html

since production cost might favor a non-reprogrammable microcontroller. If the item the pervasive system will be included in is a low cost item, allocating large sums of man hours to develop and verify the software might prove too costly, both in terms of money and time.

There is another distinct difference between traditional software development and software development for pervasive system. Traditional, general purpose software development has standardised on a few languages such as C/C++ and Java, and some domain specific languages such as PHP and Perl for web based development. On the other hand, in the world of microcontrollers each manufacturer has its own assembly language and they typically produces a large variety of microcontrollers having differently sized memory, different machine word size - typically 8, 16 or 32 bits - and different instruction sets; some microcontrollers may include a multiplication instruction or even specialised instructions for Digital Signal Processing, others may only provide addition, subtraction and bit manipulation. Program analysers and other techniques for program analysis need only be developed for one language, e.g. Java, to be useful for a large number of developers. On the other hand software development tools and program analysers, to the extent they exist for microcontrollers, tend to be product specific. For most series of microcontrollers there exists a C-compiler, however typically only a subset of Ansi-C is supported. Furthermore on the smaller microcontrollers the available resources may be so limited that only assembly languages are an option. Where C is a typed language that to some extent can assist the programmer in detecting type related bugs, assembly languages offer no such techniques.

Assembly programmers can also take advantage of simple tricks to speed up software development such as reusing old code. Reusable code tends to be over-general and resource inefficient. Where code reuse may increase productivity of the developers, it can lead to a waste of resources on the target system. If the program environment is resource constrained, automatic tools such dead code analysis and register remapping can be helpful for the programmers. Again such tools would be specific to the particular flavor of assembly language used.

Development costs of analysis tools

While the European Space Agency may have the resources to develop analysis tools tailored to their use, as happened in the Ariane 5 incident, smaller organisations, e.g. developing pervasive systems, may not have the resources to start from scratch and develop analysers for verifying their software. In that case some general framework for applying analyses to a range of microcontrollers would be appropriate.
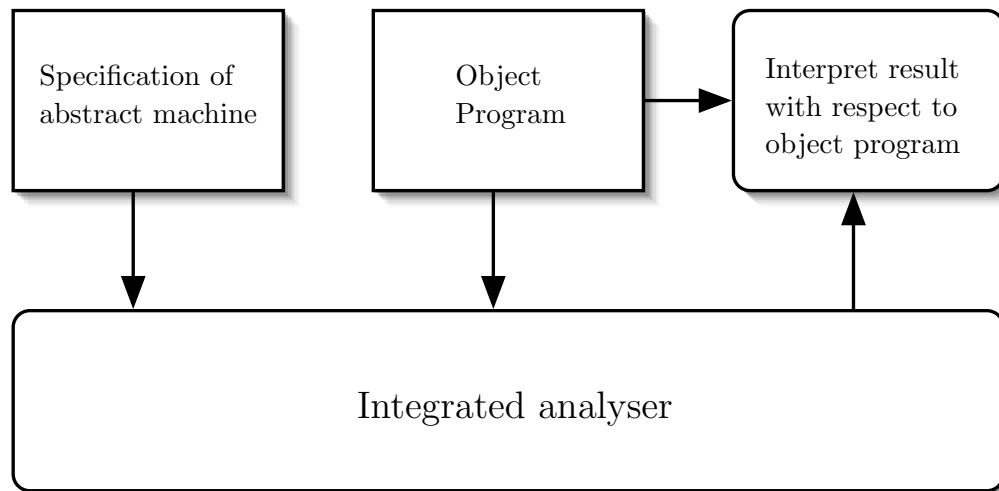
Figure 1.1: Overview of the analysis framework

Analysis using declarative languages

Most software, whether it is pervasive software or software for personal computers, games etc., is developed in imperative programming languages; in fact all of the languages mentioned so far are imperative languages. Declarative languages however are more suited for program analysis, specialisation and transformation.

The framework described here for analysing low level programs uses program specialisation and transformation to create a high level declarative language equivalent of the low level assembly language program. This specialised program can then be analysed using tools written specifically for the declarative language.

Ideally, once the framework is established, the analyser would require only a specification of the target machine and an object program to analyse, to produce analysis results that can be interpreted with respect to the object program. Figure 1.1 illustrates this.

The analyser would be constructed using well known program analysis and specialisation techniques. The following mentioned techniques will be used to construct the analyser:

Abstract Interpretation

Static Analysis is a technique used for acquiring information about a program's run-time behavior without actually executing it. The information obtained in

3

static analyses can either be used by other programming tools, such as a compiler, or the programmer can use the results to discover properties of the analysed program.

A static analysis is generally not complete, since some program properties are undecidable. The result of a static analysis is therefore a sound approximation of the correct result. An approximation is an inexact representation of the correct solution. A sound approximation contains all the correct results, in addition to the (many) incorrect results. The sound approximation is sometimes also called a *safe* approximation. Given a program, call it P, and suppose only the correct answers to an analysis are in a set S, and given a set S' that is a safe approximation of S, that is $S \subseteq S'$, then any property that does not hold for all elements of S', does not hold for all elements of S, and can therefore not be established by P. Hence program behavior that can be excluded by a safe approximation, is excluded from the (possibly uncomputable) correct behavior of P.

Abstract interpretation is a generic and established framework for static program analysis. Normally a program is evaluated over a concrete domain, where a typical concrete domain is a collection of machine words or floating point numbers. In abstract interpretation a program is evaluated (or interpreted) over an abstract domain, where the abstract domain will have better computational behavior than the concrete domain. The abstract domain can contain symbolic values, such as types found in programming languages, or possibly geometric shapes such as polyhedra. Results obtained for the abstract domain will be safe approximations of the results that would have been obtained evaluating the program over the (possibly uncomputable) concrete domain.

The Ariane 5 incident mentioned in the beginning led to one of the first examples of a large scale static code analysis using abstract interpretation [49].

Meta-programming

A meta-program is a program that has other programs as inputs. Using meta-programs in program analysis is an old concept. Particularly in logic programming, meta-programs have been used to analyse other logic programs. Using logic programming based meta-programs to study other languages than logic programming has also been studied in the past, but not to the same extent.

Program Specialisation and Transformation

Partial evaluation is a well established source code to source code program transformation technique. Partial evaluation is often described as a technique used to produce faster versions of a general purpose program. It is however also well known that applying partial evaluation to interpreters will result in compiler generators.

In other words partial evaluation is one method that can be used to translate an object program written in one language to an equivalent program written in a different language, by means of an interpreter.

1.2   Thesis Overview

Part I This part of the thesis will focus on analysing and specialising logic programs. Chapter 2 will introduce different flavors of logic programming that will later be used as the target language of analyses, and analysis of logic programs based on pre-interpretations are introduced. Chapter 3 will introduce abstract interpretation as a framework for program analysis. In Chapter 4 a classical abstract domain is described, namely the Convex Polyhedral Domain. An analyser for constraint logic programming is constructed, that will later form part of the analyser for pervasive software. Chapter 5 describes how pre-interpretations suitable for analysis of logic program can automatically be constructed from regular type definitions; a tool for both deriving these pre-interpretations and analysis based on these pre-interpretations is described.

Part II In Chapter 6 a specific microcontroller used in pervasive applications is chosen as a case study. An emulator for this case study microcontroller is constructed. Existing general logic programming specialisation and analysis tools, such as the constructed convex polyhedron analyser for constraint logic programming are then applied to this emulator and a given object program to be analysed. The results obtained are linked back to the object program. The process is fully automatic once the emulator and analysis tools are contructed, and a Web Interface is described, that demonstrates that using logic programming as a meta-language for program analysis can be kept transparent to the user of the tools.

1.3   Thesis Contributions

The work in this thesis was partly funded by the EU IST FET project *Advanced Specialization and Analysis for Pervasive Computing*[2] (ASAP) and Roskilde University. The ASAP project, the partners involved in the project and in particular my supervisor John Gallagher, have influenced the topics, methods and tools used in this thesis.

---

[2]`http://clip.dia.fi.upm.es/Projects/ASAP/`

Convex Polyhedron Analyser for CLP. A Convex Polyhedron Analyser for CLP programs has been implemented following a well known method for constructing bottom-up analysers for logic programs. The polyhedra operations are implemented using a freely available programming library called PPL[3]. The analyser implements a combination of different techniques for improving the precision of the analysis such as delayed widening, widen up to and a simple narrowing procedure. The resulting analyser is available online and is the first publicly-available tool, to our knowledge, combining state-of-the-art widening and narrowing procedures allowing experimentation with novel ways of combining these techniques.

Pre-interpretations derived from regular type definitions. A method for automatically deriving pre-interpretations from regular type definitions is described. It is based on existing algorithms from the literature on Finite Tree Automata theory. An optimised algorithm is described and implemented. The technique developed here has been integrated in a fully automatic binding time analyser for Prolog programs. A type analysis tool based on this method for deriving pre-interpretations and a few additional existing type inference techniques have also been made available online.

Property Programming. Using recent results for solving Datalog programs using Binary Decision Diagrams, a method for specifying Control Flow and Data Flow properties of abstract machines as Datalog rules is described.

Liveness analysis from redundant argument filtering. The connection between an existing method for transforming logic programs to remove redundant predicate arguments and liveness analysis of specialised abstract machines, is explained.

Analysing microcontrollers using CLP. A framework for applying CLP program analysers to low-level hardware such as microcontrollers is described. It is based on well founded techniques such as partial evaluation of interpreters and abstract interpretation. It is shown how a parametric Worst Case Execution Time (WCET) analysis can automatically be derived for the analysed program.

The following publications are a result of collaboration with my supervisor and contributed to this thesis.

[1] Kim S. Henriksen and John P. Gallagher. Analysis and Specialisation of a PIC processor. In *Proceedings of the 2004 IEEE Conference on Systems, Man and Cybernetics*, The Hague, Netherlands, October 10-13 2004.

---

[3]http://www.cs.unipr.it/ppl/

[2] Kim S. Henriksen and John P. Gallagher. Abstract Interpretation of PIC programs through Logic Programming. In *SCAM'06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 184-196, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Kim S. Henriksen and John P. Gallagher. A Web-based Tool Combining Different Type Analyses. In W. Vanhoof and S. Muñoz-Hernandez, editors, *WLPE-06: Workshop on Logic-Based methods in Programming Environments: ICLP-06 Workshop*, 2006.

[4] John P. Gallagher and Kim S. Henriksen. Abstract Domains based on Regular Types. In V. Lifschitz and B. Demoen, editors, *Proceedings of the International Conference on Logic Programming (ICLP'2004)*, volume 3132 of *Springer-Verlag Lecture Notes in Computer Science*, pages 27-42, 2004.

[5] John P. Gallagher, Kim S. Henriksen and G. Banda. Techniques for scaling up analyses based on Pre-Interpretations. In M. Gabbrielli and G. Gupta, editors, *Proceedings of the 21st International Conference on Logic Programming, ICLP'2005*, volume 3668 of *Springer-Verlag Lecture Notes in Computer Science*, pages 280-296, 2005.

[6] S.-J. Craig, J. P. Gallagher, M. Leuschel and K. S. Henriksen. Fully Automatic Binding Time Analysis for Prolog. In Sandro Etalle, editors, *Pre-Proceedings, 14th International Workshop on Logic-Based Program Synthesis and Transformation, LOPSTR 2004*, Verona, August, 2004, pages 61-70.

[7] Kim S. Henriksen, G. Banda and John P. Gallagher. Experiments with a Convex Polyhedral Analysis Tool for Logic Programs. In P. Hill and W. Vanhoof, editors, *WLPE-07: Workshop on Logic-Based methods in Programming Environments: ICLP-07 Workshop*, 2007.

Part I

Analysis and Specialisation of Logic Programs

Chapter 2

Logic Programming

Throughout this thesis logic programming in one flavor or the other will be used. This chapter should serve as an introduction to the terminology used when discussing logic programming.

Mathematical logic as a programming language emerged in the 1970s, mainly in the fields of automatic theorem proving and artificial intelligence. Logic programming belongs to the group of *declarative languages*. In the declarative languages the programmer specifies *what* is supposed to be computed and not *how* it should be computed. In logic programming languages, *logic* and *control* are separated. The logic part defines what should be computed, where the control part defines how it should be computed. This means a logic program can be given a model-theoretic based semantics independent of what method would be used to solve (or prove) the logic program.

Some logic programs may be written, or automatically generated using program specialisation, that are in fact not meant to be executed - but only analysed. In other words, their model theoretic semantics is more relevant than their procedural semantics. For our purpose, logic programming will mainly serve as a language for analysis of programs.

The typical incarnation of a "solver" for logic programming, is a Prolog (PROgramming in LOGic) run-time system with an interpreter and/or compiler. Pure Prolog is a purely logical language implementing a formal logic system of first order Horn clauses. Prolog is an extension of pure Prolog including features such as negation and arithmetics.

There are extensions of, as well as restrictions of, logic programming languages. Constraint Logic Programming (CLP) is an extension of logic programming, where relations in the form of constraints can be added to the logic program. This increases the expressiveness of the language compared to pure logic programming. The first formal framework for CLP was introduced by Jaffar and Lassez in their paper published in 1987 [91]. A constraint on some variables in a program can

be viewed as a formula expressing which conditions must hold for these variables (for example $X + Y > 0$ or $X = Y + 1$) . Prolog is a CLP language where the constraints are embedded in the program as formulas over terms. The constraints in CLP are evaluated over some fixed domain. If a particular domain is used it is normally stated along with the acronym; for instance for CLP over real numbers we would write CLP($\mathcal{R}$), for rational numbers CLP($\mathcal{Q}$), and for natural numbers CLP($\mathcal{N}$).

There is a restriction of logic programming called Datalog which will also be referred to throughout this thesis. Datalog can be viewed as a subset of Prolog. It is named after is origin in database theory, particularly deductive databases. Datalog will be defined in the last section of this chapter.

Chapter Overview

Some notation and terminology used later in this thesis is established in this chapter.

- Section 2.1 defines first order logic languages.

- Section 2.2.1 describes a bottom-up semantic framework based on pre-interpretations[1].

- Section 2.3 describes how the (goal independent) bottom-up based framework can be used to simulate (goal dependent) top-down semantic frameworks.

- Section 2.4 introduces Datalog and stratifiable programs.

2.1 Definitions

In this section first order logic is defined. Definitions are mainly based on [112] and [102] but syntax may differ.

Definition 1 (Alphabet). *An alphabet consists of*

- *a possibly empty set of function symbols*

- *predicate symbols*

- *variables*

- *connectives* $(\neg, \wedge, \vee, \leftarrow, \leftrightarrow)$

---

[1]Chapter 5 will described how pre-interpretations can be automatically constructed from Finite Tree Automata.

- *quantifiers ($\forall$, $\exists$)*

- *punctuation symbols, such as brackets and comma*

All functions and predicate symbols have an arity. *The arity is a natural number, possibly zero, indicating the number of arguments that the function takes. A function with arity of zero is also called a constant. A predicate with an arity of zero is called a proposition.*

For readability purposes, the following syntactical rules are followed to the extent it is possible:

- Variables are denoted by upper case letters selected from the end of the alphabet, for example X, Y, Z.

- Constants are denoted by lower case letters selected from the beginning of the alphabet, for example a, b, c.

- Function symbols are denoted by lower case letters selected from the letter f and the following letters, for example f, g, h.

- Predicates are denoted by lower case letters selected from the group of letters beginning with p and forward, for example p, q, r.

Specifically for logic programs the following rules are applied

- quantifiers are omitted.

- the conjunction operator '$\wedge$' is substituted by a comma.

- lists are written using either Prolog notation, such as $[H|T]$ and $[\ ]$, where H is the head element of the list and T is the tail and $[\ ]$ is the empty list, or if appropriate, the distinguished functor $cons(H, T)$ and *nil* will be used in order to tell apart lists in different domains.

- we adopt the notion of *don't care* from Prolog. A *don't care* variable in Prolog is an anonymous variable and is denoted '_'. The value of a don't care variable can be any possible value that can occur in a given context.

If the particular denotation of a predicates arguments is irrelevant, the arity is often written along with its name instead of its arguments, for instance the predicate $p(X, Y)$ can be written as $p/2$.

Definition 2 (Terms). *The set of terms over some alphabet is inductively defined as:*

- *a variable is a term.*

- *a constant is a term.*

- *a function symbol f of arity $n > 0$ applied to the sequence $t_1, \ldots, t_n$ of n terms, $f(t_1, \ldots, t_n)$ is also a term.*

Definition 3 (Atom). *The set of atoms over some alphabet is defined as:*

- *a proposition is an atom.*

- *a predicate symbol p of arity $n > 0$ applied to the sequence $t_1, \ldots, t_n$ of n terms, denoted by $p(t_1, \ldots, t_n)$ is an atom.*

Definition 4 (Ground). *A term is said to be ground if it contains no variables and similarly an atom is ground if it contains no variables.*

Definition 5 (Formula). *A formula over some alphabet is inductively defined as:*

- *an atom is a formula.*

- *if A and B are formulas then $\neg A$, $A \wedge B$, $A \vee B$, $A \leftarrow B$, and $A \leftrightarrow B$ are also formulas.*

- *let X be a variable and F a formula, then $\forall X \ F$ and $\exists X \ F$ are also formulas.*

Definition 6 (Literal). *If A is an atom then the formulas A and $\neg A$ are called literals. A is called a* positive *literal and $\neg A$ called a* negative *literal.*

Definition 7 (Clause). *A clause is a formula of the form $\forall (H_1 \vee \ldots \vee H_m \leftarrow B_1 \wedge \ldots \wedge B_n)$ where $m \geq 0$, $n \geq 0$ and $H_1, \ldots, H_m, B_1, \ldots, B_n$ are all literals.*

- *the left hand side of the formula, $H_1 \vee \ldots \vee H_m$ is called the head of the clause, and the right hand side is called the body of the clause.*

- *a* Horn clause *has at most one positive literal in the head of the clause.*

- *a* normal program clause *has an atom as a single head element H_1.*

- *a* definite program clause *is a normal program clause where $B_1, \ldots, B_n$ are all atoms.*

- *a* fact *is a program clause with an empty body; this is sometimes referred to as a unit clause.*

- *and a* goal *is a clause with an empty head and a non-empty body.*

A clause with only atoms containing no variables is also called a *ground instance.*

Definition 8 (Program)**.** *A normal program is a set of normal program clauses. A definite program is a set of definite program clauses.*

The following notation rules for sets of symbols, terms over a set of symbols etc., will be used in the remaining part of this chapter:

- Σ denotes the set of function symbols including constants.

- Π denotes the set of predicate symbols including propositions.

- *Var* denotes the set of variables.

- Term denotes the set terms over an alphabet. Term$_\Sigma$ denotes the set of terms over constants and functions symbols in a given alphabet, Σ.

- Atom denotes the set of atoms constructed from Π and Term.

Definition 9 (Pre-interpretation)**.** *Let L be a first order language. A pre-interpretation of L consists of*

1. *a non-empty domain of interpretation D.*

2. *an assignment of an element in D for all constants in L.*

3. *an assignment of an n-ary function $D^n \rightarrow D$ to each n-ary function symbol in Σ (n > 0).*

More informally this can be described as mapping all constants (the 0-ary functions) to an element in the supplied domain D, and set of mappings for all functions $f/n \mapsto \hat{f}^n : D_1 \times \ldots \times D_n \rightarrow D$.

There exists a particular kind of pre-interpretation called the *Herbrand preinterpretation,* which will be used later in this chapter.

Definition 10 (Herbrand Universe and Base)**.** *The* Herbrand Universe *for the language L is the set of all ground terms that can be formed from the function*

15

symbols including constants in Σ. The Herbrand Universe is denoted Term_Σ. This requires that Σ contains at least one constant.

The Herbrand Base, Atom_H, of the language L, is the set of all ground atoms that can be formed using predicate symbols from Π where its arguments are in the Herbrand Universe Term_Σ.

Definition 11 (Herbrand pre-interpretation). *The Herbrand pre-interpretation for a language L is the pre-interpretation defined by*

1. *The domain of the pre-interpretation is Term_Σ*

2. *Constants in Σ are mapped to themselves in Term_Σ*

3. *For all function symbols f/n and all terms $t_1, \ldots, t_n \in \mathsf{Term}_\Sigma$, f applied to $t_1, \ldots, t_n$ is mapped to $f(t_1, \ldots, t_n)$.*

Definition 12 (Term assignment). *Let J be a pre-interpretation of the language L with a domain D, and let V be a mapping assigning each variable in L to an element of D. A term assignment $T_J^V(t)$ is defined for each term t as follows:*

1. $T_J^V(x) = V(x)$ *for each variable x.*

2. $T_J^V(f(t_1, \ldots, t_n)) = f'(T_J^V(t_1), \ldots, T_J^V(t_n))$, $(n \geq 0)$ *for each non-variable term $f(t_1, \ldots, t_n)$, where f' is the function assigned by J to f.*

Example 1. *For a language with a set of function symbols $\{nil, cons(\ ,\ )\}$, a given domain $D = \{list, nonlist\}$, the pre-interpretation could be*

$$
\begin{aligned}
nil \quad &= \quad list \\
cons(\ ,\ ) \quad &= \quad \left\{
\begin{array}{rcl}
cons(nonlist, list) &\to& list \\
cons(list, nonlist) &\to& nonlist \\
cons(list, list) &\to& list \\
cons(nonlist, nonlist) &\to& nonlist
\end{array}
\right\}
\end{aligned}
$$

For the example term $t = cons(X, cons(X, nil))$ and assuming the mapping $V = \{X \mapsto nonlist\}$ the term assignment would be

$$T_J^V(t) = [nonlist|[nonlist|nil]] = [nonlist|list] \to list = list \quad \square$$

Assigning domain elements from D to ground terms is sometimes referred to as giving terms a *denotation*. The term in the example above, would have the denotation *list*.

Definition 13 (Substitution)**.** *A binding X_i/t_i consists of a variable X_i and a term t_i with $t_i \neq X_i$. A substitution θ is a finite set of bindings, $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$, where $X_1, \ldots, X_n$ are distinct.*

Definition 14 (Interpretation)**.** *An interpretation, denoted I, of a first order language L consist of*

1. *a pre-interpretation J over domain D*

2. *for each n-ary predicate symbol in Π, a mapping, i.e. an n-ary relation*

$$D^n \rightarrow \{true, false\}$$

For an interpretation I of a language L with a variable assignment V, assigning an element in the domain $\mathcal{D}$ of I to each variable in L, the truth values (**true**,**false**) are assigned based on the following criteria:

- if the formula is an atom $p(t_1, \ldots, t_n)$ then the truth value is obtained by calculating the value of $p'(t'_1, \ldots, t'_n)$ where p' is the mapping assigned to p by I and $t'_1, \ldots, t'_n$ are the term assignments $T_I^V(t_1), \ldots, T_I^V(t_n)$.

- if the formula has the form $\neg Q, Q \wedge R, Q \vee R$, $Q \rightarrow R$ or $Q \leftrightarrow R$ then the truth value is given by the following table:

Q	R	$\neg Q$	$Q \wedge R$	$Q \vee R$	$Q \rightarrow R$	$Q \leftrightarrow R$
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

- Formulas of the form $\exists x\, Q$ are true if $\exists d \in \mathcal{D}$ such that Q is assigned the truth value **true** with respect to I and the substitution $V(x/d)$, where x is substituted for d in V. Otherwise, Q is false.

- Formulas of the form $\forall x\, Q$ are true if $\forall d \in \mathcal{D}$ then Q is assigned the truth value **true** with respect to I and the substitution $V(x/d)$, where x is substituted for d in V. Otherwise, Q is false.

Definition 15 (Model)**.** *A model of a formula (over a domain D) is an interpretation in which the formula has the value* **true** *assigned to it.*

The concept of a model of a formula can be extended to *sets* of formulas. A model of a set S of formulas is an interpretation which is a model for all formulas in S. Two formulas are logically equivalent if they have the same set of models. A formula Q is a logical consequence of a set S of formulas, if Q is assigned the value **true** in all models of S; this is denoted $S \models Q$.

A model of P will be denoted $\mathsf{M}(P)$ in this thesis. M may be super- or subscripted to indicate over what domain the model is computed.

Definition 16 (Herbrand model). *Any interpretation based on the Herbrand pre-interpretation of a language L, is a* Herbrand interpretation *of L.*

- *A* Herbrand model *of a definite program P of the language L is any Herbrand interpretation of L that is also a model of P.*

- *A Herbrand model $H \subseteq \mathsf{Atom}_H$ for a program P is a* least Herbrand model *if no other $H' \subset H$ is also a Herbrand model of P.*

The least Herbrand model captures the meaning of a definite program. It contains all the atomic logical consequences of the program. A formula that is true in the least Herbrand model is true in all Herbrand models.

2.2 Semantics of Definite Logic Programs

The semantics of a program is the meaning assigned to this program. For a definite logic program the semantics is equivalent to the minimal Herbrand model for this program. This is called the *declarative* semantics. It defines the set of atomic logical consequences of the program. The declarative semantics can also be used indirectly to capture the computational behavior of a program and a goal; i.e. whether a goal succeeds or fails and what answer substitutions it returns. In this thesis we use the declarative semantics as the basis for goal-independent (bottom-up) and goal-dependent (top-down) analysis.

2.2.1 Bottom-up semantic frameworks

T_P-semantics

The least Herbrand model can be obtained as the least fix point of the function T_P [149] defined next. The theoretical foundation of this semantics is based on among other things, complete lattices and monotone functions over complete lattices. The definition of these is postponed to the next chapter on abstract interpretation.

The T_P semantics can be considered a standard bottom-up semantics where the meaning of the program P is defined to be the set of ground atoms implied by the program. Variations of this semantics are frequently found in bottom-up frameworks [18, 3, 40]. A transfer function T_P from interpretation to interpretation is defined. This transfer function is sometimes also called the immediate consequence operator. It is defined as

$$T_P : 2^{\mathsf{Atom}_H} \rightarrow 2^{\mathsf{Atom}_H}$$

$$T_P(I) = \left\{ \left. H \; \right| \; \begin{array}{l} H \leftarrow B_1, \ldots, B_n \in ground(P) \\ \forall i \in [1, \ldots, n] : B_i \in I \end{array} \right\}$$

$$\mathsf{M}\llbracket P \rrbracket = \mathsf{lfp}(T_P)$$

where $ground(P)$ is the set of ground instances of clauses from P, and where lfp denotes the least *fix point*. A *fix point* of a function is a point that is mapped to itself by the function. Algorithm 1 outlines this. A fix point for this function is calculated by initially applying the function to the empty interpretation, and then iteratively applying it to itself until a fix point is reached. Another name for this kind of semantics is *fix point semantics*.

Algorithm 1 Fix point iterations for transfer function T_P

initialise:
  $i = 0; \quad I_0 = \emptyset$
repeat
  $I_{i+1} = T_P(I_i)$
  $i = i + 1$
until $I_i = I_{i-1}$

The Herbrand pre-interpretation and least Herbrand model are essential to many bottom-up analysis frameworks. Other pre-interpretations can however also be useful in the context of abstraction.

Semantics parameterized by a pre-interpretation

We present first a few definitions from [112].

Definition 17 (Domain atom)**.** *Let J be a pre-interpretation of a language L, with domain D, and let p be an n-ary predicate symbol from L. Then a domain atom for J is any structure $p(d_1, \ldots, d_n)$ where $d_i \in D$, $1 \leq i \leq n$.*

Let A be the atom $p(t_1, \ldots, t_n)$. Then the *domain instance* of A with respect to J and V is the domain atom $p(T_J^V(t_1), \ldots, T_J^V(t_n))$. Denote by $[A]_J$ the set of all domain instances of A with respect to J and some V.

In the interpretation given in Example 1 and for an atom $p(nil)$, then $[p(nil)]_J$ would be $\{p(list)\}$, and for the atom $p(cons(X,Y))$ the result would be

$$[p(cons(X,Y))]_J = \{p(list), p(nonlist)\}$$

The definition of domain instance extends naturally to formulas. In particular, let C be a clause. Denote by $[C]_J$ the set of all domain instances of the clause with respect to J.

The core bottom-up declarative semantics is parameterised by a pre-interpretation of the language of the program.

Definition 18 (Core bottom-up semantics function T_P^J). *Let P be a definite program, and J a pre-interpretation of the language of P. Let $Atom_J$ be the set of domain atoms with respect to J.*

$$T_P^J : 2^{Atom_J} \rightarrow 2^{Atom_J}$$

$$T_P^J(I) = \left\{ \left. A' \; \right| \; \begin{array}{l} A \leftarrow B_1, \ldots, B_n \in P \\ A' \leftarrow B'_1, \ldots, B'_n \in [A \leftarrow B_1, \ldots, B_n]_J \\ \{B'_1, \ldots, B'_n\} \subseteq I \end{array} \right\}$$

$$\mathsf{M}^J[\![P]\!] = \mathsf{lfp}(T_P^J)$$

$\mathsf{M}^J[\![P]\!]$ *is the minimal model of P with pre-interpretation J.*

Example 2. *Continuing the pre-interpretation given in Example 1, let*

$$P = \left\{ \begin{array}{lll} p(nil) & \leftarrow & \\ p(cons(X, Xs)) & \leftarrow & p(Xs) \end{array} \right\}$$

and let the interpretation be $I = \emptyset$. Then $T_P^J(I) = \{p(list)\}$, since $p(nil) \leftarrow$ is a fact (has an empty body) and must be true in any interpretation and the domain instances $[p(nil) \leftarrow]_J$ is $\{p(list) \leftarrow\}$. Re-iterating will produce the same result and a fix-point is reached, so the minimal model of P with respect to the pre-interpretation J is $\{p(list)\}$. $\square$

The concrete semantics will be defined next. It is based on the Herbrand pre-interpretation described earlier.

2.2.2 Concrete Semantics

The concrete semantics is a very precise semantics that closely describes the set of all results that can possibly be obtained from a program. Here, the concrete semantics will be based on the Herbrand pre-interpretation. This is obtained by taking J to be the Herbrand pre-interpretation, which we call H. Thus Atom_H is the Herbrand base of (the language of) P and $\mathsf{M}^H[\![P]\!]$ is the least Herbrand model of P.

The least Herbrand model consists of ground atoms. In order to capture information about the occurrence of variables, we extend the signature with an infinite set of extra constants $\mathcal{V} = \{v_0, v_1, v_2, \ldots\}$. The Herbrand pre-interpretation over the extended language is called HV. The model $\mathsf{M}^{HV}[\![P]\!]$ is our concrete semantics.

The elements of $\mathcal{V}$ do not occur in the program or goals, but can appear in atoms in the minimal model $\mathsf{M}^{HV}[\![P]\!]$. Let $\mathcal{C}(P)$ be the set of all atomic logical consequences of the program P, known as the Clark semantics [33]; that is, $\mathcal{C} = \{A \mid P \models \forall A\}$, where A is an atom. Then $\mathsf{M}^{HV}[\![P]\!]$ is isomorphic to $\mathcal{C}(P)$. More precisely, let Ω be some fixed bijective mapping from $\mathcal{V}$ to the variables in L. Let A be an atom; denote by $\Omega(A)$ the result of replacing any constant v_j in A by $\Omega(v_j)$. Then $A \in \mathsf{M}^{HV}[\![P]\!]$ iff $P \models \forall(\Omega(A))$. By taking the Clark semantics as our concrete semantics, we can construct abstractions capturing the occurrence of variables.

Example 3. *Take a program containing a head-only variable, say $P = \{p(X) \leftarrow true, p(a) \leftarrow true\}$, and given the set of extra constants $\mathcal{V} = \{v_0, v_1, v_2, \ldots\}$, then the concrete semantics is $\mathsf{M}^{HV}[\![P]\!] = \{p(a), p(v_0), p(v_1), \ldots\}$.* □

2.3 Goal dependent semantics

The previous section described semantics of logic programs focusing on models. Such semantics are generally goal independent. The bottom-up semantics described so far provides information about the success set of a program, that is, the answers that can occur during evaluation of the program for a given predicate for *any* supplied goal.

For many applications we are not interested in the whole program model, but only in the computations particular to some goal(s).

Example 4. *Take as an example a simple program;*

$$
\begin{aligned}
p(X) &\leftarrow q(X) \\
q(a) &\leftarrow \\
q(b) &\leftarrow
\end{aligned}
$$

For a goal ← q(Y) the meaning of the predicate p(X) is no longer needed. For a more specific goal, for instance ← q(b), the solution q(a) is no longer needed either. □

Where the bottom-up analysis would provide *success patterns* for the clauses in a program, a top-down goal dependent analysis would provide information about the *call patterns* of the program. The call patterns can provide a basis for program specialisation techniques such as Partial Evaluation (see Section 6.3.1).

2.3.1   Query-Answer Transformation

A query-answer tranformation is a standard transformation technique that will provide sound information about the *queries* or *calls* that can arise during top-down evaluation of a particular goal. A query pattern provides information about how a predicate will be used, such as how it may be instantiated prior to the call of the predicate, in the program. Other names sometimes used for this program transformation technique or related techniques, are *magic set* and *Alexander* transformations [132, 9]. In [11] they are referred to as the *Generalized Magic-sets* algorithms.

To begin with the magic set transformation was used on deductive database queries written in Datalog [9] (see Section 2.4). It would allow the queries to (automatically) be rewritten so they could be implemented bottom-up in a way that would restrict the number of facts generated from the queries, making it possible to implement more efficient joins. Later query-answer transformations were adopted by the program analysis community [115, 141, 117, 35, 72, 61, 73]. A query-answer transformation tool will allow a bottom-up analyser to simulate top-down goal-directed analyses.

A query-answer transformation is a source-to-source program transformation. Given a program P and a goal G, the query-answer transformation derives a new program P_G^q, whose least model $M[\![P_G^q]\!]$ contains those calls and answers needed in a top-down left-to-right computation of P with respect to G. This is both more efficient to compute and yield a better approximation of the part of $M[\![P]\!]$ which is relevant with respect to the supplied goal G [38]. It can however also lead to an increase in the size of the program to be analysed; in the worst case the increase is quadratic [37].

Example 5. *Below is shown an example program along with its query-answer*

transformed version with respect to the query $p(a)$.

$$p(X) \leftarrow q, r(X) \qquad p^a(X) \leftarrow p^q(X), q^a, r^a(X)$$
$$p^q(a) \leftarrow$$
$$q \leftarrow \qquad\qquad\qquad q^a \leftarrow q^q$$
$$q^q \leftarrow p^q(\_)$$
$$r(a) \leftarrow \qquad\qquad\quad r^a(a) \leftarrow r^q(a)$$
$$r(b) \leftarrow \qquad\qquad\quad r^a(b) \leftarrow r^q(b)$$
$$r^q(X) \leftarrow p^q(X), q^a$$

$\square$

A more detailed description of how such a transformation can be implemented is available in [38].

2.4 Datalog

Datalog programs are essentially Prolog programs where no function symbols have an arity greater than zero, specifically meaning only constants appear in programs. Lists are for instance not allowed in Datalog, since the *cons* function is binary. In addition to this, range restrictions may be imposed. A clause, C, is said to be range restricted if every variable occurring in the head of C also occurs in a positive literal in the body of C. For our work we do not enforce the range restrictions.

Negation is allowed in *stratifiable* programs (called stratifiable negation). The dependency graph for a program is a graph where each predicate has its own node and where an edge exists for each rule from head to body predicates. If the literal is negated then the corresponding edge is marked for instance negative. A program is stratifiable if there are no cycles in the graph containing a negative edge. Non-recursive programs are always stratifiable. The following program is *not* stratifiable:

$$
\begin{array}{rcl}
a & \leftarrow & b, c. \\
c & \leftarrow & \neg\, b. \\
b & \leftarrow & a.
\end{array}
$$

The dependency graph for the program above is show in Figure 2.1 on the next page. The negative edge is marked with '$\div$'. The cycle $\{a, c, b\}$ contains a negative edge, hence the program is not stratifiable.

For stratifiable programs the program models are definable as fix point computations. We do not go into detail here since we will not be concerned with abstractions of the semantics. We will use a publicly available tool for computing the model of a stratified Datalog program. For programs with finite sets of predicate and function symbols the model is always finite.
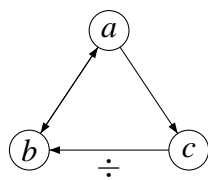
Figure 2.1: Dependency graph of a program that is not stratifiable

Chapter 3

Abstract Interpretation

The previous chapter on logic programming described semantic frameworks based on computing fix points. Most program models are infinite, for example the T_P semantics in Section 2.2.1. $\mathsf{lfp}(T_P)$ cannot be computed, hence we need *safe approximations* that can be computed in reasonable time.

A static analysis of a program is a sound, finite and (usually) approximate calculation of a program's execution. Abstract interpretation [51] is a framework for static analysis using sound approximations of the semantics of a computer program, or more generally, discrete dynamic systems [47]. Cousot and Cousot developed abstract interpretation in 1977 as a way of specifying and subsequently validating static analyses. Prior to Cousot and Cousot, a similar technique was invented by Peter Naur for checking ALGOL source programs for compatibility of type identifiers [124]. He described the technique as a basic method for pseudo-evaluation of the expressions of a program, proceeding like a run-time evaluation, but operating on descriptions of the types and kinds of operands, rather than actual values in the program. This could also be described as a partial execution of the program, gaining information about the semantics of the particular program without actually performing all the computations.

Abstract interpretation has since its introduction been an active field of research, and it has gradually found new areas where it can be applied, such as types in programming languages and constraint solving. A recent overview of abstract interpretation is available in [48]. Abstract interpretation has been applied to a wide set of programming languages, both imperative and declarative languages. It was first introduced for logic programming by Mellish [116], Debray and Warren [62] and Bruynooghe *et al.* [27, 24] who described an established framework for abstract interpretation of logic programming.

An abstraction of a program can be used to simplify problems otherwise not computable in realistic time, to a manageable problems that can be solved using currently available computing resources in terms of computing time and storage.

The abstraction should still maintain enough precision to be able to reason about the analysed program from the results obtained. When an analysis otherwise not computable in realistic time has been reduced to a simpler but computable analysis, commonly, the results obtained in the simpler analysis is only an approximation of the correct answer to the complex analysis. An *approximation* is an inexact representation of the correct solution. A *sound approximation*, sometimes also referred to as *safe* or *over* approximation, is a set of answers to the analysis where all the correct answers are included in the approximation. The partial ordering relation $\sqsubseteq$ defined later, can informally be viewed as describing accuracy of approximations. That is, if $a \sqsubseteq b$ then b is a safe approximation of a. If $a \sqsubseteq b \sqsubseteq c$ and $b \neq c$, then b is a more accurate approximation of a than the approximation c, i.e. containing less incorrect answers.

An abstract operation in an abstract domain is a sound approximation of a concrete operation in the concrete domain. The abstract operations can be coarse enough to be computable and still precise enough to be useful in practice. Abstract interpretation is based on monotonic functions over ordered sets, making it a good theoretic foundation for reasoning about the soundness of the approximations.

Chapter Overview

- As is the case with many program analysis frameworks, the formal description of abstract interpretation relies on partially ordered sets (posets), complete lattices and fix point computations. These are defined in Section 3.1.

- A key point in abstract interpretation is accelerating convergence to a fix point. This is described in Section 3.2.

3.1 Posets, Lattices and Fix Points

Definition 19 (Partial order). *A partial order over a set L, is a binary relation* $\sqsubseteq: L \times L \to \{true, false\}$ *which is reflexive, transitive and anti-symmetric:*

- *Reflexive:* $\forall l \in L : l \sqsubseteq l$

- *Transitive:* $\forall l_1, l_2, l_3 \in L : l_1 \sqsubseteq l_2$ *and* $l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$

- *Anti-symmetric:* $\forall l_1, l_2 \in L : l_1 \sqsubseteq l_2$ *and* $l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$

Definition 20 (Partially ordered set). *A partially ordered set is a set with an associated partial ordering and its shorter name is a* poset $\langle L, \sqsubseteq \rangle$. *An example of such a poset could be the natural numbers ordered by the* $\leq$ *relation* $\langle \mathbb{N}, \leq \rangle$.

An upper bound of a subset A of L is an element $l \in L$ such that $\forall l_0 \in A : l_0 \sqsubseteq l$. The least upper bound l of A is an element that for all upper bounds l_0 of A satisfies $l \sqsubseteq l_0$. Similarly a lower bound $l \in L$ of A is an element such that $\forall l_0 \in A : l \sqsubseteq l_0$. The greatest lower bound l of A is an element that for all lower bounds l_0 of A satisfies $l_0 \sqsubseteq l$.

Definition 21 (Complete lattice)**.** *A complete lattice is a poset $\langle L, \sqsubseteq \rangle$ if every subset A of L has both a greatest lower bound (*glb*) and a least upper bound (*lub*).*

The glb of a set A is denoted $\bigsqcap A$ and are in some contexts called the *meet* operator. The lub of A is denoted $\bigsqcup A$ and will sometimes be referred to as the *join* operator.

The *least element* of a lattice $\langle L, \sqsubseteq \rangle$ is denoted $\bot = \bigsqcap L$ and the *greatest element* is denoted $\top = \bigsqcup L$. The glb, lub, least element and greatest element can be included in the tuple describing a complete lattice L, i.e. $L = \langle L, \sqsubseteq, \bigsqcap, \bigsqcup, \bot, \top \rangle$.

Particularly for abstract interpretation, the top element $\top$ is said to contain no information. The partial order relation can be described as either denoting which of two elements contains more information, are more defined or is a more precise approximation. For example, for $a \sqsubseteq b$ the following statements are equivalent; a contains more information than b, a is more defined than b and a is a more precise approximation than b.

Definition 22 (Semi lattice)**.** *A semi-lattice is a poset $\langle L, \sqsubseteq \rangle$ where for all subsets A of L they are either all closed under meet, $\bigsqcap A$, or all closed under join, $\bigsqcup A$.*

Which of the operators it is closed under is normally specified in the denotation of the semi-lattice, for example $L = \langle L, \sqsubseteq, \bigsqcap \rangle$.

Lattices are sometimes illustrated using Hasse diagrams. A line going upwards from l_1 to l_2 means that $l_1 \sqsubseteq l_2$. Figure 3.1 illustrates the partially ordered set $\langle \mathcal{P}(\{a, b, c\}), \subseteq \rangle$.

Definition 23 (Monotonic function)**.** *A function f is called* monotonic *if*

$$\forall l_1, l_2 \in L : l_1 \sqsubseteq l_2 \Rightarrow f(l_1) \sqsubseteq f(l_2)$$

Monotonic functions on ordered sets are order preserving functions. Intuitively this requires for the function f to be monotonic, that if the input l_1 is more defined than l_2, then the output $f(l_1)$ is also more defined than $f(l_2)$. The composition of monotonic funtions is also monotonic; if f and g are monotonic functions, then $l_1 \sqsubseteq l_2 \Rightarrow f(l_1) \sqsubseteq f(l_2) \Rightarrow g(f(l_1)) \sqsubseteq g(f(l_2))$.

Definition 24 (Fix point)**.** *A* fix point *of a function is a point that is mapped to itself by the function.*

$$\top = \{a, b, c\}$$

$$\{a, b\} \qquad \{a, c\} \qquad \{b, c\}$$

$$\{a\} \qquad \{b\} \qquad \{c\}$$

$$\bot = \emptyset$$
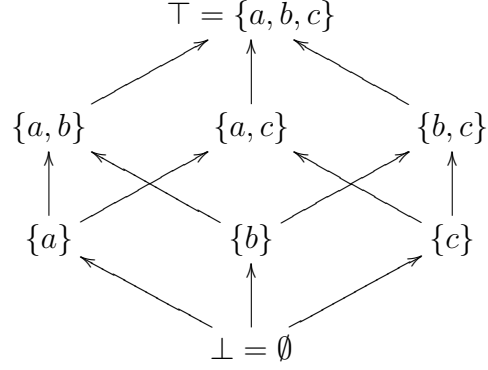
Figure 3.1: Hasse diagram of the lattice $\langle \mathcal{P}(\{a, b, c\}), \subseteq \rangle$

Consider a monotonic function $f : L \to L$ on a complete lattice $L = \langle L, \sqsubseteq, \sqcap, \sqcup, \bot, \top \rangle$. A fixed point of f is an element $l \in L$ such that $f(l) = l$. The set of fixed points is denoted and defined as

$$Fix(f) = \{l | f(l) = l\}$$

The function f is *reductive* at l if and only if $f(l) \sqsubseteq l$. The set of elements on which f is reductive, is denoted and defined as

$$Red(f) = \{l | f(l) \sqsubseteq l\}$$

Finally the function f is *extensive* at l if and only if $l \sqsubseteq f(l)$. The set of elements on which f is extensive, is denoted and defined as

$$Ext(f) = \{l | l \sqsubseteq f(l)\}$$

It follows from the definitions that $Fix(f) = Red(f) \bigcap Ext(f)$; if $f(l) \sqsubseteq l$ and $l \sqsubseteq f(l)$ holds, it implies $f(l) = l$.

Tarski's fix point theorem [144] states the following:

Theorem 1. *Let L be a complete lattice and $f : L \to L$ be a monotonic function. Then the set of fixed points, $Fix(f) \subseteq L$, is also a complete lattice.*

Since a complete lattice cannot be empty, this guarantees the existence of at least one fix point of f and even the existence of a least and greatest fix point. The greatest lower bound of $Fix(f)$ is denoted $\mathsf{lfp}(f)$, *least fix point* of f. Tarski's Theorem ensures [125]:

$$\mathsf{lfp}(f) = \bigsqcap Fix(f) = \bigsqcap Red(f) \in Fix(f) \subseteq Red(f)$$

28

A least upper bound of $Fix(f)$ also exists in L and is denoted by $\mathsf{gfp}(f)$, *greatest fix point* of f. This also follows from Tarski's Theorem:

$$\mathsf{gfp}(f) = \bigsqcup Fix(f) = \bigsqcup Ext(f) \in Fix(f) \subseteq Ext(f)$$

When lattice theory is used in the context of program analysis it usually involves the construction of some sequence of elements of a complete lattice. Here a sequence will be denoted $(l_n)_n$, where $n \in \mathbb{N}$.

Definition 25 (Chain). *For a partially ordered set L, a subset $Y \subseteq L$, is a chain if*

$$\forall l_1, l_2 \in Y : (l_1 \sqsubseteq l_2) \vee (l_2 \sqsubseteq l_1)$$

A sequence of elements, $\{l_n | n \in \mathbb{N}\}$, in L is an *ascending chain* if $n \leq m \Rightarrow l_n \sqsubseteq l_m$ and *descending chain* if $n \leq m \Rightarrow l_n \sqsupseteq l_m$.

Definition 26 (Ascending Chain Condition). *A partially ordered set, L, satisfies the* Ascending Chain Condition *if every ascending chain $l_1 \sqsubseteq l_2 \sqsubseteq \ldots$ of elements in L is eventually stationary. The chain is stationary if there exists an $n \in \mathbb{N}$ such that $l_m = l_n$ for all $m > n$.*

Definition 27 (Continuous functions between partially ordered sets). *Let L be a complete lattice, then a function $f : L \rightarrow L$ is* continuous *if for all $Y \subseteq L$, then $\bigsqcup f(Y) = f(\bigsqcup Y)$ holds.*

In other words, continuous functions preserve limits. Continuity of a function also implies monotonicity for this function. And if L is finite, monotonicity would also imply continuity. In program analysis it is usually necessary to compute some fix point of a monotonic function $f : L \rightarrow L$ where L is a complete lattice. Typically this is done by computing the limit of an *ascending Kleene chain*. The *Kleene fix point theorem* [54] (Proposition 23) states:

Theorem 2. *For any complete lattice L and any continuous function $f : L \rightarrow L$ the $\mathsf{lfp}(f)$ is the least upper bound of the ascending Kleene chain:*

$$\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq \ldots$$

Often this property is expressed as a formula:

$$\mathsf{lfp}(f) = \bigsqcup \{f^i(\perp) | i \in \mathbb{N}\}$$

If the chain stabilises it will stabilise to $\mathsf{lfp}(f)$; this occurs for continuous functions f. For finite lattices this would also occur for the monotone functions. In abstract interpretation the abstract domain over which the analysis is executed can be

any kind of domain that can be described by a lattice. There are numerical domains such as intervals, difference-bound matrices or convex polyhedra, and there are symbolic domains such as types describing the set of values a variable can be assigned or types describing states that a variable can be in (live, undefined, constant etc.).

In particular, the numerical domains can be computationally expensive and may not satisfy the ascending chain condition. Convergence of the fix point computations can be *enforced* and *accelerated*. In this process precision is traded for better computational properties. The next section describes this approach.

3.2 Accelerating convergence to fix point

For abstract interpretation frameworks the classical method for reaching a fix point is based on *widening* [54]. Galois Connections (see Section 3.2.3) are used to construct a set of approximate *semantic equations* expressing the semantics of a given program over some abstract domain. Widening combined with Galois Connections will ensure eventual termination of the fix point iterations over the set of semantic equations.

3.2.1 Widening

Widening is a technique often used to approximate $\mathsf{lfp}(f)$. Widening can be used to construct chains that converge to a fix point much faster than the Kleene chain, giving the particular abstract interpretation better behavior with regards to computing time required for completion of the analysis. Widening will enforce stabilisation of the fix point computations. The fix point obtained using widening will be a safe approximation of $\mathsf{lfp}(f)$.

A new operator, called a *widening operator*, is constructed. How this operator is constructed will affect the precision of the approximated $\mathsf{lfp}(f)$ and the computational cost of finding the approximation. There is usually a trade-off between precision and efficiency in the process of accelerating the fix point computations to convergence. Usually a widening operator that is quicker to reach convergence will also result in a less precise result. Different widening operators can be applied, even within the same analysis application [46], depending on whether precision or speed is more important. The first widening operator was suggested by Cousot and Halbwachs [57].

The principle behind the widening operator comes from the fact that any sequence of elements in a lattice L can be transformed into an ascending chain by an

upper bound operator. An upper bound operator $\breve{\sqcup} : L \times L \to L$ always returns an element greater than its arguments; i.e. $\forall l_1, l_2 \in L, l_1 \sqsubseteq (l_1 \breve{\sqcup} l_2) \sqsupseteq l_2$.

Given a total function $\phi : L \times L \to L$ and a sequence of elements from L a new sequence l_n^ϕ can be defined as

$$l_n^\phi = \begin{cases} l_n & \text{if } n = 0 \\ l_{n-1}^\phi \; \phi \; l_n & \text{if } n > 0 \end{cases}$$

A widening operator belongs to the family of upper bound operators.

Definition 28 (Widening operator). *An operator $\nabla : L \times L \to L$ is a widening operator if and only if*

1. *It is an upper bound operator*

2. *For all ascending chains $(l_n)_n$ the ascending chain $(l_n^\nabla)_n$ satisfies the ascending chain condition*

After the widening operator has been defined, the next step is to find a safe approximation of $\mathsf{lfp}(f)$. This is done by calculating the sequence

$$f_\nabla^n = \begin{cases} \bot & \text{if } n = 0 \\ f_\nabla^{n-1} \nabla f(f_\nabla^{n-1}) & \text{otherwise} \end{cases}$$

This sequence will eventually stabilise at $f(f_\nabla^m) \sqsubseteq f_\nabla^m$ for some m, and we take $\mathsf{lfp}_\nabla(f) = f_\nabla^m$ as the safe over approximation of $\mathsf{lfp}(f)$.

3.2.2 Narrowing

Narrowing is a technique designed to improve the precision of the approximation of $\mathsf{lfp}(f)$ obtained in the widening step. If $f(f_\nabla^m) \sqsubseteq f_\nabla^m$ holds, it follows from the definition of the set of reductive elements, $Red(f) = \{l \mid f(l) \sqsubseteq l\}$, that f is reductive at f_∇^m, and from Tarski that $\mathsf{lfp}(f) = \bigsqcap Red(f) \in Fix(f)$. Narrowing is defined as a descending chain $f^n(f_\nabla^m) \in Red(f)$ ensuring $\mathsf{lfp}(f) \sqsubseteq f^n(f_\nabla^m)$.

Definition 29 (Narrowing operator). *An operator $\Delta : L \times L \to L$ is a narrowing operator if and only if*

1. $\forall l_1, l_2 \in L : l_1 \sqsubseteq l_2 \Rightarrow l_1 \sqsubseteq (l_2 \Delta l_1) \sqsubseteq l_2$

2. *for all descending chains $(l_n)_n$ the sequence $(l_n^\Delta)_n$ eventually stabilises*

Once the narrowing operator has been defined, the next step is to construct the descending chain:

$$f_\Delta^n = \begin{cases} f_\nabla^m & \text{if } n = 0 \\ f_\Delta^{n-1} \Delta f(f_\Delta^{n-1}) & \text{if } n > 0 \end{cases}$$

The descending chain $(f_\Delta^n)_n$ will stabilise to an overapproximation of $\mathsf{lfp}(f)$. It it safe to impose a limit on the number of elements in this sequence. If condition 2 of Definition 29 is not satisfied for a given operator then it is safe to apply this operator a fixed number of times, as long as condition 1 is satisfied.

3.2.3 Galois Connections

Computations on a lattice L can be costly and possibly infinite. One of the principal ideas behind abstract interpretation is the substitution of the uncomputable lattice L with a simpler lattice M. The elements in L will be mapped to elements in M. Analysis will then take place on the lattice M. The simpler lattice will generally be less expressive than the complex lattice, so a loss of precision will occur when elements are mapped from L to M. *Galois Connections* are useful for approximating one set of computations with another set of computations requiring less time, resources etc. to solve. The result of the simpler set of computations will be a safe approximation of the result of the set of complex computations, but in general it will also be a less precise approximation.

An *abstraction function* is introduced to describe elements of L in terms of elements in M:

$$\alpha : L \to M$$

Similarly a *concretisation function* is introduced to map the elements of M to elements of L:

$$\gamma : M \to L$$

The relationship between α and γ is formalised by Galois Connections.

Definition 30 (Galois Connection). $\langle L, \alpha, \gamma, M \rangle$ *is a* Galois Connection *between the lattices* $\langle L, \sqsubseteq_L \rangle$ *and* $\langle M, \sqsubseteq_M \rangle$ *if and only if:*

1. $\alpha : L \to M$ *is monotonic*

2. $\gamma : M \to L$ *is monotonic*

3. $\forall l \in L : \gamma(\alpha(l)) \sqsupseteq_L l$

4. $\forall m \in M : \alpha(\gamma(m)) \sqsubseteq_M m$

These restrictions guarantees an important property of the abstraction and concretisation functions namely a safety property. For an element $l \in L$, going

from lattice L to lattice M, $m = \alpha(l), m \in M$ and back, $l_\alpha = \gamma(m), l_\alpha \in L$, is a safe approximation of l. It may be an over approximation where precision is lost, but it will be a safe approximation, i.e. $l \sqsubseteq_L l_\alpha$. This property can also be described more informally as preserving ordering between the two lattices; going from one lattice to the other and back can never result in going downwards in the lattices.

The partial ordering relation $\sqsubseteq$ over a set L can be extended to functions on L;

$$F \sqsubseteq F' \Leftrightarrow \forall x \in L : F(x) \sqsubseteq F'(x)$$

Approximation of L by M can then be extended to approximation of functions on L by introducing *functional abstraction* and *functional concretisation* [50], with

$$F^\alpha(m) \stackrel{def}{=} \alpha(F(\gamma(m)))$$

This definition of F^α gives the most precise possible abstraction of F. Any function approximating F^α would suffice.

By proposition 30 in [54], if L is a poset, F is continuous and $\bot_M = \alpha(\bot_L)$ then $\mathsf{lfp}(F) \sqsubseteq \gamma(\mathsf{lfp}(F^\alpha))$.

3.2.4 Systems of Semantic Equations

For a program P let N be the set of nodes (or *program points*) in the control flow graph of P. Let us assume that computing the abstract interpretation of P over some abstract domain is equivalent to find the least solution of a system of semantic equations

$$X = F(X) \equiv \begin{cases} x_1 &= f_1(x_1, \ldots, x_n) \\ &\vdots \\ x_n &= f_n(x_1, \ldots, x_n) \end{cases} \tag{3.1}$$

where each index $i \in N = [1 \ldots n]$ and each function $f_i : L^n \to L$ is a monotone function on a lattice L of abstract properties, for example could each function f_i be a sub-function computing the computational state at a specific program point.

Performing a program analysis usually involves computing a fix point of a set of (recursive) semantic equations $X = F(X)$ starting from an initial value $\bot$, as outlined in Section 3.1. These computations can be infinite, prohibitively expensive to compute or perhaps the desired program property to analyse with respect to, cannot easily be stated as formulas. In this situation Galois Connections can be used to approximate computations on a concrete domain with computations on

an abstract domain, replacing $X = F(X)$ with $X = F^\alpha(X)$, for which a fix point can be computed and for which the fix point on the abstract domain is a safe approximation of a fix point in the concrete domain.

Chapter 2 introduced logic programming where the concrete semantics is based on the Herbrand pre-interpretation. The next chapters will describe abstract inter-pretation of logic programs. Galois Connections will map elements of the abstract domain to sets of elements of the concrete domain.

Two specific abstract domains will be described; a symbolic (finite) domain over pre-interpretations derived from regular type definitions (Chapter 5) and a numerical (infinite) domain over convex polyhedra (Chapter 4). Infinite domains may not satisfy the ascending chain condition hence widening will be required for the convex polyhedron domain.

3.2.5 Combining Widening and Galois Connections

Galois Connections are used to construct semantic equations, $X = F(X)$, and fix point iterations are then used to solve these equations iteratively. Widening can then be used to accelerate convergence to a fix point. The next section will describe a technique for finding which equations widening should be applied to in order to ensure convergence. Once a fix point has been reached, narrowing can be used to refine the approximation of the least fix point.

3.2.6 Applying Widening

Much effort has gone into examining strategies for applying widening operators in order to minimise loss of precision and minimise computational effort. Since widening results in a loss of precision, usually these strategies will try to minimize the number of times widening is applied in order to reach a fix point.

A *chaotic iteration* [52] over the semantic equations shown in Equation 3.1 is an iteration of the system of equations where the iteration has been limited to any subset of the equations in the system. Cousot and Cousot showed in [52, 45] that any chaotic iteration method converges to the optimal solution, given f_i is a continuous function on a complete lattice. Any such strategy that takes advantage of these two properties are generally referred to as *chaotic iteration strategies*.

Applying a chaotic iteration strategy to the system of semantic equations $X = F(X)$ would result in a finite sequence $i_1, \ldots, i_m$ of elements in N, so that the iterations $x_{i_j} = f_{i_j}(x_1, \ldots, x_n) \in X = F(X)$ for $j \in \{1, \ldots, m\}$ would result in a fix point solution to that particular system of equation. How a particular sequence is derived would depend on what particular chaotic iteration strategy is used to iterate over a given set of semantic equations. Of two different chaotic iteration

strategies, the better strategy would provide a shorter sequence to iterate over (i.e. smaller m).

To accelerate convergence to a safe approximation of the fix point for the equations, the widening operator is applied to a subset $W \subseteq N$ of the equations. Widening with respect to a set of *widening points* W is denoted and defined as:

Definition 31 (Widening with respect to W). *For the semantic equations, $X = F(X)$, with $n = |X|$, and for a set of widening points, $W \subseteq \{1, \ldots, n\}$, and a widening operator ∇, widening with respect to a set of widening points is denoted*

$$X = X \ \nabla_W \ F(X)$$

where

$$
\begin{aligned}
x_i &= x_i \nabla f_i(x_1, \ldots, x_n) \quad \textit{for } i \in W \\
x_i &= f_i(x_1, \ldots, x_n) \qquad \textit{otherwise}
\end{aligned}
$$

The dependency graph of the equations is a directed graph $G = (N, E)$, where N is a set of nodes, and E is a set of edges, with an edge $(i, j) \in E$, if f_j depends on its parameter x_i. This graph will typically be closely related to the control flow graph of the program P.

Introducing widening into the semantic equations generally results in a loss of precision. One way of minimising this loss of precision is to find a good set of widening points, W, as small as possible. When W is chosen such that every loop in the dependency graph contains at least one element also in W, then any chaotic iteration strategy will terminate with a safe approximation of $\mathsf{lfp}(f)$ [21].

The problem of finding the smallest set of nodes in a graph, such that all loops contains a node in this set, is also called the *Feedback Vertex Set*. It is an NP-complete problem and was in fact among the first problems shown to be NP-complete. It is part of Richard Karp's list of 21 NP-Complete Problems [95] from 1972. The definition of the problem is; given a directed graph $G = (N, E)$ and a positive integer k, is there a subset $X \subseteq N$ with $|X| \leq k$ such that $G \setminus X$ is cycle free? Exact computation of this problem for medium sized graphs can for instance be solved using relational algebra [15].

Methods for finding good sets of widening points have previously been suggested by, among others, François Bourdoncle who suggested methods based on *weak topological orderings* [21], and Patrick Cousot who suggested a method for finding *loop head nodes* [46]. Graphs can easily be constructed where both methods suggested would provide a solution that is not optimal. Figure 3.2 on the following page shows a graph where the optimal solution is a single node, here node 3. The weak topological ordering method and the loop head nodes method would both provide a solution containing two nodes, either $W = \{2, 3\}$ or $W = \{3, 4\}$.
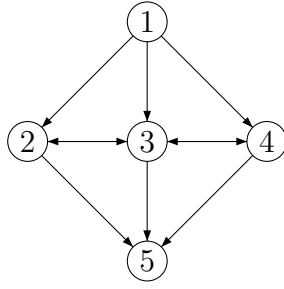
Figure 3.2: Directed Graph with minimal $W = \{3\}$

In Section 4.3.5 we will present a new algorithm for computing widening points and discuss its advantages.

Chapter 4

Convex Polyhedron Analysis

Convex polyhedron analysis, first used by Cousot and Halbwachs [57], is an application of abstract interpretation. The program to be analysed is interpreted over an abstract domain of convex polyhedra. Operations in the program are approximated by operations on convex polyhedra. The convex polyhedral domain is not finite so a method such as widening must be established for this particular analysis to ensure termination.

A convex polyhedron analysis results in abstraction of the variables in a program by a set of linear constraints relations among these variables. This has been the basis of a variety of program analyses, including the field of logic and constraint logic programming [12, 13], for instance for argument-size analysis, time-complexity analysis and termination analysis [98]. In analysis of imperative languages it has been used for purposes such as detection of overflows and loop invariants.

In the context of logic programming, we wish to obtain an approximation of the program model in which each n-ary predicate is approximated by an n-dimensional polyhedron, representing some numerical description of its argument's values.

For example, a termination analyser is used to determine whether a program will *definitely* terminate or *possibly* not terminate. Termination analysis for logic programs have been studied widely [63]. Different approaches have been suggested and a large proportion of these rely on abstracting the programs using *norms*.

Definition 32 (Norm)**.** *A norm is a mapping* $\| \, . \, \|: \mathsf{Term}_\Sigma \to \mathbb{N}$.

Two classic norms are the *list length* norm [148] mapping lists to their length, and the *term size* norm [118] counting the number of functors in a term. Only the list length will be defined here.

Definition 33 (List length norm). *The list length norm, denoted $\| \, . \, \|_l$, is defined as*

$$
\begin{aligned}
\| \, [t_1|t_2] \, \|_l &= 1 + \| \, t2 \, \|_l \quad \text{for } t_1, t_2 \in \mathsf{Term}_\Sigma \\
\| \, t \, \|_l &= 0 \qquad\qquad \text{otherwise}
\end{aligned}
$$

Example 6. *The append program and the abstracted program with the list length norm applied is shown below.*

$append([\,], Xs, Xs) \leftarrow$	$append^{\mathcal{A}}(0, Xs, Xs) \leftarrow$
$append([X|Xs], Ys, [X|Zs]) \leftarrow$	$append^{\mathcal{A}}(1 + Xs, Ys, 1 + Zs) \leftarrow$
$append(Xs, Ys, Zs)$	$append^{\mathcal{A}}(Xs, Ys, Zs)$

□

The convex polyhedron analyser would then be applied to the abstracted program to find, for instance, the linear relations between the sizes of the arguments of the program.

Chapter Overview

This chapter describes a convex polyhedron analyser for constraint logic programs. The implementation of the analyser is based on the Parma Polyhedra Library.

- Section 4.1 defines convex polyhedra.

- Section 4.2 describes how convex polyhedra can serve as an abstract domain for abstract interpretation.

- Section 4.3 describes an implementation of a convex polyhedron analyser for constraint logic programs.

4.1 Convex Polyhedra

4.1.1 Definition of polyhedra

Polyhedra are geometric representations of linear systems of equalities and inequalities. A convex polyhedron is a region of an n-dimensional space that is bounded by a finite set of hyperplanes. The definition is:

Definition 34 (Closed Convex Polyhedron). *The set $\mathcal{P} \subseteq \mathbb{R}^n$ is a closed convex polyhedron if either*

- *it is the intersection of a finite family of closed linear halfspaces of the form $\{x|ax \geq c\}$ where $\mathbb{R}^n$ is the n-dimensional vector space on real numbers, $a \in \mathbb{R}^n$ is a non-zero row vector, $c \in \mathbb{R}$ is a scalar constant and $x = \langle x_1, \ldots, x_n \rangle$*

- *$n = 0$ and $\mathcal{P} = \emptyset$*

The set of all closed convex polyhedra on the vector space $\mathbb{R}^n$ is denoted $\mathbb{CP}_n$.

Certain constraint systems cannot easily be described using closed convex polyhedra, such as constraints partitioning the real numbers into disjoint sets. Accurate descriptions would require the use of strict inequalities.

Example 7. *The two branches of an if-then-else construct such as the one shown below, cannot be distinguished using closed polyhedra.*

$$if\ x \geq 0\ \{\ldots\}$$
$$else\ \{\ldots\}$$

For not necessarily closed convex polyhedra, strict inequalities are allowed.

Definition 35 (Not Necessarily Closed Polyhedron)**.** *The set $\mathcal{P} \subseteq \mathbb{R}^n$ is a not necessarily closed polyhedron (NNC polyhedron) if either*

- *it is the intersection of a finite family of open or closed linear halfspaces of the form $\{x|ax \geq c\}$ or $\{x|ax > c\}$ where $\mathbb{R}^n$ is the n-dimensional vector space on real numbers, $a \in \mathbb{R}^n$ is a non-zero row vector, $c \in \mathbb{R}$ is a scalar constant and $x = \langle x_1, \ldots, x_n \rangle$*

- *$n = 0$ and $\mathcal{P} = \emptyset$*

The set of all NNC polyhedra on the vector space $\mathbb{R}^n$ is denoted $\mathbb{P}_n$.

4.1.2 Representation of polyhedra

Every polyhedron $\mathcal{P}$ has a dual representation; a constraint representation and a parametric representation consisting of a set of lines, rays, points and closure points together called a generator system for a polyhedron. Only the constraint representation will be defined here. A constraint in this context is a linear equality or linear inequality. We denote sets of constraints using matrix notation. For the constraints containing non-strict inequalities the notation is

$$\begin{bmatrix} a_{1,1} & \ldots & a_{1,n} \\ \vdots & & \vdots \\ a_{m,1} & \ldots & a_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \geq \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \ \text{or in short form } Ax \geq b$$

$$\mathcal{P} = \left\{ (x,y) \,\middle|\, \begin{pmatrix} y & \geq & 1 \\ x & \geq & 1 \\ -x + -y & > & -5 \end{pmatrix} \right\}$$
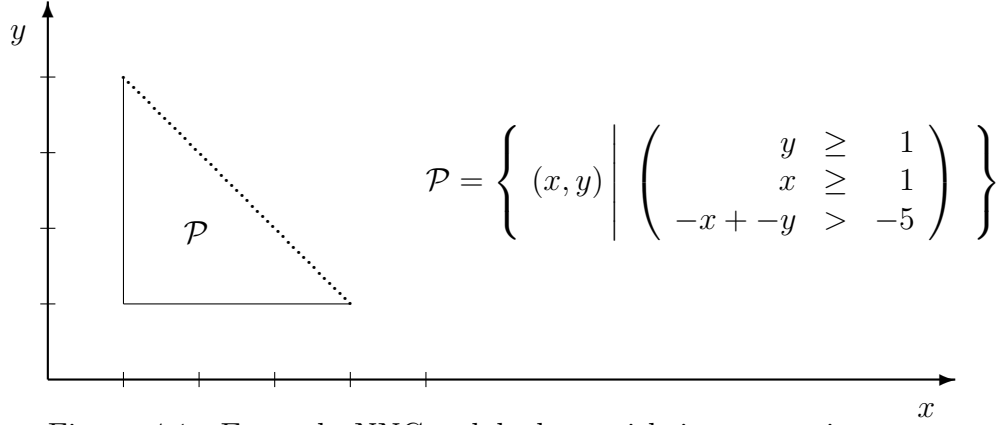
Figure 4.1: Example NNC polyhedron with its constraint representation. The dotted line illustrates the open edge produced by the strict inequality.

In this notation the non-strict inequality can be substituted for equalities or strict inequalities. Each NNC polyhedron is the set of solutions to a constraint system.

$$\mathcal{P} \stackrel{def}{=} \{x \in \mathbb{R}^n | A_1 x = b_1, A_2 x \geq b_2, A_3 x > b_3\},$$

where $\forall i \in \{1,2,3\}$, $A_i \in \mathbb{R}^{m_i} \times \mathbb{R}^n$ and $b_i \in \mathbb{R}^{m_i}$ and m_1, m_2, $m_3 \in \mathbb{N}$ are the numbers of equalities, inequalities and strict inequalities respectively. We use $\mathcal{C}$ to denote the constraint representation of the polyhedron $\mathcal{P}$:

$$\mathcal{P} = con(\mathcal{C})$$

Figure 4.1 shows a polyhedron $\mathcal{P} \in \mathbb{CP}_2$ with 3 vertices and its representation as constraints.

4.1.3 Operations on polyhedra

For a convex polyhedron analysis of a program a few operations on polyhedra are required. These operations are described next. They can be implemented for both the constraint and parametric representation of polyhedra.

Intersection

The *intersection* of two polyhedra $\mathcal{P}_1, \mathcal{P}_2 \in \mathbb{P}_n$ is denoted $\mathcal{P}_1 \cap \mathcal{P}_2$.
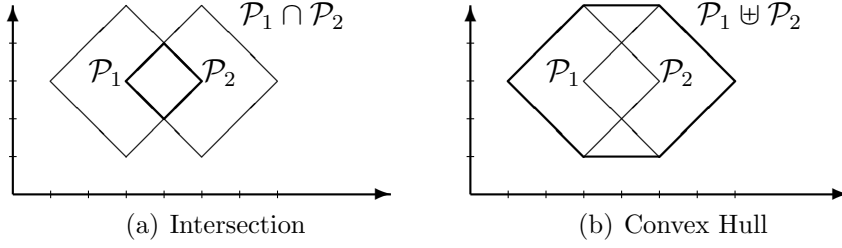
Figure 4.2: Operations on convex polyhedra

Convex Hull

The *convex polyhedral hull* of two polyhedra $\mathcal{P}_1, \mathcal{P}_2 \in \mathbb{P}_n$ is denoted $\mathcal{P}_1 \uplus \mathcal{P}_2$, and it is the smallest NNC convex polyhedron that contains both $\mathcal{P}_1$ and $\mathcal{P}_2$. The convex hull is an upper approximation of union, since convex polyhedra generally are not closed under union.

Figure 4.2(a) shows the intersection between two polyhedra, $\mathcal{P}_1, \mathcal{P}_2 \in \mathbb{CP}_2$, and Figure 4.2(b) the convex hull of $\mathcal{P}_1$ and $\mathcal{P}_2$.

Emptiness

Given a set of constraints the polyhedral representation of this set may be empty if the constraints are not satisfiable. For example the set $\{X < 0 \wedge Y < 0 \wedge Z > 0 \wedge Z = X + Y\}$ of constraints is not satisfiable - the sum of two negative real numbers can not be a positive real number. Checking satisfiability of a set of *linear* constraints is decidable.

Inclusion

The convex polyhedron analysis proceeds as a fix point computation. Some mechanism of establishing when convergence is reached is needed. This can be achieved by checking inclusion between two polyhedra $\mathcal{P}_1, \mathcal{P}_2 \in \mathbb{P}_n$, $\mathcal{P}_1$ is *entailed* by $\mathcal{P}_2$ if $\mathcal{P}_1 \subseteq \mathcal{P}_2$. This is also decidable.

Projection

The projection operation returns the most precise polyhedron which does not depend on a given dimension. Given an n-dimensional polyhedron $\mathcal{P}$, the projection $\mathcal{P}' = proj(\mathcal{P}, j)$, will return the most precise polyhedron, $\mathcal{P}'$, of dimensions $n - 1$ that is entailed by $\mathcal{P}$ excluding its constraints on dimension j.
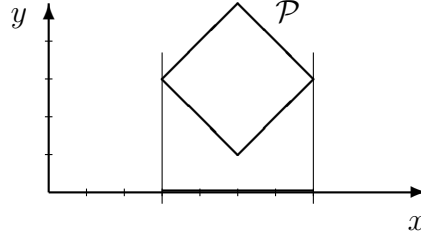
Figure 4.3: Projection of $\mathcal{P}$ onto x

Depending on which programming language is being analysed, the program constructs that will be approximated by a polyhedra can be objects such as function calls, loops etc. and specifically for constraint logic programs, the denotation of a predicate with numeric arguments could be approximated by a polyhedron. The dimension of the approximating polyhedron will be equivalent to the number of variables occurring in that program construct. Some of these variables may have a local context, i.e. they only occur in that particular program construct. For the predicates in a logic programming the local context is a clause body. Any variable occurring in the clause body but not in the clause head must be removed before the predicate is given its denotation. This is where the projection operation is applied.

A clause head can of course contain variables that do not occur in the body. These variables will be unrestricted. Increasing the dimension of a polyhedron is simpler than projecting dimensions out. In this situation no precision is lost, since any polyhedron of dimension n has an exact representation as a polyhedron of dimension $n + 1$.

Figure 4.3 illustrates a 2 dimensional polyhedron projected onto the x-axis. The resulting polyhedron is an interval over x.

4.2 Convex Polyhedral Domain for Abstract Interpretation

This section describes how convex polyhedra can serve as an abstract domain for the abstract interpretation of logic programs.

4.2.1 Partial Ordering on Polyhedra

Polyhedra can be partially ordered by set inclusion i.e. $\langle \mathbb{P}_n, \subseteq \rangle$. We can add intersection as the greatest lower bound ($\sqcap$), the convex polyhedral hull as the least upper bound ($\sqcup$) and $\mathbb{R}^n$ as top element and $\emptyset$ as the bottom element; thus we

have a complete lattice $\langle \mathbb{P}_n, \subseteq, \cap, \uplus, \emptyset, \mathbb{R}^n \rangle$. This property makes convex polyhedra a suitable domain for program analysis - in particular for abstract interpretation.

The polyhedron will provide an abstraction of each program point in the program, where the definition of a program point would depend on the programming language and the program analysis. For a logic program the program points would be the predicates in that program.

The abstract domain is the set of mappings $(Pred \mapsto \mathbb{P})$ where an n-ary predicate $p \in Pred$ is mapped to an n-dimensional polyhedron $\mathcal{P} \in \mathbb{P}^n$. Inclusion is used as the ordering over the mappings. For instance, for the mappings m_1 and m_2 the ordering is

$$m_1 \sqsubseteq m_2 \equiv \forall p \in Pred, \ m_1(p) \subseteq m_2(p)$$

For a finite set of predicates $\{p_1, \ldots, p_k\}$ we can represent the mappings as a tuple of polyhedra $\langle \mathcal{P}_1, \ldots, \mathcal{P}_k \rangle$. The concretisation function of the mapping m would be defined as

$$\gamma(\langle p_1, \ldots, p_k \rangle) = \{p_i(\bar{t}) | 1 \leq i \leq k, \bar{t} \in \mathcal{P}_i\}$$

where $\bar{t}$ denotes a point in the polyhedron $\mathcal{P}_i$.

For implementation purposes we can also represent such a mapping as a set of constrained atoms

$$p(x_1, \ldots, x_n) \leftarrow c(x_1, \ldots, x_n)$$
$$\text{where} \quad \mathcal{P} = con(c(x_1, \ldots, x_n)) \text{ for some } n\text{-dimensional polyhedron } \mathcal{P}$$

The abstract domain for a program would typically be a set, S, of polyhedra, one element for each program point. The lattice structure is an extension of the one defined above, for example $\langle P_1, P_2, \ldots, P_k \rangle \sqsubseteq \langle Q_1, Q_2, \ldots, Q_k \rangle$ iff $P_1 \subseteq Q_1 \land P_2 \subseteq Q_2 \land \ldots \land P_k \subseteq Q_k$.

4.2.2 Widening for Convex Polyhedra

In the domain of convex polyhedra infinite ascending chains can occur, hence the ascending chain condition is not satisfied. For program analysis purposes some mechanism to accelerate convergence of the fix point computations may be required to ensure termination. The most used mechanism for this is widening [54], as described in Section 3.2.1. Widening in the convex polyhedra domain was defined by Cousot and Halbwachs [57] and refined in Halbwach's PhD thesis.

For $\mathcal{P}_1 = con(\mathcal{C}_1), \mathcal{P}_2 = con(\mathcal{C}_2) \in \mathbb{P}_n$ the widening $\mathcal{P}' = \mathcal{P}_1 \nabla \mathcal{P}_2$ is obtained by assigning to $\mathcal{P}' = con(\mathcal{C}')$ all the constraints of $\mathcal{C}_1$ excluding all the inequalities not satisfied by $\mathcal{C}_2$. This meets both requirements for a widening operator: $\mathcal{P}_1 \subseteq \mathcal{P}_2$ and both are contained in $\mathcal{P}'$ and since $|\mathcal{C}'| \leq |\mathcal{C}_1|$ widening cannot be applied indefinitely without converging.

This widening operator is generally referred to as the *standard widening* and few attempts have been made to improve the operator itself. Recently Bagnara *et al.* [5] suggested a framework for constructing improved widening operators for the convex polyhedral domain. Where the standard widening operator is restricted to only looking at the constraint representation of a polyhedron, the new operators can be based on both the constraint and the parametric representation. In short, the method allows operators that otherwise do not meet the widening operator criteria to be used, while still ensuring termination. The resulting widening operator is not guaranteed to be more precise than the standard widening but it is never *less* precise.

Narrowing for Convex Polyhedra

Narrowing in the convex polyhedra domain has up until recently been an unexplored area [5, 55, 56], though it had previously been suggested that narrowing for the convex polyhedral domain could produce more precision [12].

Strictly speaking a narrowing operator must ensure eventual stabilisation according to the second requirement of Definition 29 on page 31. Quoting Cousot and Cousot in [54]:

> A simple narrowing is obtained by limiting the length of the decreasing iteration sequence to some $k \geq 1$ (experience shows that $k > 1$ often brings no significant improvement).

This would indicate that it might be sufficient to look at the first requirement of Definition 29, and settle for a relatively low number of iterations, ignoring the convergence requirement.

We propose the use of the greatest lower bound (glb) as a simple narrowing operator. For polyhedra the glb is the intersection operation. The use of the intersection operator is safe, in the sense that it yields a safe approximation of the least fix point but it does not guarantee convergence.

The glb operator, in this case intersection, is applied at each program point, p, to the polyhedron derived from the widened fix point computations, $\mathcal{P}_{p_\nabla}$, and the polyhedron derived from applying the semantic transfer function f to the program point p. If narrowing results in a more precise approximation for some program point p, then the narrowing procedure may be reiterated to ensure that the most precise approximation is used for narrowing at those program points that depends on p in the call graph of the analysed program.

4.2.3   Other widening strategies

As mentioned earlier, few attempts have been made to introduce new widening operators. Work has focused on different widening strategies for improving precision - or rather minimising loss of precision.

For the Convex Polyhedral domain a few improved widening strategies have been proposed.

Delayed Widening

This technique is quite simple. The application of the widening operator is delayed for a number of iterations. This will allow the analyser to build a set of more explicit constraints to widen on [78] and produce more precise analysis results as shown in [12].

Widening with thresholds/widening up-to

This technique was described for the interval domain in [16] and the convex polyhedral domain in [80]. In the *interval domain* variables in a program are abstracted by a tuple $[a, b]$ where $a \leq b$ indicating a lower and upper bound on values that abstracted variables can have during execution of the program. The need for this arose from the fact that narrowing for the interval domain would not recover bounds on loop variables if the exit condition for the loop contained disequalities A finite set of *threshold* values, T, including $-\infty$ and $+\infty$ are used to find better approximations of lower or upper bounds, than simply $-\infty$ and $+\infty$. Widening with thresholds for the interval domain is defined as

$$[a, b] \; \nabla_T \; [a', b'] = [a_l, b_h] \text{ where } \quad a_l = \begin{cases} a & \text{if } a' \geq a \\ max\{l \in T | l \leq a'\} & \text{otherwise} \end{cases}$$

$$\text{and} \quad b_h = \begin{cases} b & \text{if } b' \leq b \\ min\{h \in T | h \geq b'\} & \text{otherwise} \end{cases}$$

The set of thresholds may be derived from the analysed program itself [80] or otherwise specified by the user.

We will implement this in the convex polyhedron analyser for constraint logic programs described later. We suggest initially to derive the set of widening-up-to constraints, which we will call the *bounding constraints*, from the analysed program itself. For constraint logic programs, this set can easily be obtained for each predicate in the program, by taking the convex hull of the polyhedra derived from intersecting the constraints on the built-in arithmetic predicates occurring in each clause body.

Definition 36 (Clause constraints). *The set of clause constraints for a program P is defined as:*

$$\mathcal{C}_P = \left\{ A \leftarrow \mathcal{C} \middle| \begin{array}{l} A \leftarrow B_1, \ldots, B_n \in P \\ \mathcal{C} = \bigcup \left\{ B_{lin} \middle| \begin{array}{l} i \in \{1, \ldots, n\} \\ B_{lin} = linearise(B_i) \end{array} \right\} \end{array} \right\}$$

where $linearise(B)$ returns the linear approximation of an atom B if B is a non-linear built-in arithmetic predicate for which a safe linear approximation can be supplied. Built-in linear predicates such as '+' are not modified by this procedure. If no linear approximation can be derived for the atom B it will be unconstrained (i.e. the universe).

Definition 37 (Bounding constraints). *The bounding constraints for a program P is defined as:*

$$\mathcal{C}_P^{\hat{\nabla}} = \left\{ p \leftarrow \mathcal{C} \middle| \begin{array}{l} p \in \Pi_P \\ \mathcal{C} = \biguplus p \leftarrow C \in \mathcal{C}_P \end{array} \right\}$$

Widening with Landmarks

The widening with landmarks [142] shares some common traits with widening with thresholds. Where upper or lower bounds would be lost using standard widening, they can in some cases be recovered using narrowing. The widening with landmarks is a refinement of widening with thresholds, that will produce results precise enough that narrowing would not be needed to recover lost bounds. This technique doesn't throw away constraints that are not satisfied when widened, such as the standard widening does, but on the basis of these unsatisfiable constraints, it estimates which constraint(s) to widen up-to, that would result in the least loss of precision.

Look-ahead widening

The lookahead widening [78] is a recent method. Two polyhedra are used for abstracting each program point; a main and a pilot polyhedron. Widening and subsequent narrowing are only performed on the pilot polyhedron. The program is evaluated with respect to the main polyhedron, and program points that are not reachable under the main polyhedron are ignored. Once the pilot stabilises it is promoted to a main polyhedron and the program is reevaluated. This technique tries to solve the situations where during a loop a variable may be either increasing or decreasing. Widening in this situation may cause a loss of both upper and lower bounds. The widening with threshold as described in [80] would infer widening

thresholds for single variables. The look ahead widening would make it possible to propagate upper and lower bounds expressed with more than one variable.

The next section describes our implementation of a NNC polyhedra based convex polyhedron analyser for constraint logic programs.

4.3   Convex Polyhedron Analyser for CLP

For a convex polyhedron analyser a few polyhedra operations are required. These are projection, emptiness checking, entailment testing, the forming of convex hulls and some means of ensuring termination of analysis, typically widening. Programming libraries implementing these operations are readily available.

4.3.1   Parma Polyhedra Library

The Parma Polyhedra Library (PPL) is a programming library targeted especially at analysis and verification [8, 7]. It implements the operations needed for a convex polyhedron analyser and it has interfaces for a variety of programming languages including Ciao Prolog [29]. In addition it supports operations on not necessarily closed polyhedra [6, 8].

It supports both the standard widening and the improved widening mentioned in Section 4.2.2. They are called *H79* and *BHRZ03* in the PPL library.

The library is developed at the Department of Mathematics, University of Parma, Italy and the School of Computing, University of Leeds, United Kingdom. Development began in 2001 and the library is under further development.

Other libraries for manipulating polyhedra exist, such as the NewPolka library[1]. This library has C and OCaml interfaces but no Prolog interface. It also has some support for NNC polyhedra. The library is now integrated into the *Apron numerical abstract domain library*[2].

PolyLib[3] is another library that has been around since 1993 and is also under current development. This too is lacking a Prolog interface and support for NNC polyhedra.

---

[1]http://pop-art.inrialpes.fr/people/bjeannet/newpolka/index.html
[2]http://apron.cri.ensmp.fr/library/
[3]http://icps.u-strasbg.fr/PolyLib/

4.3.2 Semantics

This section will describe the semantics of the convex polyhedron analyser. To begin with it might be helpful to understand the semantics before the implementation is described. A concrete semantics and an abstract semantics will be defined. The concrete semantics is described similar to the T_P semantics mentioned back in Chapter 2. The concrete domain will be over the set of all formulas, $Atom \leftarrow Con$, where Con is a constraint system.

Definition 38 (Concrete Semantics). *The immediate consequence operator is defined as:*

$$T_P^{\mathcal{C}} : 2^{Atom \leftarrow Con} \rightarrow 2^{Atom \leftarrow Con}$$

$$T_P^{\mathcal{C}}(I) = \left\{ A \leftarrow \mathcal{C} \left| \begin{array}{l} A \leftarrow B_1, \ldots, B_n \in P \\ \{A_1 \leftarrow \mathcal{C}_1, \ldots, A_n \leftarrow \mathcal{C}_n\} \in I \\ and \ \exists \ substitution \ \theta \ such \ that \\ mgu((B_1, \ldots, B_n), (A_1, \ldots, A_n)) = \theta \\ \mathcal{C}' = \bigcup_{i=1,\ldots,n} \{\mathcal{C}_i \theta\} \\ \mathcal{C}' \not\equiv false \\ \mathcal{C} = project(\mathcal{C}', Var(\mathcal{C}') \setminus Var(A)) \end{array} \right. \right\}$$

$$M^{\mathcal{C}}[\![P]\!] = \mathsf{lfp}(T_P^{\mathcal{C}})$$

It is assumed that all built-in constraint predicates are in I (for example X < Y :- X < Y $\in I$). We also assume some adequate satisfiability and projection algorithm for constraints appearing in the program exists. The set of constrained atoms $\{\langle B_1 \leftarrow \mathcal{C}_1 \rangle, \ldots, \langle B_n \leftarrow \mathcal{C}_n \rangle\}$ is renamed apart. The domain of the interpretation is the powerset of the set of facts of the form $p(X_1, ..., X_n) \leftarrow \mathcal{C}$, where p is a predicate in P and $\mathcal{C}$ is a set of constraints over $X_1, ..., X_n$.

The abstract semantics is defined next.

Definition 39 (Abstract Semantics). *Let $T_P^{\mathcal{C}}$ be the concrete semantics function, then the abstract semantics of the program P is the fix point defined as*

$$T_P^{\mathbb{P}} : 2^{Atom \leftarrow Con} \rightarrow 2^{Atom \leftarrow Con}$$
$$T_P^{\mathbb{P}}(I) = I \uplus T_P^{\mathcal{C}}(I)$$

$$M^{\mathbb{P}}[\![P]\!] = \mathsf{lfp}(T_P^{\mathbb{P}})$$

where the convex hull operator is extended to operate on constrained atoms.

The domain of the abstract semantics is the powerset of the set of facts of the form $p(X_1, ..., X_n) \leftarrow \mathcal{P}^n$, where p is a predicate in P and $\mathcal{P}^n$ is a NNC convex polyhedron.

As stated previously in Section 4.2.2, when the abstract domain does not satisfy the ascending chain condition, as is the case for the convex polyhedral hull domain, some method of accelerating the fix point computations is required, such as widening. The application of the widening operator occurs at a specific set of program points, W, as outlined in Chapter 3. For a given set of widening points, W, a new widening sequence is defined. And as also previously mentioned, precision can be improved by delaying widening for $d \geq 0$ steps. This can be incorporated in the widening sequence.

Definition 40 (Delayed Widening Sequence). *Let $d \geq 0$ be the number of iterations to delay widening, let W be the set of program points at which to widen, and let $T_P^{\mathbb{P}}$ be the abstract semantics function, then the delayed widening sequence is defined as*

$$
\begin{array}{rcll}
S_0 & = & \bot & \\
S_{i+1} & = & T_P^{\mathbb{P}}(S_i) & \text{for } i \leq d \\
S_{i+1} & = & S_i \,\nabla_W\, T_P^{\mathbb{P}}(S_i) & \text{for } i > d
\end{array}
$$

The delayed widening sequence will reach a fix point after k iterations so $S_{k+1} = S_k$. The widening operator can be substituted with the widen-up-to operator, using, for example, the constraint representation of the bounding polyhedron, $\mathcal{P}^{\hat{\nabla}}$, as the widening up to constraints.

The results of the widening sequence can be refined by applying a narrowing operator. The narrowing sequence is defined as

Definition 41 (Narrowing sequence). *Let S_k be the fix point of the delayed widening sequence after k iterations, and let $l \geq 0$ be a given number of iterations to narrow, then the narrowing sequence is defined as*

$$
\begin{array}{rcll}
N_0 & = & S_k & \\
N_{i+1} & = & N_i \,\cap\, T_P^{\mathbb{P}}(N_i) & \text{for } i < l, i > 0
\end{array}
$$

The next section outlines the implementation.

4.3.3 Implementing the convex polyhedron analyser

Bottom up analysis

Our analyser is originally based on a bottom-up evaluator for logic programs, developed by Codish and Søndergaard [36, 41]. The naive bottom up interpreter is a small Prolog program, closely implementing the T_P semantics described in Section 2.2.1. An implementation of the bottom up evaluator is shown in Figure 4.4 on the facing page. It is assumed the program to be analysed is stored in facts that have the form $my\_clause(Head, Body)$. The analysis is initiated using the proposition tp as goal. A fix point is reached when the call to tp terminates. The model of the program is stored as facts of the form $fact(Head)$.

The evaluator uses the built-in *fail* predicate to force reevaluation of a given predicate until this predicate can no longer be proven. This evaluation strategy is not an efficient strategy but it is simple to implement. In the naive bottom-up evaluator the *operator* predicate attempts to prove some clause in the evaluated program from the asserted facts. If a clause can be proven it is asserted as a fact. The *cond_assert* predicate checks if a facts is an instance of some existing fact; if not, the fact is asserted and a flag is raised signifying that the program must be reevaluated. An iteration terminates when no new facts can be asserted. If a flag was raised a new iteration is computed. The program is evaluated iteratively until a fix point is reached. A fix point is reached, when no new heads can be asserted on some iteration.

Bottom up convex polyhedron analyser

For the convex polyhedron analyser the naive bottom up interpreter is modified so each fact is associated with a polyhedron. Thus the predicate fact has two arguments, an atom of the form $p(X_1, \ldots, X_n)$ and a set of constraints $\mathcal{C}$ over (some of) the variables $X_1, \ldots, X_n$.

To implement the (delayed) widening sequence two polyhedra are stored for each predicate; the polyhedron obtained in the previous iteration (called *oldfact*) and the polyhedron calculated in the current iteration (called *newfact*). The program to be analysed is as before assumed to be asserted as facts of the form $my\_clause(Head, Body)$. The heads are assumed to be normalised, i.e. a head is of the form $p(X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ are distinct variables. When the program clauses are asserted the variables occurring in the clause heads are systematically renamed so first argument becomes X_1, second argument is X_2 and so on. Any constraints appearing as an argument are replaced by an appropriately

$$tp \leftarrow iterate.$$

$$iterate \leftarrow operator, fail.$$
$$iterate \leftarrow retract(flag), iterate.$$
$$iterate.$$

$$raise\_flag \leftarrow (flag \rightarrow true; assert(flag)).$$

$$cond\_assert(F) \leftarrow \neg(numbervars(F, 0, \_), fact(F)),$$
$$assert(fact(F)), raise\_flag.$$

$$operator \leftarrow my\_clause(H, B), prove(B), cond\_assert(H).$$

$$prove([\,]).$$
$$prove([true]).$$
$$prove([B|Bs]) \leftarrow fact(B), prove(Bs).$$

Figure 4.4: Naive bottom up evaluator for logic programs [41]

named variable, and the constraint is added to the clause body, for instance the unit clause $p(0) \leftarrow$ would be transformed into the clause $p(X_1) \leftarrow X_1 = 0$. Similarly for variables appearing at more than one argument position, each occurrence is renamed and the relationship between the renamed arguments are added to the clause body, for example $p(Y, Y) \leftarrow$ becomes $p(X_1, X_2) \leftarrow X_1 = X_2$.

Figure 4.5 on page 53 shows the modified bottom up interpreter implementing the analyser. The convex polyhedron analyser differs from the naive evaluator in the *operator*-predicate, the stored facts and the widening and narrowing steps, plus the necessary polyhedron operations. The fix point is reached when an iteration step does not result in any polyhedra not already included in the polyhedra acquired in the previous iteration.

Solving clause bodies

The clause bodies can contain arithmetic expressions and calls to predicates defined in the program itself (user defined predicates). The constraint on the user defined predicates can be retrieved from the stored facts. The arithmetic expressions can be substituted with equivalent constraints. Non-linear operators such as multiplication must either be given a linear approximation or simply be ignored. The linearisation procedure will be explained in Section 4.3.4.

Projection on to clause head

The set of constraints for a given clause body is the union of the constraints on the user defined predicates and the constraints from the linearised expressions. If this set of constraints on the clause body is satisfiable, that is it forms a non-empty polyhedron, the next step is to project the polyhedron onto the head of the clause. This step is accomplished by the *formPolyhedron/3*-predicate in Figure 4.5 on the facing page. Projection will, as described in Section 4.1.3, reduce the dimensions of the polyhedron to the arity of the clause head. The PPL library provides this functionality.

Forming convex hull of new and old polyhedron

If the "reduced" polyhedron is entailed by the polyhedron already associated with the given predicate, no further steps are needed. The existing polyhedron will in this case either be the polyhedron from the previous iteration or a polyhedron derived in the current iteration for the same predicate. If it is not the case that the new polyhedron is entailed by an existing polyhedron, then the convex hull is formed between the existing convex polyhedron and the projected polyhedron. The new convex polyhedron - $\mathcal{P}' = \mathcal{P}_{existing} \uplus \mathcal{P}_{new}$ - is associated with the given clause head, replacing any existing polyhedron derived in the current iteration, but does not replace the polyhedron obtained in the previous iteration. The step is handled by the *cond_assert/2*-predicate shown in Figure 4.5 on the next page. The later widening step will need both the old polyhedron and the newly derived polyhedron.

Widening step

In each iteration the constraints are *widened* to ensure termination of the analysis. The analyser supports delayed widening, where widening is not applied until after a specified number of iterations have been executed and different widening operators.

When all clauses have been processed widening is applied to the polyhedra obtained in the previous iteration, stored in *oldfact*, and the polyhedra derived in the current iteration, stored in *newfact*, i.e. $\mathcal{P}' = \mathcal{P}_{old} \nabla_W \mathcal{P}_{new}$. Widening need not be applied to all predicates. In this case only a subset W of predicates are marked as widening points. These predicates are chosen according to the algorithm described later in Section 4.3.5. For those predicates chosen as widening points, the resulting polyhedron $\mathcal{P}'$ replaces $\mathcal{P}_{old}$ and $\mathcal{P}_{new}$ is discarded. For those predicates not chosen as widening points, $\mathcal{P}_{new}$ simply replaces $\mathcal{P}_{old}$. Figure 4.8 on page 55 shows this process. The predicate *widenfacts/0* will apply the predicate *widenhead/1* to all widening points and move any asserted *newfact/2* to *oldfact/2*. The predicate *widen/3* will apply the appropriate widening operator according to

$analyse \leftarrow assert(flag(first)), iterate, narrowfacts.$

$iterate \leftarrow my\_clause(H, B), operator(H, B), fail;$
$\qquad retractall(flag(first)).$
$iterate \leftarrow widenfacts, reiterate.$
$iterate.$

$reiterate \leftarrow flag(\_), iterate.$

$operator(H, B) \leftarrow (changed(B); flag(first)),$
$\qquad formPolyhedron(H, B, \mathcal{P}_H), cond\_assert(H, \mathcal{P}_H).$

$formPolyhedron(H, B, \mathcal{P}_H) \leftarrow prove(B, \mathcal{C}), linearise(\mathcal{C}, \mathcal{C}_{lin}),$
$\qquad satisfiable(\mathcal{C}_{lin}, \mathcal{P}), vardiff(H, \mathcal{C}_{lin}, H_{dim}), project(\mathcal{P}, H_{dim}, \mathcal{P}_H).$

$prove([\ ], [\ ]).$
$prove([B|B_s], [\mathcal{C}|\mathcal{C}_s]) \leftarrow constraint(B, \mathcal{C}), prove(B_s, \mathcal{C}_s).$
$prove([B|B_s], \mathcal{C}_s) \leftarrow getoldfact(B, \mathcal{C}_1), prove(B_s, \mathcal{C}_2),$
$\qquad append(\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_s).$

$getoldfact(B, \mathcal{C}) \leftarrow oldfact(B', H), getConstraints(H, \mathcal{C}'), melt((B', \mathcal{C}'), (B, \mathcal{C})).$

$fact(H, \mathcal{P}) \leftarrow newfact(H, \mathcal{P}); oldfact(H, \mathcal{P}).$

$cond\_assert(H, \mathcal{P}) \leftarrow \neg entailedby(H, \mathcal{P}), existingPolyhedron(H, \mathcal{P}_{old}),$
$\qquad convexhull(\mathcal{P}, \mathcal{P}_{old}, \mathcal{P}_{new}), assert(newfact(H, \mathcal{P}_{new})), raiseflag(H).$

$existingPolyhedron(H, \mathcal{P}) \leftarrow retract(newfact(H, \mathcal{P})); oldfact(H, \mathcal{P}).$
$existingPolyhedron(\_, empty).$

$entailedby(H, \mathcal{P}) \leftarrow fact(H, \mathcal{P}_{fact}), entails(\mathcal{P}_{fact}, \mathcal{P}).$

Figure 4.5: Bottom up convex polyhedron analyser for logic programs

$$constraint(X = Y, X = Y).$$
$$constraint(X =:= Y, X = Y).$$
$$constraint(X \ is \ Y, X = Y).$$
$$constraint(X < Y, X < Y).$$
$$constraint(X > Y, X > Y).$$
$$constraint(X >= Y, X >= Y).$$
$$constraint(X =< Y, X =< Y).$$
$$constraint(X = Y, X = Y).$$

$$constraint(X \backslash == Y, 0 = 0).$$
$$constraint(X = \backslash = Y, 0 = 0).$$
$$constraint(true, 0 = 0).$$
$$constraint(false, 1 = 0).$$

Figure 4.6: The *constraint* predicate turns body atoms into constraints. Here $0 = 0$ denotes a constraint that will be ignored (always satisfied) and $1 = 0$ denotes a false constraint that can never be satisfied.

$$raise\_flag(F) \leftarrow functor(F, Fn, N),$$
$$((nextflag(Fn/N) \rightarrow true); (assert(nextflag(Fn/N)))).$$

$$changed([\ ]) \leftarrow fail.$$
$$changed([B|B_s]) \leftarrow flagset(B).$$
$$changed([B|B_s]) \leftarrow \neg flagset(B), changed(B_s).$$

$$flagset(F) \leftarrow functor(F, Fn, N), currentflag(Fn/N).$$

Figure 4.7: Control predicates flag which clauses must be reevaluated in the next iteration and which clauses must be reevaluated in the current iteration. This is used for the semi-naive optimisation strategy we have implemented. Between iterations the *nextflag*-facts are retracted and asserted as *currentflag*-facts. In the naive bottom-up evaluator every clause was reevaluated in every iteration so a global flag would suffice. For the semi-naive strategy a flag must be maintained for each clause.

$$widenfacts \leftarrow widenpoint(H), newfact(H, \_), oldfact(H, \_),$$
$$widenhead(H), fail.$$
$$widenfacts \leftarrow retract(newfact(H, \mathcal{P})), retractall(oldfact(H, \_)),$$
$$assert(oldfact(H, \mathcal{P})), fail.$$
$$widenfacts.$$

$$widenhead(H) \leftarrow retract(newfact(H, \mathcal{P}_{new})), retract(oldfact(H, \mathcal{P}_{old})),$$
$$widen(\mathcal{P}_{new}, \mathcal{P}_{old}, \mathcal{P}'), assert(oldfact(H, \mathcal{P}')).$$

Figure 4.8: Widening in convex polyhedron analyser for logic programs.

the configuration of the analysis. For delayed widening, the procedure is extended with a fact for each widening point containing a counter that would be decremented each time *widen/3*-predicate is called until the counter reaches zero. The widening operator will only be applied if the counter has reached zero for that widening point.

Fix point detection

A new polyhedron will not necessarily be derived for all predicates. Those predicates for which $\mathcal{P}_{old}$ was replaced by a new polyhedron are flagged as modified. These flags served a dual purpose. They are used used both to detect when a fix point is reached and for the particular iteration strategy we have implemented. The iteration strategy will be described in Section 4.3.6.

Narrowing

When a fix point is reached, a number of narrowing iterations can be applied. This is shown in Figure 4.9 on the next page. The number of narrowing iterations that will be applied is specified before the analysis begins and stored as a fact. The *narrowfacts/0*-predicate is then called the number of times specified.

4.3.4   Linear approximation

The polyhedral operations implemented using the PPL library take only linear expressions as constraints, so each occurrence of a non-linear expression in a clause body must be approximated by a linear expression before the PPL operations can be invoked. Those expressions for which no linear approximation can be given, are simply ignored. This is safe to do, but can lead to significant loss of precision in the analysis. Linearising an expression is a recursive procedure, for example

$$narrow\,facts \leftarrow my\_clause(H, B), narrowoperator(H, B), fail.$$
$$narrow\,facts.$$

$$narrowoperator(H, B) \leftarrow formPolyhedron(H, B, \mathcal{P}_H),$$
$$narrow\_cond\_assert(H, \mathcal{P}_H).$$

$$narrow\_cond\_assert(H, \mathcal{P}) \leftarrow existingWidenedPolyhedron(H, \mathcal{P}_{old}),$$
$$intersect(\mathcal{P}, \mathcal{P}_{old}, \mathcal{P}_{new}), retractall(oldfact(H, \_)),$$
$$assert(oldfact(H, \mathcal{P}_{new})).$$

$$existingWidenedPolyhedron(H, \mathcal{P}) \leftarrow retract(oldfact(H, \mathcal{P})).$$
$$existingWidenedPolyhedron(\_, empty).$$

Figure 4.9: Narrowing in convex polyhedron analyser.

$$linearise([\,], [\,]).$$
$$linearise([\mathcal{C}|\mathcal{C}_s], [\mathcal{C}|\mathcal{C}'_s]) \leftarrow linearConstraint(\mathcal{C}), !, linearise(\mathcal{C}_s, \mathcal{C}'_s).$$
$$linearise([\mathcal{C}|\mathcal{C}_s], [\mathcal{C}'|\mathcal{C}'_s]) \leftarrow linearApproximation(\mathcal{C}, \mathcal{C}'), !, linearise(\mathcal{C}_s, \mathcal{C}'_s).$$
$$linearise([\_|\mathcal{C}_s], \mathcal{C}'_s) \leftarrow linearise(\mathcal{C}_s, \mathcal{C}'_s).$$

$$linearConstraint(X = Y) \leftarrow linearTerm(X), linearTerm(Y).$$

$$linearTerm(X) \leftarrow const(X), !.$$
$$linearTerm(X + Y) \leftarrow linearTerm(X), linearTerm(Y).$$

$$const(X) \leftarrow number(X), !.$$
$$const(X) \leftarrow var(X), !, fail.$$

$$linearApproximation(X = Y \backslash/Z, X =< Y + Z) \leftarrow !.$$

Figure 4.10: The *linearise* predicate will eliminate non-linear constraints from a given list of constraints. Some non-linear constraints may be given a safe linear approximation such as the boolean operator 'OR' ($\backslash/$) shown in this figure. The list of clauses shown here is only a small sample of the complete set of clauses in the analyser.

an operator such as '$+$' or '$-$' may only yield a linear result if both operands are linear expressions themselves.

Equalities and inequalites

These are straightforward. The terms `X is Y` and `X = Y` both translate into the constraints $X = Y$. Similarly for the inequalities, e.g. `X >= Y` translates into the constraints $X \geq Y$. Terms containing not-equals, e.g. `X =\= Y` are ignored since they have no linear approximation.

The right hand side of the term contains the operators that must also be approximated.

Approximation of arithmetic operations

For the arithmetic operations this is also straightforward. The addition and subtraction operations are both linear and e.g. `X is Y + Z` translates to $X = Y + Z$, if both Y and Z themselves are linear expressions.

The multiplication operator can also be approximated by a linear constraint if either both of the operands are constants, or if one operand is a constant and the other operator is linear. So the term `X is Y * Z` translates into $X = Y * Z$ if either Y or Z (or both) are constant. The recursive evaluation of the expression will ensure that the result is only linear if the non-constant operand is linear.

At present, expressions that are not linear are simply ignored, so that their variables are completely unconstrained. Figure 4.10 on the facing page shows the implementation of the linearise procedure.

4.3.5 Widening Points

For the convex polyhedral analyser for constraint logic programs we have constructed our own algorithm for finding widening points. A few properties differentiate programs "in general" from the programs that will later be analysed using the convex polyhedron analyser. Not all program will have a uniquely identified "first" program point. This is for instance the case for logic programs.

The Control Flow Graph (CFG) of a program is a graph representation of which program points are immediately reachable from any other program point. Normally all components of this graph would be connected unless the program for instance contains dead code. For a logic program the equivalent to a CFG is a *predicate call graph* with a node for every predicate in the program and an edge from the node for a predicate p to the node for a predicate q if there exists a clause of p with a call to q in its body. For logic programs the goal may be the only formula that connects parts of the program.

Typically the predicate call graph for a logic program will contain many small strongly connected components. Predicates are typically relatively small and directly recursive. Imperative programs on the other hand may contain fewer, larger components; especially if for instance goto instructions are used.

Our algorithm must work for CFGs with more than one entry point and more than one component. The last section of this thesis will describe assembler programs modelled as constraint logic programs; these logic programs will have a CFG resembling that of an imperative language. It can be demonstrated that for these programs, the depth first numbering, the retreating edges method and the strongly connected component method suggested by Bourdoncle [21] all produce a higher number of widening points. The convex polyhedron analyser will be applied to these programs, in addition to being applied to more "traditional" CLP programs. So our algorithm must provide a good set of widening points for both types of programs.

Cut-loop algorithm for computing widening points

The *Minimal Feedback Vertex Set* problem deals with any directed graph, from the sparsely connected to the densely connected graphs. The Control Flow Graphs of typical programs will be sparsely connected graphs containing some, possibly nested, loops. We will try to construct an algorithm of low complexity that will produce sets of widening points that are smaller than or equal to those produced by the simple *retreating edges* method of linear complexity described in [21]. A spanning tree of a graph can be constructed by traversing the graph. During traversal a retreating edge is found when an edge from a decendant to an ancestor is encountered. By identifying the retreating edges in the CFG for a program, the loops in the program are identified.

Definition 42. $G = (N, E)$ *is a directed graph with nodes* N *and edges* $E \subseteq N \times N$.

Definition 43. *The set of immediate predecessors of a node* n *is denoted*

$$Pred(n) \stackrel{def}{=} \{m | (m, n) \in E\}$$

Definition 44. *The set of immediate successors of a node* n *is denoted*

$$Succ(n) \stackrel{def}{=} \{m | (n, m) \in E\}$$

Definition 45. *A path* p *from node* n_1 *to* n_k *is a sequence of nodes such that* $(n_i, n_{i+1}) \in E$ *for* $1 \leq i \leq k - 1$.
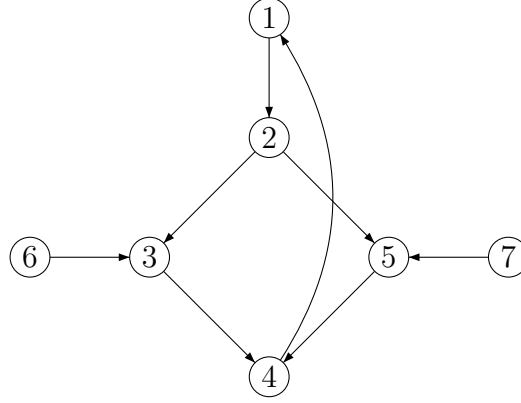
Figure 4.11: Directed Graph with multiple entry nodes, e.g. $\{1, 6, 7\}$.

Definition 46. *A loop l is a path $[n_1, \ldots, n_k]$ with $k \geq 1$, where there exists an edge $(n_k, n_1) \in E$.*

The procedure is shown in Algorithm 2 on the next page. The graph is traversed depth or breadth first from an entry node n_1. The currently visited node is denoted n_c. A loop is detected if $\exists n_i \in [n_1, \ldots, n_c]$ such that $n_i \in Succ(n_c)$. The loop is the path $[n_i, \ldots, n_c]$ which is a subset of the path $[n_1, \ldots, n_c]$. These loops are stored for later processing. The nodes are marked as visited when they are traversed. If there are still unvisited nodes left in the graph upon completion of the traversal of n_1, a new entry node n_1 is selected among the not yet visited nodes, and a new traversal is started from this node.

When the traversals are completed, i.e. all nodes are marked visited, widening points are selected from N, by choosing the node that cuts the highest number of loops, and removing those loops that are now cut by a widening point. This procedure continues until all loops are removed.

Applying this algorithm to the graph shown previously in Figure 3.2 on page 36 produces $W = \{3\}$. Figure 4.11 shows a graph that produces different results using depth first numbering or retreating edge detection, depending on which node is selected as the first entry node. Depth first numbering, with nodes chosen in the order shown in the figure, would pick $W = \{1, 3, 4, 5\}$. The retreating edge detection would pick $W = \{3, 4\}$ or $W = \{4, 5\}$ if traversal starts at either node 6 or 7. Out cut-loop algorithm would pick $W = \{1\}$, for any node chosen as the entry node.

Algorithm 2 Find widening points using cut-loops (not optimised)

Input: Graph $G = (N, E)$
Output: Wideningpoints $W \subseteq N$

Begin
   $loops = \emptyset$
   for $n \in N$
     traverse$(n, [\ ]_{ancestors})$
   $W = \emptyset$
   while $loops \neq \emptyset$
     for $n \in N$
       $loopCount[n] = |\{l | l \in loops, n \in l\}|$
     $candidates = \{n \in N | loopCount[n] = max_{j \in N}(loopCount[j])\}$
     select $wp \in candidates$
     $loops = loops \setminus \{l \in loops | wp \in l\}$
     $W = W \cup \{wp\}$
End

traverse$(n, n_{ancestors})$ {
  if visited(n, G) then
    if $n \in n_{ancestors}$ then
      $loop = $ path from $n \in n_{ancestors}$ to head of $n_{ancestors}$
      $loops = loops \cup loop$
    endif
  else
    markVisited(n, G)
    for $n_s \in Succ(n, G)$
      traverse$(n_s, [n | n_{ancestors}])$
  endif
}

4.3.6 Iteration Strategy

As previously stated any chaotic iteration strategy can be used to solve the system of semantic equations, $X = F(X)$, for monotone functions over complete lattices. Bourdoncle suggested two strategies based on weak topological ordering. Both of these strategies rely on an analysis of the control flow graph to determine the components to iterate over, and once generated these components remain static.

This section will describe a dynamic iteration strategy inspired by work in related fields dealing with some of the same problems - finding a fix point using as few computations as possible. Which strategy is better can depend on which problem is being solved [101].

The automatically generated logic programs derived from imperative assembly languages, will contain long chains of mutually-recursive predicates. This would result in a low number of strongly connected components in the Control Flow Graph.

For handwritten logic programs on the other hand, the set of strongly connected components would typically contain many small components. Each component can be iterated over separately until a fix point is found for the component.

Applying iteration strategies based on dividing the program into components, such as strongly connected components or weak topological orderings, might prove less suitable for our use. If the program contains only a few large component, iterating over an entire component may still be inefficient.

A naïve algorithm for solving a system of semantic equations as shown in Equation 3.1 on page 33, would be to recompute all equations in every iteration. The first observation would be that for the equations $x_i = f_i()$ where f_i has an empty set of variables that it depends on, the function would return the same result in every iteration. In the corresponding dependency graph for $X = F(X)$ these would be the nodes with an empty set of predecessors. A second observation would be that for those functions f_i where the set of arguments it depends on did not change in the previous iteration, these functions would also return the same result in the current iteration as it did in the previous iteration. Recomputing these equations would be redundant. Restricting the set of equations to recompute in each iteration to only those that depends on results that changed in previous iteration, can be described as a *semi naive optimisation strategy* [146].

4.3.7 Narrowing

The simple narrowing method implemented does not guarantee that a fix point is reached for the narrowing iterations. A limit on the number of iterations to compute must be specified in advance. A narrowing iteration proceeds much like a normal fix point iteration. The convex hull for each predicate is formed from the

polyhedra approximating each clause body. In each clause body the intersection of the polyhedra approximating the calls to user defined predicates is intersected with the polyhedron approximating the built-in arithmetic predicates including constraints that occurs in the clause. The resulting polyhedron for each predicate is intersected with the polyhedron obtained after a fix point was reached. Primarily this will recover lost upper or lower bounds.

4.3.8 Widen up to

To use *widening up-to* supported by the PPL library a set of constraints to widen up-to must be supplied to the widening operator. This set of constraints can easily be obtained from the bounding constraints obtained as described in Section 4.2.3. In the PPL library different versions of the widening up-to exists. One is the "standard" widening up-to, called *limited extrapolation* and another is the *bounded extrapolation* that includes a few "stop points" - $\{-2, -1, 0, 1, 2\}$ - that are selected as typical lower and upper bounds that can be recovered. These stop points are hardcoded in the library[4].

We have added the option to manually add these stop points to the program. A special purpose predicate *invariant/1* can be used to add one or more stop points to widen up-to for a specific user defined predicate, for example "*invariant*$(p(I, J)) \leftarrow I =< 100$." will add a suggested upper bound of 100 to the first argument of $p/2$. Some of these invariants may turn out to be invalid and will be ignored by the the widening up-to operator. Once a fix point is found it is easy to check which suggested invariants were valid. Simply form a polyhedron from the invariant and check whether the fix point polyhedron obtained for the user defined predicate polyhedron is entailed by the invariant polyhedron.

Example 8. *The program shown in Figure 4.12 on the facing page is an example of a program where widening would lose both upper and lower bounds, i.e. Y is unconstrained. The simple narrowing can in this case only recover lower bound for $down(Y) \leftarrow Y > 0$, and upper bound for $up(Y) \leftarrow Y < 100$. Widening up to the bounding polyhedron results in the precise approximation $down(Y) \leftarrow 0 < Y < 101$, but the approximation of $up(Y)$ remains the same, $Y < 100$. Adding stop points, such as the ones built into PPL ($\{-2, -1, 0, 1, 2\}$) results in the precise approximation for $up(Y) \leftarrow -1 < Y < 100$.*

The resulting approximation of the program is

$$
\begin{aligned}
up(Y) &\leftarrow -1 < Y < 100 \\
down(Y) &\leftarrow 0 < Y < 101
\end{aligned}
$$

---

[4]E-mail exchange with Roberto Bagnara revealed this; it is missing from the documentation of the PPL library.

$$
\begin{aligned}
up(50) \quad &\leftarrow \\
up(Y) \quad &\leftarrow \quad Y < 100, Y \;=\; Y1 + 1, up(Y1) \\
up(Y) \quad &\leftarrow \quad Y =< 0, Y \;=\; Y1 - 1, down(Y1) \\[1em]
down(Y) \;&\leftarrow \quad Y >= 100, Y \;=\; Y1 + 1, up(Y1) \\
down(Y) \;&\leftarrow \quad Y > 0, Y \;=\; Y1 - 1, down(Y1)
\end{aligned}
$$

Figure 4.12: Simple program modelling up/down behavior of a variable.

The bounding constraints for the predicates are the following

$$
\begin{aligned}
\mathcal{C}^{\hat{\nabla}}(up(Y)) \quad &= \quad Y < 100 \\
\mathcal{C}^{\hat{\nabla}}(down(Y)) \;&= \quad Y > 0
\end{aligned}
$$

□

4.3.9 Implementation details

The analyser is implemented in Ciao Prolog [29]. The script facility of Ciao allows the program to be executed from a command line. The specific version of the PPL library used is version 0.9. The input program must be a CLP($\mathcal{N}$) program containing only numerical terms. An example of such a program is shown in Example 6 on page 38.

4.3.10 Comparison with existing analysers

A convex hull analyser for CLP($\mathcal{R}$) was reported in [12, 14]. This analyser was compared with a previous analyser, selecting the tricky test cases for comparison. The results from their analyser was reported as fractions, making them easier to read, e.g. $0.666667x \leq y \wedge y \leq 1.5x$ is reported as $\frac{2}{3}x \leq y, y \leq \frac{3}{2}x$. Output from our analyser will be rewritten for easy comparison between the two analysers. Both analysers support delayed widening but only our analyser supports the simple narrowing. Results will be reported for "best" configuration.

The results reported by Benoy and King in their LOPSTR'96 article [12] were also reported in Benoy's PhD thesis [14] with a few modifications. A few redundant constraints were eliminated and what appears to be a typo was fixed - for the *perm* program, the constraints for the *del* were reported as $z = y + 1$, but it should be $z = y - 1$. Comparison will therefore be made with the results from Benoy's thesis.

For our analyser, the selected widening operator is specified along with the number of iterations that widening is delayed (d) and the number of narrowing iterations (n).

Table 4.1 shows the first set of results. Our Convex Polyhedron Analyser is abbreviated CPA. For all programs we obtain equally precise approximations. For the last program, *split*, we eliminate the redundant constraint $0 \leq y$. Also note that this is an example of a program where delayed widening is needed. Table 4.2 shows the last set of results for comparison.

Program	Abstract Program	Benoy and King	CPA
$leq(X, X).$ $leq(X, succ(Y)) ←$ $\quad leq(X, Y).$	$leq^A(X, X) ←$ $\quad 0 ≤ X.$ $leq^A(X, 1 + Y) ←$ $\quad 0 ≤ X, 0 ≤ Y$ $\quad leq^A(X, Y).$	$leq^A(x, y) ←$ $\quad \{x ≤ y,$ $\quad 0 ≤ x\}$	$∇ = H79,$ $d = 0, n = 0$ $\overline{\qquad\qquad}$ $leq^A(x, y) ←$ $\quad \{x ≤ y,$ $\quad 0 ≤ x\}$
$trd([], []).$ $trd([\_,\_,|X], [\_,\_,|Y]) ←$ $\quad trd(X, Y).$ $trd([\_,\_,\_|X], [\_,\_,|Y]) ←$ $\quad trd(X, Y).$	$trd^A(0, 0).$ $trd^A(2 + X, 3 + Y) ←$ $\quad trd^A(X, Y).$ $trd^A(3 + X, 2 + Y) ←$ $\quad trd^A(X, Y).$	$trd^A(x, y) ←$ $\quad \{\frac{2}{3}x ≤ y,$ $\quad y ≤ \frac{3}{2}x\}$	$∇ = H79,$ $d = 0, n = 0$ $\overline{\qquad\qquad}$ $trd^A(x, y) ←$ $\quad \{\frac{2}{3}x ≤ y,$ $\quad y ≤ \frac{3}{2}x\}$
$perm([], []).$ $perm([H|T], [A|P]) ←$ $\quad del(A, [H|T], L),$ $\quad perm(L, P).$	$perm^A(0, 0).$ $perm^A(1 + T, 1 + P) ←$ $\quad del^A(\_, 1 + T, L),$ $\quad perm^A(L, P).$	$perm^A(x, y) ←$ $\quad \{y = x,$ $\quad 0 ≤ x\}$	$∇ = H79,$ $d = 0, n = 0$ $\overline{\qquad\qquad}$ $perm^A(x, y) ←$ $\quad \{y = x,$ $\quad 0 ≤ x\}$
$del(X, [X|Y], Y).$ $del(U, [Y|Z], [Y|W]) ←$ $\quad del(X, Z, W).$	$del^A(\_, 1 + Y, Y).$ $del^A(\_, 1 + Z, 1 + W) ←$ $\quad del^A(\_, Z, W).$	$del^A(x, y, z) ←$ $\quad \{z = y - 1\}$	$del^A(x, y, z) ←$ $\quad \{z = y - 1\}$
$split([], [], []).$ $split([X|Xs], [X|Os], Es) ←$ $\quad split(Xs, Es, Os).$	$split^A(0, 0, 0).$ $split^A(1 + Xs, 1 + Os, Es) ←$ $\quad split^A(Xs, Es, Os).$	$split^A(x, y, z) ←$ $\quad \{x = y + z,$ $\quad y ≤ x,$ $\quad y - \frac{1}{2}x ≤ \frac{1}{2},$ $\quad \frac{1}{2}x ≤ y,$ $\quad 0 ≤ y\}$	$∇ = H79,$ $d = 2, n = 0$ $\overline{\qquad\qquad}$ $split^A(x, y, z) ←$ $\quad \{x = y + z,$ $\quad y ≤ x,$ $\quad y - \frac{1}{2}x ≤ \frac{1}{2},$ $\quad \frac{1}{2}x ≤ y\}$

Table 4.1: Comparing PPL based convex polyhedron analyser with Benoy & King's convex hull analyser

Program	Abstract Program	Benoy	CPA
$quicksort([\,],[\,]).$ $quicksort([X\|Xs],S) \leftarrow$ $\quad part(X,Xs,L,G),$ $\quad quicksort(L,SL),$ $\quad quicksort(G,SG),$ $\quad append(SL,[X\|SG],S).$	$quicksort^A(0,0).$ $quicksort^A(1+Xs,S) \leftarrow$ $\quad part^A(\_,Xs,L,G),$ $\quad quicksort^A(L,SL),$ $\quad quicksort^A(G,SG),$ $\quad append^A(SL,1+SG,S).$	$quicksort^A(x,y) \leftarrow$ $\quad \{x = y,$ $\quad 0 \leq x\}$ $part^A(w,x,y,z) \leftarrow$ $\quad \{x = y + z,$ $\quad y \leq x,$ $\quad 0 \leq x\}$ $append^A(x,y,z) \leftarrow$ $\quad \{z = x + y,$ $\quad 0 \leq x\}$	$\nabla = H79,$ $d = 0, n = 1$ $\rule{3cm}{0.4pt}$ $quicksort^A(x,y) \leftarrow$ $\quad \{x = y,$ $\quad 0 \leq x\}$ $part^A(w,x,y,z) \leftarrow$ $\quad \{x = y + z,$ $\quad z \leq x,$ $\quad 0 \leq z\}$ $append^A(x,y,z) \leftarrow$ $\quad \{z = x + y,$ $\quad y \leq z\}$
$part(\_,[\,],[\,],[\,]).$ $part(X,[Y\|Ys],L,[Y\|G]) \leftarrow$ $\quad X \leq Y,$ $\quad part(X,Ys,L,G).$ $part(X,[Y\|Ys],[Y\|L],G) \leftarrow$ $\quad Y \leq X,$ $\quad part(X,Ys,L,G).$	$part^A(\_,0,0,0).$ $part^A(X,1+Ys,L,1+G) \leftarrow$ $\quad part^A(X,Ys,L,G).$ $part^A(X,1+Ys,1+L,G) \leftarrow$ $\quad part^A(X,Ys,L,G).$		
$append([\,],Y,Y).$ $append([X\|Xs],Y,[X\|Zs]) \leftarrow$ $\quad append(Xs,Y,Zs).$	$append^A(0,Y,Y).$ $append^A(1+Xs,Y,1+Zs) \leftarrow$ $\quad append^A(Xs,Y,Zs).$		

Table 4.2: Comparing Benoy's quicksort results.

Figure 4.13: Selecting analysis parameters

4.3.11   Web Interface

A web interface has been added to the convex polyhedron analyser. The purpose of the interface is to demonstate the facilities of the analyser and to allow quick analysis of programs. The analyser by Benoy and King used in the previous section for comparison was not available for experimentation. As a result the comparison of the analysers have only been based on the relatively small set of example programs used for the Benoy and King analyser. A publicly available version of the analyser will allow other researchers to easily compare results for specific test programs between analysers.

The tool is built on PHP and XML. The tool has two pages; a front page where the programmer can upload a program for analysis and select analysis parameters and a result page where the analysis results are shown. A screen shot of the front page is shown in Figure 4.13. A test program, the append program with list length norm applied, has been selected.

Figure 4.14 shows the result page for the append program. The constraints are shown after a fix point is found using widening to accelerate convergence, and the constraints are shown after narrowing has been applied.

Demonstration of tool is available online at

67

Convex Hull Analyser for logic programs

2. Predicate results

[a] Constraints after widening: [1*B>=0,1*A>=0,1*A+1*B+ -1*C=0]
Constraints after narrowing: [-1*B+1*C>=0,1*B>=0,1*A+1*B+ -1*C=0]

    B=C,
    B>=0.
[a] applen(A,B,C):-
    applen(D,B,E),
    A is D+1,
    C is E+1.
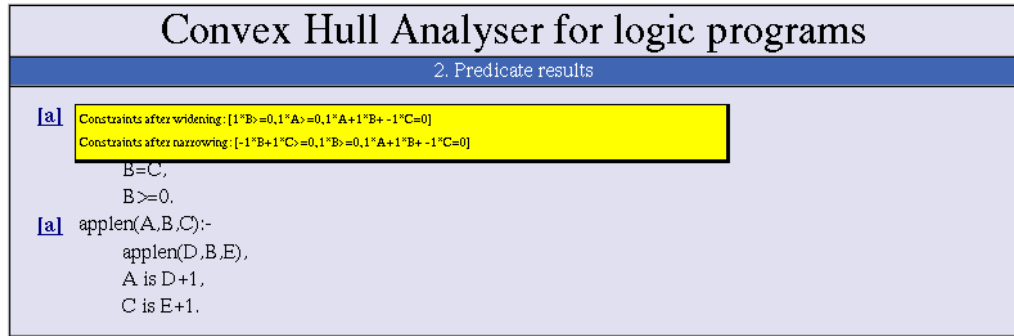
Figure 4.14: Displaying analysis result

`http://wagner.ruc.dk/CHA/`

The design of the web interface originates from WebLogen[5] [106], a web interface for the Logen program specialiser by Michael Leuschel at Heinrich-Heine-Universität Düsseldorf, with whom we have cooperated in developing the web interface for the ASAP online tools[6].

---

[5]`http://www.stups.uni-duesseldorf.de/~pe/weblogen/`
[6]`http://www.stups.uni-duesseldorf.de/~asap/asap-online-demo/`

Chapter 5

Type Analysis Tool for Logic Programming

Types are widely used in programming languages. Generally speaking types are an abstraction of a program's meaning. Associating a particular type with a particular program construct is a way of describing which values can possibly occur and which values can definitely not occur in that program construct.

At the top level, types in programming languages come in two flavors. First there are the *prescriptive* types that are supplied by the programmer as a partial specification of the intended meaning of the program. Secondly there are the *descriptive* types which are derived from the program itself and approximate the actual meaning of that particular program.

The prescriptive types are present in most modern languages and their use enhances programmer productivity and the quality and efficiency of the developed software. Typed programs can be *type checked* - in a sense checked for consistency with the intended meaning. Type errors can be flagged to the programmer if the written program code is not consistent with the supplied type specifications.

Descriptive types can be used in both typed and untyped programming languages. Although there is not the same notion of "badly typed" programs with respect to descriptive types, they can, like the prescriptive types, be used to detect errors in programs. Descriptive type languages can describe values that are sometimes not expressible in prescriptive type languages. These features sometimes allow more precise assignment of types than is possible with prescriptive types.

Chapter Overview:

The description of types in this chapter is based on regular type rules.

   - Section 5.1 gives a short introduction to notation and definitions.

- Section 5.2 gives an introduction to Finite Tree Automata (FTA).

- Section 5.3 describes FTAs as abstract domains for abstract interpretation of logic programs.

- In Section 5.4 a method for building analysis domains from any FTA on a given program's signature will be described. This method is based on the transformation of the FTA to a pre-interpretation for the program. From the programmer's point of view, the type specifications remain regular expressions that are easy to handle.

- An improved algorithm for determinising types is described in Section 5.5.

- Examples of applications for the deterministic regular types is given in Section 5.6.

- Implementation issues relating to applying the derived pre-interpretation to program analysis is described in Section 5.7.

- Experiments with the improved determinisation algorithm are described in Section 5.8.

- At the end of this chapter a tool for applying inferred types prescriptively will be described.

5.1 Regular Type Definitions

Logic programming languages are in general untyped. Exceptions exist such as Mercury [143]. Type related programming errors in typed languages can be flagged at compile time, informing the programmer of the error without the need for executing the program. For an untyped logic language such as Prolog this is not possible, which leads to many type related bugs during development of Prolog programs [81]. A classic Prolog example is the *append* program.

Example 9. *The typical implementation for appending lists in Prolog is shown below.*

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

The goal $append([\,],foo,foo)$ *would succeed though this behavior would not be expected since the term foo is not a list. A more proper definition of* append *would be something like the following.*

```
append([],Ys,Ys) :- list(Ys).
append([X|Xs],Ys,[X|Zs]) :-
       append(Xs,Ys,Zs).

list([]).
list([_|Ys]) :- list(Ys).
```
□

A *typed logic program* consists of a logic program, a type definition and a *predicate signature* for each predicate in the program. The predicate signature declares a type for each argument of each predicate. Given such a typed logic program a type checker can verify whether the program is *well-typed* or not. It is well-typed if the actual parameters passed to the predicates are instances of the predicate signature [25, 88]. Inferring types for a logic program boils down to deriving this typed logic program. Commonly, inferred types are less precise than declared types.

Types can be specified using *regular type rules*.

Definition 47 (Regular Type Rule). *Let Σ be a set of function symbols where every symbols is assigned an arity $n \in \mathbb{N}_0$. A regular type rule for a type t is defined by*

$$t \longrightarrow f_1(t_{1,1}, \ldots, t_{1,n_1}); \ \ldots \ ; f_m(t_{m,1}, \ldots, t_{m,n_m})$$

where $f_i \in \Sigma$ for $1 \leq i \leq m$ and n_i is the arity of f_i and for $1 \leq j \leq n_i$, $t_{i,j}$ is itself a regular type.

Example 10. *The list type would be defined using regular type rules, as;*

$$list \ \longrightarrow \ [\ ]; \ [any|list]$$

where any is the type containing all terms. □

This particular definition of lists is not parametrised. Such types are commonly referred to as *monomorphic* types. *Polymorphic* types allow variables to occur as parameters to a type definition, where the variables are universally quantified over the defined types.

Example 11. *The parametrised type rule for lists would be the following.*

$$list(X) \ \longrightarrow \ [\ ]; \ [X|list(X)]$$

Suppose a type for integers was also given, int, then the type $list(int)$ would denote a list of integers and $list(list(int))$ would denote a list of lists of integers. □

The monomorphic types are less descriptive than the polymorphic types. In this chapter we focus on the less descriptive monomorphic types. We will also focus on types that are approximations of the minimal model of a program, rather than well-typings.
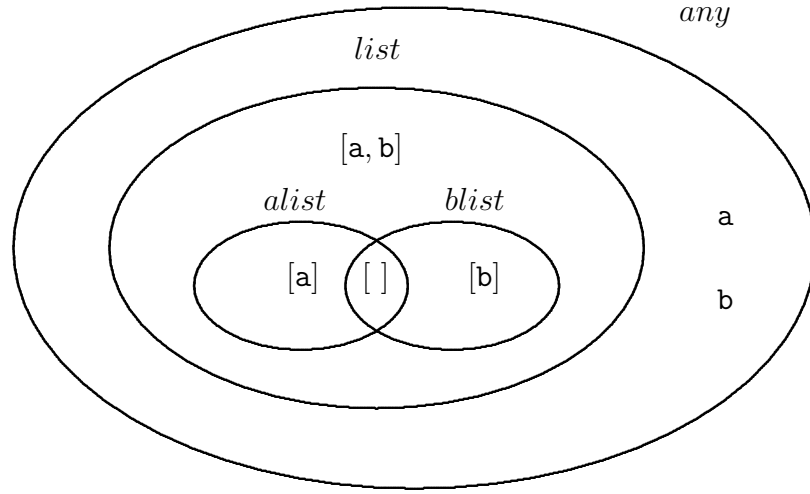
Figure 5.1: Overlapping regular types

Using types for detecting faults in the program, such as predicates that have no solutions, requires precise type definitions. More complex or elaborate types than the ones mentioned so far can be written, but these types will not necessarily lead to more precise analysis results if the types overlap. Example 12 will illustrate this.

Example 12. *An example regular type definition for lists of anything, lists of only 'a's and lists of only 'b's could look like this:*

$$
\begin{aligned}
list &\longrightarrow [\ ];\ [any|list] \\
alist &\longrightarrow [\ ];\ [ta|alist] \\
blist &\longrightarrow [\ ];\ [tb|blist] \\
ta &\longrightarrow a \\
tb &\longrightarrow b \\
any &\longrightarrow a;b;[\ ];[any|any]
\end{aligned}
$$

Precision is lost when elements occur in more than one type. This is perhaps better illustrated in a Venn diagram as shown in Figure 5.1. The element '[a]' occurs in both the set alist and the set list, and '[b]' occurs in both blist and list. This can lead to situations where a list containing only 'a's is not distinguished from the more general list containing anything, including 'a's.

Regular types are in general easy to write down, but this is however not the case with writing disjoint regular types. It would be possible to write a regular

$$list - (alist \cup blist)$$

$$alist - (list \cup blist) \qquad list \cap alist \cap blist \qquad blist - (list \cup alist)$$
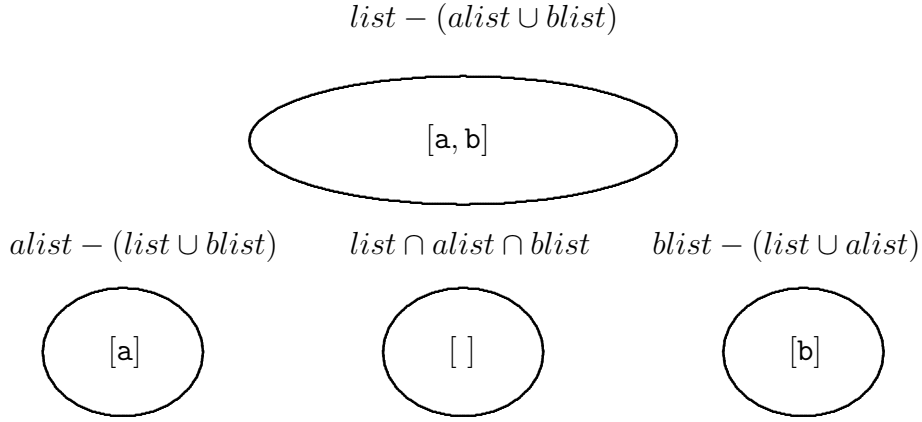
Figure 5.2: Disjoint regular types

type definition for Example 12 that would distinguish between a list of anything and a more specific list of *only* 'a's. These specifications would however not be easy to write down manually and changes and additions to the specifications could require major (and error prone) rewrites.

Deterministic Regular Types

In this Chapter *deterministic regular types*[1] are introduced and a method for transforming overlapping regular types to a set of disjoint regular types is given. The type specification from Example 12 would be transformed into the types shown in Figure 5.2. There are now 4 types with new names: the type $list - (alist \cup blist)$ should read "contained in the original type *list* but at the same time *not* contained in either *alist* or *blist*". The type $list \cap alist \cap blist$ should read "contained in both *list*, *alist* and *blist*". This particular type contains the element '[ ]'.

There is a well established connection between regular types and finite tree automata (FTA). Roughly speaking FTAs are specifications of regular types. The described method in this chapter will use a standard algorithm from tree automaton theory to transform some programmer supplied regular type definitions into a set of disjoint types that can be used for analyses.

---

[1]More precisely, *bottom-up* deterministic regular types

5.2 Tree Automata

Tree automata are "machines" that recognise terms. During the 1980s tree automata appeared as an approximation of programs on which fully automated tools could be used. New results were obtained in areas like properties of programs, type systems and rewriting systems with automata.

Terminology

Let Σ be a set of function symbols. Each function symbol in Σ has a rank (arity) which is a natural number. Whenever we write an expression such as $f(t_1, \ldots, t_n)$, we assume that $f \in \Sigma$ and has arity n. We write f^n to indicate that function symbol f has arity n. If the arity of f is 0 we often write the term $f()$ as f and call f a *constant*. Σ^0 is the set of all constant symbols.

The set of *ground terms* (or *trees*) Term_Σ associated with Σ is the least set containing the constants and all terms $f(t_1, \ldots, t_n)$ such that $t_1, \ldots, t_n$ are elements of Term_Σ and $f \in \Sigma$ has arity n. More formally the set is defined by:

- $\Sigma^0 \subseteq \mathsf{Term}_\Sigma$ and

- if $n \geq 1, f \in \Sigma^n$ and $t_1, ..., t_n \in \mathsf{Term}_\Sigma$, then $f(t_1, ..., t_n) \in \mathsf{Term}_\Sigma$

Example 13. *Let $\Sigma = \{cons(\ ,\ ), nil, a\}$. Here 'cons' is a binary function symbol, 'nil' and 'a' are constants. A example of a term in Term_Σ could be the following: $cons(a, cons(a, nil))$. This can be represented in a graphical way by a tree structure as shown in Figure 5.3.* $\square$
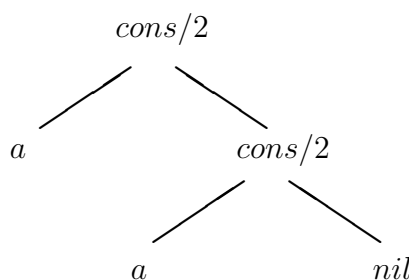
Figure 5.3: The term $cons(a, cons(a, nil))$ represented as a tree.

Finite tree automata provide a means of finitely specifying possibly infinite sets of ground terms, just as finite automata specify sets of strings. A finite tree

automaton (FTA) is defined as a quadruple $\langle Q, Q_f, \Sigma, \Delta \rangle$, where Q is a finite set called *states*, $Q_f \subseteq Q$ is called the set of accepting (or final) states, Σ is a set of ranked function symbols and Δ is a set of *transitions*. Each element of Δ is of the form $f(q_1, \ldots, q_n) \rightarrow q$, where $f \in \Sigma$ and $q, q_1, \ldots, q_n \in Q$.

FTAs can be "run" on terms in Term_Σ; a successful run of a term and an FTA, is one in which the term is *accepted* by the FTA. When a term is accepted, it is accepted by one or more of the final states of the FTA. Different runs may result in different accepting states. At each step of a successful bottom-up run, some sub-term identical to the left hand side of some transition, is replaced by the right hand side, until eventually the whole term is reduced to some accepting state. Implicitly, a tree automaton R defines a set of terms, that is, a tree language, denoted $L(R)$, as the set of all terms that it accepts.

5.2.1 Tree Automata and Types

An accepting state of an FTA can be regarded as a type. Given an automaton $R = \langle Q, Q_f, \Sigma, \Delta \rangle$, and $q \in Q_f$, define the automaton R_q to be $\langle Q, \{q\}, \Sigma, \Delta \rangle$. The language $L(R_q)$ is the set of terms corresponding to type q. We say that a term *is of type q*, written $t : q$, if and only if $q \in L(R_q)$.

A transition $f(q_1, \ldots, q_n) \rightarrow q$, when regarded as a type rule, is usually written the other way around, as $q \rightarrow f(q_1, \ldots, q_n)$. Furthermore, all the rules defining the same type, $q \rightarrow R_1, \ldots, q \rightarrow R_n$ are collected into a single rule of the form $q \rightarrow R_1; \ldots; R_n$. When speaking about types we will usually follow the type notation, but when discussing FTAs we will use the notation for transitions, in order to make it easier to relate to the literature.

Example 14. *Let*

$$
\begin{aligned}
Q &= \{listnat, nat\} \\
Q_f &= \{listnat\} \\
\Delta &= \left\{
\begin{aligned}
[\,] &\rightarrow listnat \\
[nat|listnat] &\rightarrow listnat \\
0 &\rightarrow nat \\
s(nat) &\rightarrow nat
\end{aligned}
\right\}
\end{aligned}
$$

The type listnat is the set of lists of natural numbers in successor notation; the type rule notation is

$$
\begin{aligned}
listnat &\longrightarrow [\,]; [nat|listnat] \\
nat &\longrightarrow 0; s(nat)
\end{aligned}
$$

$\square$

Example 15. *Let*

$$
\begin{aligned}
Q &= \{zero, one, list0, list1\} \\
Q_f &= \{list1\}
\end{aligned}
$$

$$
\Delta = \left\{
\begin{array}{rcl}
[\,] &\rightarrow& list1 \\
[one|list1] &\rightarrow& list1 \\
[zero|list0] &\rightarrow& list1 \\
[\,] &\rightarrow& list0 \\
[zero|list0] &\rightarrow& list0 \\
0 &\rightarrow& zero \\
s(zero) &\rightarrow& one
\end{array}
\right\}
$$

This would be written in type notation as

$$
\begin{aligned}
list1 &\longrightarrow [\,]; [one|list1]; [zero|list0] \\
list0 &\longrightarrow [\,]; [zero|list0] \\
zero &\longrightarrow 0 \\
one &\longrightarrow s(zero)
\end{aligned}
$$

The type $list1$ is the set of lists consisting of zero or more elements $s(0)$ followed by zero or more elements 0 (such as $[s(0), 0]$, $[s(0), s(0), 0, 0, 0]$, $[0, 0]$, $[s(0)], \ldots$). This kind of set is not normally thought of as a type, indicating that the use of FTAs to describe types gives added expressiveness. $\quad\square$

5.2.2 Deterministic and Non-deterministic Tree Automata

There are two notions of non-determinism in tree automata: bottom-up and top-down. Computations can start at the leaves and work upwards (a bottom-up tree) or computations can start at the root and work downwards (top-down trees). They are equivalent with respect to the languages they recognise, but top-down deterministic automata are strictly less powerful than non-deterministic automata. As far as expressiveness is concerned, we can therefore limit our attention to FTAs in which the set of transitions Δ contains no two transitions with the same left-hand-side - these are the *bottom-up deterministic* finite tree automata. For every FTA R there exists a bottom-up deterministic FTA R' such that $L(R) = L(R')$. The sets of terms accepted by states of bottom-up deterministic FTAs are thus disjoint.

5.2.3 Completeness

An automaton $R = \langle Q, Q_f, \Sigma, \Delta \rangle$ is called *complete* if for all n-ary functions $f \in \Sigma$ and states $q_1, \ldots, q_n \in Q$, it contains a transition $f(q_1, \ldots, q_n) \to q$. We may always extend an FTA $\langle Q, Q_f, \Sigma, \Delta \rangle$ to make it complete, by adding a new state $q^\flat$ to Q. Then add transitions of the form $f(q_1, \ldots, q_n) \to q^\flat$ for every combination of f and states $q_1, \ldots, q_n$ (including $q^\flat$) that does not appear in Δ. A complete bottom-up deterministic finite tree automaton, in which every state is an accepting state, partitions the set of terms into disjoint subsets (types), one for each state. In such an automaton $q^\flat$ can be thought of as the error type, that is, the set of terms not accepted by any other type.

Example 16. *Let $\Sigma = \{[\ ], [\_|\_], 0\}$, and let $Q = \{list, listlist, any\}$. We define the set Δ_{any}, for a given Σ, to be the following set of transitions:*

$$\{f(\overbrace{any, \ldots, any}^{n \text{ times}}) \to any \mid f^n \in \Sigma\}$$

Let

$$Q_f \;=\; \{list, listlist\}$$

$$\Delta \;=\; \Delta_{any} \cup \left\{ \begin{array}{rcl} [\ ] & \to & list \\ [any|list] & \to & list \\ [\ ] & \to & listlist \\ [list|listlist] & \to & listlist \end{array} \right\}$$

The type 'list' is the set of lists of any terms, while the type 'listlist' is the set of lists whose elements are of type list (which includes listlist).

The automaton is not bottom-up deterministic; for example, three transitions have the same left-hand-side, namely, $[\ ] \to list, [\ ] \to listlist$ and $[\ ] \to any$. So for example the term $[[0]]$ is accepted by the states $list, listlist$ and any. A determinisation algorithm can be applied, yielding the following:

q_1 corresponding to the type $any \cap list \cap listlist$

q_2 to the type $(list \cap any) - listlist$

q_3 to $any - (list \cup listlist)$

Thus q_1, q_2 and q_3 are disjoint. The automaton is given by

$$
\begin{aligned}
Q &= \{q_1, q_2, q_3\} \\
\Sigma &= \text{as before} \\
Q_f &= \{q_1, q_2\} \\
\Delta &= \left\{ \begin{array}{rcl@{\qquad}rcl}
[\,] & \rightarrow & q_1 & [q_1|q_1] & \rightarrow & q_1 \\
[q_2|q_1] & \rightarrow & q_1 & [q_1|q_2] & \rightarrow & q_2 \\
[q_2|q_2] & \rightarrow & q_2 & [q_3|q_2] & \rightarrow & q_2 \\
[q_3|q_1] & \rightarrow & q_2 & [q_2|q_3] & \rightarrow & q_3 \\
[q_1|q_3] & \rightarrow & q_3 & [q_3|q_3] & \rightarrow & q_3 \\
0 & \rightarrow & q_3 & & &
\end{array} \right\}
\end{aligned}
$$

□

The determinisation algorithm for this example will be discussed in more detail in Section 5.4.

An FTA is *top-down deterministic* if it has no two transitions with both the same right-hand-side and the same function symbol on the left-hand-side. Top-down determinism introduces a loss in expressiveness. It is *not* the case that for each FTA R there is a top-down deterministic FTA R' such that $L(R) = L(R')$. Note that a top-down deterministic automaton can be transformed to an equivalent bottom-up deterministic automaton, as usual, but the result might not be top-down deterministic.

Typical type notations are restricted to the top-down deterministic FTAs. Allowing the use of arbitrary FTAs provides more expressiveness than typical type notations.

Example 17. *Take the automaton from Example 15. This is not top-down deterministic due to the presence of transitions $[one|list1] \rightarrow list1$ and $[zero|list0] \rightarrow list1$. No top-down deterministic automaton can be defined that has the same language. Thus the set accepted by $list1$ could not be defined as a type, using type notations that require top-down deterministic rules (see for instance [121, 154]).*
□

Example 18. *We define the set Δ_{any} as before. Consider the automaton with transitions $\Delta_{any} \cup \{[\,] \rightarrow list, [any|list] \rightarrow list\}$. This is top-down deterministic, but not bottom-up deterministic (since $[\,] \rightarrow list$ and $[\,] \rightarrow any$ both occur). Determinising this automaton would result in one that is not top-down deterministic.*
□

Further details on FTAs and their properties and associated algorithms can be found elsewhere [42].

5.3 Abstract Domains based on FTAs

Logic programs were defined in Chapter 2 along with pre-interpretations and concrete semantics. This section covers abstract interpretation, defined in Chapter 3, for logic programs.

5.3.1 Abstract Interpretation of Logic Programs

Since abstract interpretation can be viewed as a "pseudo-evaluation" of a program, abstract interpretation for logic programming often follows the top-down fashion an actual evaluation of a logic program would follow [24]. Chapter 2 outlined a bottom-up analysis framework based on pre-interpretations. The bottom-up abstract interpretation frameworks for logic programming were first formalised by Marriott and Søndergaard [114].

In the next section we define the connection between the bottom-up pre-interpretation based framework and abstract interpretation.

Bottom-up Abstract Interpretation Framework

Consider a concrete domain $\mathcal{D}$ describing a set of computational states associated with a program P. This could be the set of interpretations over some pre-interpretation such as the Herbrand pre-interpretation. Assume $\mathcal{D}$ is a complete lattice ordered under some partial ordering $\sqsubseteq_{\mathcal{D}}$ and that a transfer function T_P defined on $\mathcal{D}$ exists. The computational behavior of P can be characterised by a solution to the (recursive) equation $I = T_P(I)$. The tuple $\langle \mathcal{D}, \sqsubseteq_{\mathcal{D}}, T_P \rangle$ defines the *standard* fix point semantics for the program P. The computations in this case can be infinite and therefore not useful for static program analysis.

A non-standard abstract fix point semantics for P can be defined over an abstract (not necessarily finite) domain, ordered under some partial ordering and with an abstract transfer function - $\langle \mathcal{D}_\alpha, \sqsubseteq_{\mathcal{D}_\alpha}, T_P^\alpha \rangle$. If each of these elements are designed properly, the abstract fix point semantics would be solvable for P but at the expense of precision; the result would not be exact, but a safe approximation.

Extending this to pre-interpretations, we can define a concretisation function next. The abstract domain $\mathcal{D}_\alpha$ with respect to some pre-interpretation J, is a set of atoms, $p(d_1, \ldots, d_n)$, where p is an n-ary predicate symbol and $d_i \in \mathcal{D}_\alpha$ with $0 \leq i \leq n$. The abstract domain with respect to some pre-interpretation J will be denoted $Atom_J$.

Definition 48 (Concretisation function). *Let P be a program with signature Σ. Let J be any pre-interpretation and let HV be the Herbrand pre-interpretation*

extended over a finite or infinite set of extra constants $\mathcal{V} = \{v_0, v_1, \ldots\}$ not in Σ capturing occurrences of variables. Let $Atom_J$ be the set of domain atoms with respect to pre-interpretation J. The concretisation function $\gamma : 2^{Atom_J} \rightarrow 2^{Atom_{HV}}$ is defined as:

$$\gamma(S) = \left\{\, A \,\middle|\, [A]_J \subseteq [\![S]\!] \,\right\}$$

$\mathsf{M}^J[\![P]\!]$ is an abstraction of the atomic logical consequences of P, in the following sense.

Proposition 1. *Let P be a program with signature Σ, and $\mathcal{V}$ be a set of constants not in Σ (where $\mathcal{V}$ can be either infinite or finite). Let HV be the Herbrand interpretation over $\Sigma \cup \mathcal{V}$ and J be any pre-interpretation of $\Sigma \cup \mathcal{V}$. Then $\mathsf{M}^{HV}[\![P]\!] \subseteq \gamma(\mathsf{M}^J[\![P]\!])$.*

Thus, by defining pre-interpretations and computing the corresponding least model, we obtain safe approximations of the concrete semantics.

5.3.2   Correspondence of FTAs and Pre-Interpretations

A pre-interpretation of a language's underlying signature, Σ, is equivalent to a complete bottom-up deterministic FTA over Σ. Any FTA can be turned into a complete bottom-up deterministic FTA; and how to do this is described in a later section. This principle can be used to construct pre-interpretations from any given FTA.

For a pre-interpretation with a finite domain D over a signature Σ, it defines a complete bottom-up deterministic FTA over the same signature, as follows.

1. The domain D is the set of states of the FTA.

2. Let $\hat{f}$ be the function $D^n \rightarrow D$ assigned to $f \in \Sigma$ by the pre-interpretation. In the corresponding FTA there is a set of transitions $f(d_1, \ldots, d_n) \rightarrow d$, for each $d, d_1, \ldots, d_n$ such that $\hat{f}(d_1, \ldots, d_n) = d$.

5.4   Deriving a Pre-Interpretation from a Regular Type Definition

The analysis framework for logic programming based on pre-interpretations is outlined briefly. The bottom-up declarative semantics (see Section 2.2) describes the model of a program. The previous section described how a bottom-up abstract interpretation framework can be based on (bottom-up) deterministic FTAs leading to a safe approximation of the concrete semantics. The next step is to derive,

from a programmer supplied regular type definition, a pre-interpretation which in turn can be used to compute a model of a given program. The following sections describe how the regular type definition is transformed into a pre-interpretation, first using a textbook algorithm, then using a novel and more efficient algorithm. Then a known method for computing the model of a program with respect to a pre-interpretation, based on *abstract compilation*, is outlined. A more efficient method for computing the model of programs where abstract compilation have been applied is then described. This method is based on Datalog and a BDD-based approach to computing the model of a Datalog program.

5.4.1 Textbook algorithm for determinising Finite Tree Automata

An algorithm for transforming a non-deterministic FTA (NFTA) to a deterministic FTA (DFTA) is presented in [42]. The algorithm is shown here as Algorithm 3, in a modified version that is more suitable for implementation.

Algorithm 3 NFTA to DFTA - rewritten textbook algorithm

---

input: NFTA $R = \langle Q, Q_f, \Sigma, \Delta \rangle$
begin
    Set Q_d to $\emptyset$
    Set Δ'_d to $\emptyset$
    repeat
        Set $\Delta_d = \Delta'_d$
        for each $f^n \in \Sigma$
            for each choice $s_1, \ldots, s_n \in Q_d$
                for each $\langle q_1, \ldots, q_n \rangle \in s_1 \times \ldots \times s_n$
                    $s = \{q \in Q | f(q_1, \ldots, q_n) \to q \in \Delta\}$
                    if $s \neq \emptyset$ **then**
                        Set $\Delta'_d = \Delta'_d \cup \{f(s_1, \ldots, s_n) \to s\}$
                        Set Q_d to $Q_d \cup \{s\}$
                    end if
                end for each
            end for each
        end for each
    until $\Delta'_d = \Delta_d$
    Set Q_{d_f} to $\{s \in Q_d \mid s \cap Q_{d_f} \neq \emptyset\}$
    output: DFTA $R_d = \langle Q_d, Q_{d_f}, \Sigma, \Delta_d \rangle$
end

Description of determinisation algorithm

The algorithm transforms the NFTA from one that operates on states, to one that operates on sets of states from the NFTA. The NFTA allowed multiple occurrences of the same state on the left hand side of a transition. In the DFTA, which is the output of the algorithm, all reachable states in the NFTA are contained in sets that makes up the new states - these are contained in the set Q_d. A state in the NFTA *can* occur in more than one state in the DFTA. Potentially every non-empty subset of set of states of the NFTA can be a state of the DFTA.

The sets in Q_d and the new set of transitions, Δ_d, are generated in an iterative process. In an iteration of the process, a function f is chosen from Σ. Then a number of sets, $s_1, \ldots, s_n$ corresponding to the arity of f, is selected from Q_d - the same set can be chosen more than once. The cartesian product is then formed, $(s_1 \times \cdots \times s_n)$, and for each element in the cartesian product, $q_1, \ldots, q_n$, such that a transition $f(q_1, \ldots, q_n) \to q$ exists, q is added to a set s. When all elements in the cartesian product have been selected, the set s is added to Q_d if s is non-empty and not already in Q_d. A transition $f(s_1, \ldots, s_n) \to s$ is added to Δ_d if s is non-empty.

The algorithm terminates when Q_d is such that no new transitions are added. Initially Q_d is the empty set, so no set containing a state can be chosen from Q_d and therefore only the constants (0-ary functions) can be selected.

Example 19. *In Example 16 a non-deterministic FTA is shown;*

$$
\begin{aligned}
\Sigma &= \{[\ ]^0, [\_|\_]^2, 0^0\} \\
Q &= \{list, listlist, any\} \\
\Delta &= \Delta_{any} \cup \left\{
\begin{array}{rcl}
[\ ] &\to& list \\
[any|list] &\to& list \\
[\ ] &\to& listlist \\
[list|listlist] &\to& listlist
\end{array}
\right\}
\end{aligned}
$$

A step by step application of the algorithm follows:

Step 1: $Q_d = \emptyset, \Delta_d = \emptyset$. *Choose f as a constant, $f = [\ ]$. Now $s = \{q \in Q \mid [\ ] \to q \in \Delta\} = \{any, list, listlist\}$. Add s to Q_d and the transition $[\ ] \to \{any, list, listlist\}$ to Δ_d.*

Step 2: *Choose $f = 0$. Now $s = \{q \in Q \mid 0 \to q \in \Delta\} = \{any\}$. Add s to Q_d and the transition $0 \to \{any\}$ to Δ_d.*

Step 3: *Choose $f = [\ \_ \mid \_\ ]$, $s_1 = s_2 = \{any, list, listlist\}$. Now $s = \{q \in Q \mid \exists q_1 \in s_1, \exists q_2 \in s_2, [q_1 \mid q_2] \to q \in \Delta\} = \{any, list, listlist\}$. Add s to Q_d and the transition $[\{any, list, listlist\} \mid \{any, list, listlist\}] \to \{any, list, listlist\}$ to Δ_d.*

Step 4: *Choose* $f = [\; \_ \mid \_\;]$, $s_1 = s_2 = \{any\}$. *Now* $s = \{q \in Q \mid \exists q_1 \in s_1, \exists q_2 \in s_2, [q_1 \mid q_2] \rightarrow q \in \Delta\} = \{any\}$. *Add* s *to* Q_d *and the transition* $[\{any\} \mid \{any\}] \rightarrow \{any\}$ *to* Δ_d.

Step 5: *Choose* $f = [\; \_ \mid \_\;]$, $s_1 = \{any\}, s_2 = \{any, list, listlist\}$. *Now* $s = \{q \in Q \mid \exists q_1 \in s_1, \exists q_2 \in s_2, [q_1 \mid q_2] \rightarrow q \in \Delta\} = \{any, list\}$. *Add* s *to* Q_d *and the transition* $[\{any\} \mid \{any, list, listlist\}] \rightarrow \{any, list\}$ *to* Δ_d.

Step 6: *Choose* $f = [\; \_ \mid \_\;]$, $s_1 = \{any, list, listlist\}, s_2 = \{any\}$. *Now* $s = \{q \in Q \mid \exists q_1 \in s_1, \exists q_2 \in s_2, [q_1 \mid q_2] \rightarrow q \in \Delta\} = \{any\}$. *Add* s *to* Q_d *and the transition* $[\{any, list, listlist\} \mid \{any\}] \rightarrow \{any\}$ *to* Δ_d.

Step 7 to 11: *No new sets added to* Q_d. *New transitions added:*

$$
\begin{aligned}
[\{any, list\} \mid \{any, list\}] &\rightarrow \{any, list\} \\
[\{any, list\} \mid \{any, list, listlist\}] &\rightarrow \{any, list, listlist\} \\
[\{any, list, listlist\} \mid \{any, list\}] &\rightarrow \{any, list\} \\
[\{any\} \mid \{any, list\}] &\rightarrow \{any, list\} \\
[\{any, list\} \mid \{any\}] &\rightarrow \{any\}
\end{aligned}
$$

Resulting DFTA:

$$
\begin{aligned}
\Sigma &= \{[\;]^0, [\_|\_]^2, 0^0\} \\
Q_d &= \{\{any, list, listlist\}, \{any\}, \{any, list\}\} \\
Q_{d_f} &= \{\{any, list, listlist\}, \{any, list\}\}
\end{aligned}
$$

$$
\Delta_d = \left\{
\begin{aligned}
[\;] &\rightarrow \{any, list, listlist\} \\
0 &\rightarrow \{any\} \\
[\{any, list, listlist\} \mid \{any, list, listlist\}] &\rightarrow \{any, list, listlist\} \\
[\{any\} \mid \{any\}] &\rightarrow \{any\} \\
[\{any\} \mid \{any, list, listlist\}] &\rightarrow \{any, list\}\}, \\
[\{any, list, listlist\} \mid \{any\}] &\rightarrow \{any\} \\
[\{any, list\} \mid \{any, list\}] &\rightarrow \{any, list\} \\
[\{any, list\} \mid \{any, list, listlist\}] &\rightarrow \{any, list, listlist\} \\
[\{any, list, listlist\} \mid \{any, list\}] &\rightarrow \{any, list\} \\
[\{any\} \mid \{any, list\}] &\rightarrow \{any, list\} \\
[\{any, list\} \mid \{any\}] &\rightarrow \{any\}
\end{aligned}
\right\}
$$

The states in Q_d *are equivalent to the states* q_1, q_2, q_3 *in Example 16.* q_1 *is equivalent to the type* $any \cap list \cap listlist$ *represented in* Q_d *as the set* $\{any, list, listlist\}$, q_2 *is equivalent to the type* $(list \cap any) - listlist$ *represented by the set* $\{any, list\}$ *and finally* q_3 *is equivalent to the type* $any - (list \cup listlist)$ *represented by the set* $\{any\}$. $\square$

In a naive implementation of the algorithm, where every combination of arguments to the chosen f would have to be tested in each iteration, the complexity lies

in forming and testing each element in the cartesian product, for every combination of states in Q_d. It is possible to estimate of the number of *operations* required in a single iteration of the process, where an operation is the steps necessary to determine whether $f(q_1, \ldots, q_n) \rightarrow q \in \Delta$. Since Δ is static, an operation can be considered to be of constant time. The number of operations can be estimated by the formula $\#op = (s * e)^a$, where s is the number of states in Q_d, e is the number of elements in a single state in Q_d (possibly an estimate) and a is the arity of the chosen f. Every time a state is added to Q_d, an iteration in the algorithm will require additional operations. Worst case is if the algorithm causes an exponential blow-up in the number of states [42].

The algorithm does not necessarily return a complete DFTA. The procedure described previously in Section 5.2.3 could be applied to complete the DFTA. An alternative solution is to add the standard transitions Δ_{any} to the input NFTA as shown in Example 19. The resulting DFTA will then be complete.

5.5 Efficient determinisation

The determinisation algorithm described in this section generates an automaton whose transitions are represented in product form, which is a more compact form leading to a correspondingly more efficient determinisation algorithm.

5.5.1 Product representation of sets of transitions

The main differences from the textbook algorithm is the form of the output, and in the explicit use of indices for efficient searching of the set of transitions. A *product transition* is of the form $f(Q_1, \ldots, Q_n) \rightarrow q$ where $Q_1, \ldots, Q_n$ are sets of states and q is a state. This product transition denotes the set of transitions $\{f(q_1, \ldots, q_n) \rightarrow q \mid q_1 \in Q_1, \ldots, q_n \in Q_n\}$. Thus $\prod_{i=1\ldots n} |Q_i|$ transitions are represented by a single product transition.

Example 20. *The transitions of the DFTA generated in Example 16 can be represented in product transition form as follows;*

$$\Delta' = \left\{ \begin{array}{rcl} [\,] & \rightarrow & q_1 \\ 0 & \rightarrow & q_3 \\ {[\{q_1, q_2, q_3\}|\{q_3\}]} & \rightarrow & q_3 \\ {[\{q_1, q_2\}|\{q_2\}]} & \rightarrow & q_2 \\ {[\{q_1, q_2\}|\{q_1\}]} & \rightarrow & q_1 \\ {[\{q_1, q_2, q_3\}|\{q_1\}]} & \rightarrow & q_2 \end{array} \right\}$$

Thus 4 product transitions replace the 9 transitions for $[\_|\_]$ *shown in Example 16. There are other equivalent sets of product transitions, for example;*

$$\Delta' = \left\{ \begin{array}{rcl} 0 & \to & q_3 \\ [\{q_1, q_2\}|\{q_3\}] & \to & q_3 \\ [\{q_3\}|\{q_3\}] & \to & q_3 \\ [\{q_1, q_2\}|\{q_2\}] & \to & q_2 \\ [\{q_3\}|\{q_2\}] & \to & q_2 \\ [\{q_1, q_2\}|\{q_1\}] & \to & q_1 \\ [\{q_3\}|\{q_1\}] & \to & q_2 \\ [\,] & \to & q_1 \end{array} \right\}$$

$\square$

5.5.2 A Determinisation Algorithm Generating Product Form

The algorithm developed in this section is based on the classical text-book algorithm as described in Section 5.4. It differs firstly by introducing an index structure to avoid traversing the complete set of transitions in each iteration of the algorithm, and secondly by noting that the algorithm only needs to compute explicitly the set of states of the determinised automaton. The set of transitions can be represented implicitly in the algorithm and generated later if required from the determinised states and the implicit form. However, in our approach the implicit form is close to product transition form and we will use this form directly. Hence, we never need to compute the full set of transitions and this is a major saving of computation. Let $\langle Q, Q_f, \Sigma, \Delta \rangle$ be an FTA. Consider the following functions.

$\mathsf{qmap}_\Delta : (Q \times \Sigma \times \mathcal{N}) \to 2^\Delta$
$\mathsf{qmap}_\Delta(q, f^n, j) = \{f(q_1, \ldots, q_n) \to q_0 \in \Delta \mid q = q_j\}$ for $1 \le j \le n$

$\mathsf{Qmap}_\Delta : (2^Q \times \Sigma \times \mathcal{N}) \to 2^\Delta$
$\mathsf{Qmap}_\Delta(Q', f^n, j) = \bigcup\{\mathsf{qmap}_\Delta(q, f^n, j) \mid q \in Q'\}$

$\mathsf{states}_\Delta : 2^\Delta \to 2^Q$
$\mathsf{states}_\Delta(\Delta') = \{q_0 \mid f(q_1, \ldots, q_n) \to q_0 \in \Delta'\}$

$\mathsf{fmap}_\Delta : \Sigma \times \mathcal{N} \times 2^{2^Q} \to 2^{2^\Delta}$
$\mathsf{fmap}_\Delta(f^n, j, \mathcal{D}) = \{\mathsf{Qmap}_\Delta(Q', f^n, j) \mid Q' \in \mathcal{D}\} \setminus \emptyset$, for $1 \le j \le n$

$\mathcal{C} : 2^Q$
$\mathcal{C} = \{\{q \mid f^0 \to q \in \Delta\} \mid f^0 \in \Sigma\}$

$$\mathsf{F}_\Delta : 2^{2^Q} \to 2^{2^Q}$$

$$\mathsf{F}_\Delta(\mathcal{D}') = \mathcal{C} \cup \{\mathsf{states}_\Delta(\Delta_1 \cap \cdots \cap \Delta_n) \mid \quad f^n \in \Sigma,$$
$$\Delta_1 \in \mathsf{fmap}_\Delta(f^n, 1, \mathcal{D}'),$$
$$\dots,$$
$$\Delta_n \in \mathsf{fmap}_\Delta(f^n, n, \mathcal{D}')\} \setminus \emptyset$$

The subscript Δ is omitted in the context of some fixed FTA. The function qmap_Δ is an index on Δ, recording the set of transitions that contain a given state q at a given position in its left-hand-side. Qmap_Δ is the same index lifted to sets of states.

The complexity of the naive textbook algorithm lies primarily in the inner most loop where all combinations of each argument of a given functor must be searched for in the set of transitions. For functors with a high arity, FTAs with many transitions and states, this gives an explosion in the number tests in each iteration of the algorithm.

The improved algorithm computes in advance for each functor and each argument of the functor the set of transitions where this functor and this particular instance of an argument of this functor is found. This eliminates the need for traversing the set of transitions for each functor and each combination of arguments in each iteration of the algorithm.

The improved algorithm further eliminates the need for examining all possible combinations of arguments for each functor by solving each argument separately. For each argument the set of possible transitions to examine can be found in the set of pre-computed transitions for each functor. The solution is the intersection of the sets of states for each argument.

The algorithm finds the least set $\mathcal{D} \in 2^{2^Q}$ such that $\mathcal{D} = \mathsf{F}(\mathcal{D})$. The set $\mathcal{D}$ is computed using fix point iterations as shown in Algorithm 4.

Algorithm 4 Fix point iterations

---

initialise:
    $i = 0; \quad \mathcal{D}_0 = \emptyset$

repeat
    $\mathcal{D}_{i+1} = \mathsf{F}(\mathcal{D}_i)$
    $i = i + 1$

until $\mathcal{D}_i = \mathcal{D}_{i-1}$

It can be shown that the sequence $\mathcal{D}_0, \mathcal{D}_1, \mathcal{D}_2, \dots$ increases monotonically (with respect to the subset ordering on 2^{2^Q}) and clearly there exists some i such that $\mathcal{D}_{i-1} = \mathcal{D}_i$ since Q is finite.

Example 21. *Consider the following regular types (FTA transitions), in which each transition has been labelled to identify it conveniently. We have $Q = \{any, list\}$ and $\Delta = \{t_1, \ldots, t_5\}$.*

$$t_1 : [\ ] \rightarrow list \qquad\qquad t_4 : [any|any] \rightarrow any$$
$$t_2 : [any|list] \rightarrow list \qquad t_5 : f(any, any) \rightarrow any$$
$$t_3 : [\ ] \rightarrow any$$

The qmap *function is as follows:*

$$\mathsf{qmap}(list, cons, 1) = \emptyset \qquad\qquad \mathsf{qmap}(list, cons, 2) = \{t_2\}$$
$$\mathsf{qmap}(list, f, 1) = \emptyset \qquad\qquad \mathsf{qmap}(list, f, 2) = \emptyset$$
$$\mathsf{qmap}(any, cons, 1) = \{t_2, t_4\} \qquad \mathsf{qmap}(any, cons, 2) = \{t_4\}$$
$$\mathsf{qmap}(any, f, 1) = \{t_5\} \qquad\qquad \mathsf{qmap}(any, f, 2) = \{t_5\}$$

There is only one constant, $[\ ]$, and hence $\mathcal{C} = \{\{any, list\}\}$. Initialise $\mathcal{D}_0 = \emptyset$; the iterations of the algorithm produce the following values.

Step 1: $\mathcal{D}_1 = F(\mathcal{D}_0) = F(\emptyset)$
$\quad\mathsf{fmap}(f, 1, \emptyset) = \mathsf{Qmap}(\emptyset, f, 1) = \mathsf{qmap}(\emptyset, f, 1) = \emptyset$
$\quad\mathsf{fmap}(f, 2, \emptyset) = \mathsf{Qmap}(\emptyset, f, 2) = \mathsf{qmap}(\emptyset, f, 2) = \emptyset$
$\quad\mathsf{fmap}(cons, 1, \emptyset) = \mathsf{Qmap}(\emptyset, cons, 1) = \mathsf{qmap}(\emptyset, cons, 1) = \emptyset$
$\quad\mathsf{fmap}(cons, 2, \emptyset) = \mathsf{Qmap}(\emptyset, cons, 2) = \mathsf{qmap}(\emptyset, cons, 2) = \emptyset$
$\quad F(\emptyset) = \{\{any, list\}\} \cup \emptyset \setminus \emptyset = \{\{any, list\}\}$

Step 2: $\mathcal{D}_2 = F(\mathcal{D}_1) = F(\{\{any, list\}\})$
$\quad\mathsf{fmap}(f, 1, \{\{any, list\}\}) = \mathsf{Qmap}(\{any, list\}, f, 1) =$
$\quad\quad\mathsf{qmap}(list, f, 1) \cup \mathsf{qmap}(any, f, 1) = \{\{t_5\}\}$
$\quad\mathsf{fmap}(f, 2, \{\{any, list\}\}) = \mathsf{Qmap}(\{any, list\}, f, 2) =$
$\quad\quad\mathsf{qmap}(list, f, 2) \cup \mathsf{qmap}(any, f, 2) = \{\{t_5\}\}$
$\quad\mathsf{fmap}(cons, 1, \{\{any, list\}\}) = \mathsf{Qmap}(\{any, list\}, cons, 1) =$
$\quad\quad\mathsf{qmap}(list, cons, 1) \cup \mathsf{qmap}(any, cons, 1) = \{\{t_2, t_4\}\}$
$\quad\mathsf{fmap}(cons, 2, \{\{any, list\}\}) = \mathsf{Qmap}(\{any, list\}, cons, 2) =$
$\quad\quad\mathsf{qmap}(list, cons, 2) \cup \mathsf{qmap}(any, cons, 2) = \{\{t_2, t_4\}\}$
$\quad F(\{\{any, list\}\}) = \{\{any, list\}\} \cup \mathsf{states}(\{t_5\}) \cup \mathsf{states}(\{t_2, t_4\}) =$
$\quad\quad\{\{any, list\}\} \cup \{\{any\}\} \cup \{\{any, list\}\} = \{\{any, list\}, \{any\}\}$

The third step results in no new elements added to $\mathcal{D}_3$, hence at this step the fix point is reached; the short version is:

1. $\mathcal{D}_1 = \{\{any, list\}\}$

2. $\mathcal{D}_2 = \{\{any, list\}, \{any\}\}$

3. $\mathcal{D}_3 = \mathcal{D}_2$

$\square$

The determinised automaton can be constructed from the fix point $\mathcal{D}$ and Qmap. The set of states $\mathcal{Q}$ is $\mathcal{D}$ itself. The set of final states is

$$\mathcal{Q}_f = \{Q' \mid Q' \in \mathcal{Q}, Q' \cap Q_f \neq \emptyset\}$$

The set of transitions is

$$\{f^n(Q_1, \ldots, Q_n) \rightarrow \ \textsf{states}(\textsf{Qmap}(Q_1, f^n, 1) \cap \cdots \cap \textsf{Qmap}(Q_n, f^n, n)) \mid$$
$$f^n \in \Sigma, Q_1 \in \mathcal{Q}, \ldots, Q_n \in \mathcal{Q}\}$$

The transition for each constant f^0 is $f^0 \rightarrow \{q \mid f^0 \rightarrow q \in \Delta\}$. Continuing Example 21, we obtain

$$
\begin{aligned}
[\,] &\rightarrow \{any, list\} \\
[\{any\} \mid \{any\}] &\rightarrow \textsf{states}(\textsf{Qmap}(\{any\}, cons, 1) \cap \\
&\qquad\qquad \textsf{Qmap}(\{any\}, cons, 2)) \\
&\rightarrow \textsf{states}(\{t_2, t_4\} \cap \{t_4\}) \\
&\rightarrow \{any\} \\
[\{any, list\} \mid \{any\}] &\rightarrow \textsf{states}(\textsf{Qmap}(\{any, list\}, cons, 1) \cap \\
&\qquad\qquad \textsf{Qmap}(\{any\}, cons, 2)) \\
&\rightarrow \textsf{states}(\{t_2, t_4\} \cap \{t_4\}) \\
&\rightarrow \{any\} \\
[\{any\}|\{any, list\}] &\rightarrow \textsf{states}(\textsf{Qmap}(\{any\}, cons, 1) \cap \\
&\qquad\qquad \textsf{Qmap}(\{any, list\}, cons, 2)) \\
&\rightarrow \textsf{states}(\{t_2, t_4\} \cap \{t_2, t_4\}) \\
&\rightarrow \{any, list\} \\
[\{any, list\}|\{any, list\}] &\rightarrow \textsf{states}(\textsf{Qmap}(\{any, list\}, cons, 1) \cap \\
&\qquad\qquad \textsf{Qmap}(\{any, list\}, cons, 2)) \\
&\rightarrow \textsf{states}(\{t_2, t_4\} \cap \{t_2, t_4\}) \\
&\rightarrow \{any, list\}
\end{aligned}
$$

(continues on the next page)

88

$$
\begin{aligned}
f(\{any\}, \{any\}) \;\;\rightarrow\;\; & \mathsf{states}(\mathsf{Qmap}(\{any\}, f, 1) \cap \mathsf{Qmap}(\{any\}, f, 2)) \\
\rightarrow\;\; & \mathsf{states}(\{t_5\} \cap \{t_5\}) \\
\rightarrow\;\; & \{any\} \\
f(\{any, list\}, \{any\}) \;\;\rightarrow\;\; & \mathsf{states}(\mathsf{Qmap}(\{any, list\}, f, 1) \cap \mathsf{Qmap}(\{any\}, f, 2)) \\
\rightarrow\;\; & \mathsf{states}(\{t_5\} \cap \{t_5\}) \\
\rightarrow\;\; & \{any\} \\
f(\{any\}, \{any, list\}) \;\;\rightarrow\;\; & \mathsf{states}(\mathsf{Qmap}(\{any\}, f, 1) \cap \mathsf{Qmap}(\{any, list\}, f, 2)) \\
\rightarrow\;\; & \mathsf{states}(\{t_5\} \cap \{t_5\}) \\
\rightarrow\;\; & \{any\} \\
f(\{any, list\}, \{any, list\}) \;\;\rightarrow\;\; & \mathsf{states}(\mathsf{Qmap}(\{any, list\}, f, 1) \cap \\
& \quad\quad \mathsf{Qmap}(\{any, list\}, f, 2)) \\
\rightarrow\;\; & \mathsf{states}(\{t_5\} \cap \{t_5\}) \\
\rightarrow\;\; & \{any\}
\end{aligned}
$$

There are nine transitions in this small example. As we will see we can also obtain a more compact representation as a set of product transitions.

5.5.3   Implementation of the Algorithm.

The function qmap is computed once at the start of the algorithm in time $O(|\Delta|)$, and it can be stored as a hash-table, which allows the computation of $\mathsf{qmap}(q, f, j)$ in constant time. The value of $\mathsf{Qmap}(Q', f, j)$ can thus be computed in $O(|Q|)$. $\mathsf{states}(\Delta')$ can be computed in $O(|\Delta|)$ after construction of a suitable index to the transitions.

The function fmap is maintained as a table, called ftable. As described above, the algorithm computes a sequence $\emptyset, \mathsf{F}(\emptyset), \mathsf{F}^2(\emptyset), \ldots$, where $\mathcal{D}_i = \mathsf{F}^i(\emptyset)$. Let $\mathcal{D}_i$ and $\mathcal{D}_{i+1}$ be successive values of the sequence. At the $i + 1^{th}$ stage of the algorithm values of the form $\mathsf{fmap}(f, j, \mathcal{D}_{i+1})$ are computed for each f and j. We use the property that $\mathsf{fmap}(f, j, \mathcal{D}_{i+1}) = \mathsf{fmap}(f, j, \mathcal{D}_i) \cup \mathsf{fmap}(f, j, (\mathcal{D}_{i+1} \setminus \mathcal{D}_i))$. The table entry $\mathsf{ftable}(f^n, j)$ holds the values of $\mathsf{fmap}(f, j, \mathcal{D}_i)$ on the ith iteration of the algorithm. Hence on the next iteration only the new values of fmap, that is, $\mathsf{fmap}(f, j, (\mathcal{D}_{i+1} \setminus \mathcal{D}_i))$, need to be added to $\mathsf{ftable}(f, j)$.

The evaluation of the function F can also be optimised taking into account the newly computed values of fmap. Assuming the existence of the ftable, define a

function F' as

$$
\begin{aligned}
\mathsf{F}'(\mathcal{D}_{new}) = \{\mathsf{states}(\Delta_1 \cap \cdots \cap \Delta_n) \mid \quad & f^n \in \Sigma, \\
& \Delta_1 \in \mathsf{ftable}(f^n, 1), \\
& \ldots, \\
& \Delta_j \in \mathsf{fmap}(f^n, j, \mathcal{D}_{new}), \\
& \ldots, \\
& \Delta_n \in \mathsf{ftable}(f^n, n), \\
& 1 \le j \le n\} \setminus \emptyset
\end{aligned}
$$

Thus for each tuple $\Delta_1, \ldots, \Delta_n$, at least one component of the tuple must be chosen from $\mathcal{D}_{new}$, ensuring that each tuple $\Delta_1, \ldots, \Delta_n$ needs to be considered only once for each f^n during the execution of the algorithm. After performing these optimisations the algorithm can be summarised as shown in Algorithm 5.

Algorithm 5 Efficient NFTA to DFTA

$\mathcal{D} = \mathcal{C}$; $\mathcal{D}_{new} = \mathcal{D}$;
for $f^n \in \Sigma$
  for $j = 1$ **to** n
    $\mathsf{ftable}(f^n, j) = \emptyset$
  endfor
endfor
repeat
  $\mathcal{D}_{old} = \mathcal{D}$;
  for $f^n \in \Sigma$
    for $j = 1$ **to** n
      $\mathsf{ftable}(f^n, j) = \mathsf{ftable}(f^n, j) \cup \mathsf{fmap}(f^n, j, \mathcal{D}_{new})$
    endfor
  endfor
  $\mathcal{D} = \mathcal{D} \cup \mathsf{F}'(\mathcal{D}_{new})$;
  $\mathcal{D}_{new} = \mathcal{D} \setminus \mathcal{D}_{old}$
until $\mathcal{D}_{new} = \emptyset$

5.5.4 Complexity

For each $f^n \in \Sigma$, the computation time is dominated by the number of tuples $Q_1, \ldots, Q_n$ that have to be considered during the computation of F. This is $\prod_{i=1\ldots n} |\mathsf{fmap}(f, i, \mathcal{D})|$. The maximum size of $|\mathsf{fmap}(f, i, \mathcal{D})|$ is the number of possible right-hand-sides in the determinised transitions for a f, say k_f. This is 2^Q in the worst case, but in practice it is often much smaller. Experiments with the

algorithm were published in [74]. The number of tuples is in fact closely related to the set of product transitions generated as shown next.

Let $f^n \in \Sigma$ and let $\mathcal{D}$ be the set of sets of states computed as the fix point in the algorithm. Then the set of product transitions for f^n ($n > 0$) is

$$\{f(\mathsf{fmap}^{-1}(\Delta_1, f^n, 1), \ldots, \mathsf{fmap}^{-1}(\Delta_n, f^n, n)) \to$$
$$\mathsf{states}(\Delta_1 \cap \cdots \cap \Delta_n) \mid \Delta_1 \in \mathsf{fmap}(f^n, 1, \mathcal{D}), \ldots, \Delta_n \in \mathsf{fmap}(f^n, n, \mathcal{D})\}$$

where

$$\mathsf{fmap}^{-1}(\Delta', f^n, i) = \{Q' \mid \mathsf{Qmap}(Q', f^n, i) = \Delta', Q' \in \mathcal{D}\}$$

$\mathsf{fmap}^{-1}(\Delta', f^n, i)$ can be computed and stored during the evaluation of $\mathsf{fmap}(f^n, i, \mathcal{D})$. For the example above, the final values of the fmap function are

$$\mathsf{fmap}(cons, 1, \mathcal{D}) = \{\{t_2, t_4\}\} \qquad \mathsf{fmap}(cons, 2, \mathcal{D}) = \{\{t_2, t_4\}, \{t_4\}\}$$
$$\mathsf{fmap}(f, 1, \mathcal{D}) = \{\{t_5\}\} \qquad \mathsf{fmap}(f, 2, \mathcal{D}) = \{\{t_5\}\}$$

The values of fmap^{-1} are:

$$\begin{aligned}
\mathsf{fmap}^{-1}(\{t_2, t_4\}, cons, 1) &= \{\{any, list\}, \{list\}\} \\
\mathsf{fmap}^{-1}(\{t_2, t_4\}, cons, 2) &= \{\{any, list\}\} \\
\mathsf{fmap}^{-1}(\{t_4\}, cons, 2) &= \{\{any\}\} \\
\mathsf{fmap}^{-1}(\{t_5\}, f, 1) &= \{\{any, list\}, \{list\}\}
\end{aligned}$$

From these values we obtain the following product transitions (including the transition for the constant $[\,]$).

$$\begin{aligned}
[\{\{any\}, \{any, list\}\} | \{\{any, list\}\}] &\to \{any, list\} \\
[\{\{any\}, \{any, list\}\} | \{\{any\}\}] &\to \{any\} \\
f(\{\{any\}, \{any, list\}\}, \{\{any\}, \{any, list\}\}) &\to \{any\} \\
[\,] &\to \{any, list\}
\end{aligned}$$

The two states $\{any\}$ and $\{any, list\}$ denote non-lists and lists respectively. The determinised automaton is a pre-interpretation over this two-element domain.

5.6 Application of deterministic regular types

In this section we look at examples involving types and modes, mixed with other types. The usefulness of this approach in a Binding Time Analysis (BTA) for off-line partial evaluation will be shown. We also illustrate the applicability of the domains to model-checking.

We assume that Σ includes one special constant v (see Section 2.2.1). The standard type any is assumed where necessary (see Example 16), and it includes the rule $v \to any$.

5.6.1 Definition of Modes as Regular Types

The mode of a predicate in a logic program indicates how its arguments will be instantiated when that predicate is called. The modes of a program represents statements about all computations that are possible from it. For example, in Prolog there is no notion of input/output arguments to a predicate, as there is for instance in the programming language C where an argument of a function can be declared as either *call by value* or *call by reference* [96]. If some arguments would be detected to be consistently input or output for some predicate, these arguments would have a mode.

Instantiation modes can be coded as regular types. The set of ground terms over a given signature, for example, can be described using regular types, as can the set of non-ground terms, the set of variables, and the set of non-variable terms. The definition of the types *ground* (g) and *variable* (var) are

$$
\begin{aligned}
g &= 0; [\,]; [g|g]; s(g) \\
var &= v
\end{aligned}
$$

Using the determinisation algorithm, we can derive other modes automatically. For these examples we assume the signature $\Sigma = \{[\,]^0, [\_|\_]^2, s^1, 0^0\}$ though clearly the definitions can be constructed for any signature. Different pre-interpretations are obtained by taking one or both of the modes g and var along with the type *any*, and then applying the determinisation procedure. The various choices are summarised in Figure 5.4.

The result of computing the least model of the naive reverse program shown in Figure 5.5 is summarised in Figure 5.6. The following abbreviations are used: ground=g, variable=v, non-ground-non-variable=$ngnv$, non-ground=ng, and non-variable=nv. A variable such as X in the abstract model indicates any element of the abstract domain for that model. If the same variable occurs more than once in the same predicate instance it further indicates that these arguments must be the same element.

The analysis based on g and *any* is equivalent to the well-known POS abstract domain [114], while the analysis based on g, var and *any* is the **fgi** domain discussed in [71]. The presence of var in an argument indicates possible freeness, or alternatively, the absence of var indicates definite non-freeness. For example, the answers for *rev* are definitely not free, the first argument of *app* is not free, and if the second argument of *app* is not free then neither is the third. Such dependencies allow accurate propagation of binding time information.

Combining Modes with Other Types

Consider the usual definition of lists, namely $list \longrightarrow [\,]; [any|list]$. Now compute the pre-interpretation derived from the types *list*, *any* and g. Note that the types

Input FTA States	Transitions		
g,var,any	$[\,]$	$\rightarrow$	$\{any, g\}$
	$[\{any\}|\_]$	$\rightarrow$	$\{any\}$
	$[\{any, var\}|\_]$	$\rightarrow$	$\{any\}$
	$[\_|\{any\}]$	$\rightarrow$	$\{any\}$
	$[\_|\{any, var\}]$	$\rightarrow$	$\{any\}$
	$[\{any, g\}|\{any, g\}]$	$\rightarrow$	$\{any, g\}$
	$s(\{any\})$	$\rightarrow$	$\{any\}$
	$s(\{any, var\})$	$\rightarrow$	$\{any\}$
	$s(\{any, g\})$	$\rightarrow$	$\{any, g\}$
	0	$\rightarrow$	$\{any, g\}$
g,any	$[\,]$	$\rightarrow$	$\{any, g\}$
	$[\{any\}|\_]$	$\rightarrow$	$\{any\}$
	$[\_|\{any\}]$	$\rightarrow$	$\{any\}$
	$[\{any, g\}|\{any, g\}]$	$\rightarrow$	$\{any, g\}$
	$s(\{any\})$	$\rightarrow$	$\{any\}$
	$s(\{any, g\})$	$\rightarrow$	$\{any, g\}$
	0	$\rightarrow$	$\{any, g\}$
var,any	$[\,]$	$\rightarrow$	$\{any\}$
	$[\_|\_]$	$\rightarrow$	$\{any\}$
	$s(\_)$	$\rightarrow$	$\{any\}$
	0	$\rightarrow$	$\{any\}$

Input FTA states	Output FTA states	Corresponding modes
g, var, any	{any,g}, {any,var}, {any}	ground, variable, non-ground-non-variable
g, any	{any,g}, {any}	ground, non-ground
var, any	{any,var}, {any}	variable, non-variable

Figure 5.4: Mode pre-interpretations obtained from g, var and any

$rev([\,],[\,]).$

$rev([X|Xs],Zs) \leftarrow$
$\quad rev(Xs,Ys), app(Ys,[X],Zs).$

$app([\,],Ys,Ys).$

$app([X|Xs],Ys,[X|Zs]) \leftarrow$
$\quad app(Xs,Ys,Zs).$

Figure 5.5: Naive Reverse program

93

Input types	Model
g, v, any	$\{rev(g,g), rev(ngnv, ngnv),$
	$app(g, var, ngnv), app(g, X, X), app(ngnv, X, ngnv)\}$
g, any	$\{rev(g,g), rev(ng, ng),$
	$app(g, X, X), app(ng, X, ng)\}$
var, any	$\{rev(nv, nv),$
	$app(nv, X, X), app(nv, X, nv)\}$

Figure 5.6: Abstract Models of Naive Reverse program

Input types	Disjoint types	Model
$list, g, any$	$\{any, ground, list\} = gl,$	$\{rev(gl, gl), rev(ngl, ngl),$
	$\{any, list\} = ngl,$	$app(gl, X, X), app(ngl, ngnl, ngnl),$
	$\{any, ground\} = gnl,$	$app(ngl, gl, ngl), app(ngl, ngl, ngl),$
	$\{any\} = ngnl$	$app(ngl, gnl, ngnl)\}$

Figure 5.7: Abstract Models of Naive Reverse program with types *list*, *g* and *any*

list, *any* and *g* intersect. Figure 5.7 shows the set of disjoint types (corresponding to ground lists, non-ground lists, non-ground-non-lists and ground non-lists) and the corresponding abstract model.

5.6.2 Infinite-State Model Checking

The following example [135, 32] is a simple model of a token ring transition system, shown in Figure 5.8. A state of the system is a list of processes indicated by 0 and 1 where a 0 indicates a waiting process and a 1 indicates an active process. The initial state is defined by the predicate $gen/1$ and the predicate $reachable/1$ defines the reachable states with respect to the transition predicate $trans/2$. The required property is that exactly one process is active in any state. The state space is infinite, since the number of processes (the length of the list) is unbounded. Hence finite model checking techniques do not suffice. The example was used in [32] to illustrate set constraint techniques for infinite-state model checking.

We define simple regular types defining the states. The set of "good" states in which there is exactly one '1' is the type *goodlist*. The type *zerolist* is the set of list of zeros. (Note that the same information was provided in [32], but added as clauses to the program. It was also necessary to give an explicit definition of a

94

$$gen([0,1]).$$
$$gen([0|X]) \leftarrow gen(X).$$
$$reachable(X) \leftarrow$$
$$\quad gen(X).$$
$$reachable(X) \leftarrow$$
$$\quad reachable(Y),$$
$$\quad trans(Y,X).$$

$$trans1([0,1|T],[1,0|T]).$$
$$trans1([H|T],[H|T1]) \leftarrow$$
$$\quad trans1(T,T1).$$
$$trans2([0],[1]).$$
$$trans2([H|T],[H|T1]) \leftarrow$$
$$\quad trans2(T,T1).$$

$$trans(X,Y) \leftarrow$$
$$\quad trans1(X,Y).$$
$$trans([1|T],[0|T1]) \leftarrow$$
$$\quad trans2(T,T1).$$

Figure 5.8: Token ring

"bad" state, which is not needed here.)

$$
\begin{aligned}
zero &\longrightarrow 0 \\
one &\longrightarrow 1 \\
goodlist &\longrightarrow [zero|goodlist];\ [one|zerolist] \\
zerolist &\longrightarrow [\ ];\ [zero|zerolist]
\end{aligned}
$$

Determinisation of the given types along with *any* results in five states representing disjoint types: $\{any, one\}$, $\{any, zero\}$, the good lists $\{any, goodlist\}$, the lists of zeros $\{any, zerolist\}$ and all other terms $\{any\}$. We abbreviate these as *one*, *zero*, *goodlist*, *zerolist* and *other* respectively. The least model of the above program over this domain is as follows;

$$
\begin{aligned}
&gen(goodlist), \\
&trans1(goodlist, goodlist), \\
&trans1(other, other), \\
&trans2(other, other), \\
&trans2(goodlist, other), \\
&trans2(goodlist, goodlist), \\
&trans(goodlist, goodlist), \\
&trans(other, other), \\
&reachable(goodlist)
\end{aligned}
$$

The key property of the model is the presence of *reachable(goodlist)* (and the absence of other atoms for *reachable*), indicating that if a state is reachable then it is a *goodlist*. Note that the transitions will handle *other* states, but in the context in which they are invoked, only *goodlist* states are propagated.

In contrast to the use of set constraints or regular type inference to solve this problem, no goal-directed analysis is necessary. Thus there is no need to define an "unsafe" state and show that it is unreachable.

5.6.3 Program Specialisation

In offline partial evaluation, as also described later in Chapter 6, there are two stages. A Binding Time Analysis (BTA) stage where data is marked *static* or *dynamic* indicating whether data is available at specialisation-time or not until run-time, and a specialisation stage where code in the program to be specialised, is either residualised or evaluated depending on the result of the Binding Time Analysis.

Binding Time Analysis for Prolog

Until recently no fully automated binding time analysis existed for Prolog [58]. This meant programmers would have to manually annotate programs, deciding which program construct would be safe to evaluate and which would have to be residualised. This is a process requiring considerable expertise to perform correctly while still producing good specialisation results. To make offline partial evaluation for logic programming useful in general, an automated BTA is required. A binding time analysis takes a program and a description of the input that will be available at specialisation time, and generates the annotations that will guide the partial evaluator during the specialisation phase. The BTA will assign *binding types* and *clause annotation* to the supplied program. A short description of these follows.

Binding Types and Annotations

Arguments of the predicates in the program are assigned a *binding type* by means of a *filter declaration.* A binding type describes the structure of the argument in terms of its static and dynamic components. The basic binding types are

- **static:** the argument is definitely known at specialisation time

- **dynamic:** the argument might be unknown at specialisation time

In addition to the filter declarations, a set of *clause annotations* must also be supplied to the partial evaluator. The clause annotations determines how calls in the clause body should be treated during specialisation. The basic clause annotations are

- **memo:** calls that should not be unfolded, but generalised calls, that are based on the supplied filter declaration, should be generated in the residual program

- **unfold:** calls that should be unfolded at specialisation time, under the control of the partial evaluator

- **call:** fully evaluate the call

- **rescall:** the call is left unmodified in the residual program

The algorithm for performing offline partial evaluation based on these annotations is described in [107].

Constructing a fully automatic binding time analysis poses a few challenges however. At first, the distinguishing of *only* static and dynamic binding types was introduced with functional languages. For logic programming the separation between fully known data and completely unknown data does not suffice. Data that is only partially instantiated occurs frequently at run-time in logic programs. Not using this information would prevent specialisation of many programs. This can be illustrated using list structures. A particular list could be $[A, B, C]$, and though the content of the elements in this list may be dynamic at specialisation time, the structure may be static - in this case the list has a fixed length of 3. Simply annotating a list-argument with the type *dynamic* would prevent the specialiser from unfolding static list structures at specialisation time.

One issue that must be resolved before an automated BTA can be constructed, is the creation of a set of "richer" binding types than just static and dynamic.

Regular Binding Types

Three instatiation modes can be defined for variables in a logic program; *static* (a ground term), *nonvar* (non-variables) and *dynamic* (any term). These can be described as regular types for a given signature.

Example 22. *For* $\Sigma = \{[\ ]^0, [\ \_ \ | \ \_ \ ]^2, s^1, 0^0, v^0\}$ *the regular type rules would be*

$$
\begin{aligned}
static &\longrightarrow 0;\ [\ ];\ [static|static];\ s(static) \\
nonvar &\longrightarrow 0;\ [\ ];\ [dynamic|dynamic];\ s(dynamic) \\
dynamic &\longrightarrow 0;\ [\ ];\ [dynamic|dynamic];\ s(dynamic);\ v
\end{aligned}
$$

Description of data structures can be added to these regular type definitions. For instance the rules for lists would be defined as

$$
list \longrightarrow [\ ];\ [dynamic|list]
$$

Figure 5.9 shows how these types overlap. □

These type rules overlap and terms like $[0, 0]$ would be recognised by all three modes and by the definition of lists. Determinising the types in Example 22 would
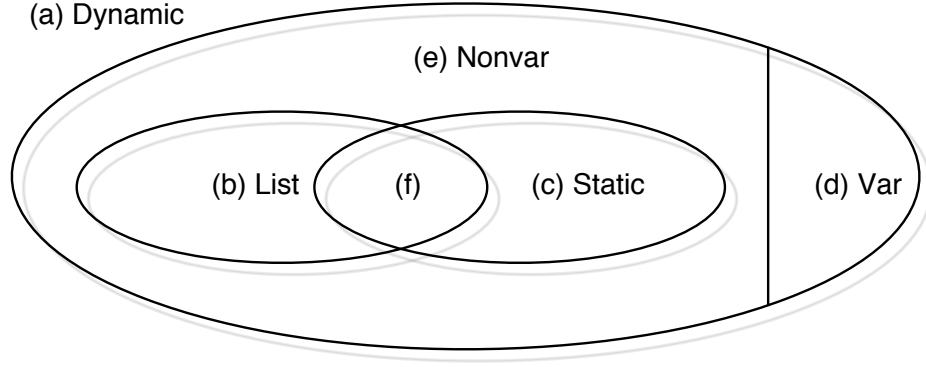
Figure 5.9: Example of binding types for offline partial evaluation

yield the result

$$
\begin{array}{lcl}
0 & \longrightarrow & \{dynamic, nonvar, static\} \\
[\{dynamic\}|\{dynamic\}] & \longrightarrow & \{dynamic, nonvar\} \\
[\{dynamic\}|\{dynamic, list, nonvar\}] & \longrightarrow & \{dynamic, list, nonvar\} \\
[\{dynamic\}|\{dynamic, list, nonvar, static\}] & \longrightarrow & \{dynamic, list, nonvar\} \\
[\{dynamic\}|\{dynamic, nonvar\}] & \longrightarrow & \{dynamic, nonvar\} \\
[\{dynamic\}|\{dynamic, nonvar, static\}] & \longrightarrow & \{dynamic, nonvar\} \\
[\{dynamic, list, nonvar\}|\{dynamic\}] & \longrightarrow & \{dynamic, nonvar\} \\
[\{dynamic, list, nonvar\}|\{dynamic, list, nonvar\}] & \longrightarrow & \{dynamic, list, nonvar\} \\
[\{dynamic, list, nonvar\}|\{dynamic, list, nonvar, static\}] & \longrightarrow & \{dynamic, list, nonvar\} \\
[\{dynamic, list, nonvar\}|\{dynamic, nonvar\}] & \longrightarrow & \{dynamic, nonvar\} \\
[\{dynamic, list, nonvar\}|\{dynamic, nonvar, static\}] & \longrightarrow & \{dynamic, nonvar\} \\
[\{dynamic, list, nonvar, static\}|\{dynamic\}] & \longrightarrow & \{dynamic, nonvar\} \\
[\{dynamic, list, nonvar, static\}|\{dynamic, list, nonvar\}] & \longrightarrow & \{dynamic, list, nonvar\} \\
[\{dynamic, list, nonvar, static\}|\{dynamic, list, nonvar, static\}] & \longrightarrow & \{dynamic, list, nonvar, static\} \\
[\{dynamic, list, nonvar, static\}|\{dynamic, nonvar\}] & \longrightarrow & \{dynamic, nonvar\} \\
[\{dynamic, list, nonvar, static\}|\{dynamic, nonvar, static\}] & \longrightarrow & \{dynamic, nonvar, static\} \\
[\{dynamic, nonvar\}|\{dynamic\}] & \longrightarrow & \{dynamic, nonvar\} \\
[\{dynamic, nonvar\}|\{dynamic, list, nonvar\}] & \longrightarrow & \{dynamic, list, nonvar\} \\
[\{dynamic, nonvar\}|\{dynamic, list, nonvar, static\}] & \longrightarrow & \{dynamic, list, nonvar\} \\
[\{dynamic, nonvar\}|\{dynamic, nonvar\}] & \longrightarrow & \{dynamic, nonvar\} \\
[\{dynamic, nonvar\}|\{dynamic, nonvar, static\}] & \longrightarrow & \{dynamic, nonvar\} \\
[\{dynamic, nonvar, static\}|\{dynamic\}] & \longrightarrow & \{dynamic, nonvar\} \\
[\{dynamic, nonvar, static\}|\{dynamic, list, nonvar\}] & \longrightarrow & \{dynamic, list, nonvar\} \\
[\{dynamic, nonvar, static\}|\{dynamic, list, nonvar, static\}] & \longrightarrow & \{dynamic, list, nonvar, static\} \\
[\{dynamic, nonvar, static\}|\{dynamic, nonvar\}] & \longrightarrow & \{dynamic, nonvar\} \\
[\{dynamic, nonvar, static\}|\{dynamic, nonvar, static\}] & \longrightarrow & \{dynamic, nonvar, static\} \\
[\,] & \longrightarrow & \{dynamic, list, nonvar, static\} \\
s(\{dynamic, list, nonvar\}) & \longrightarrow & \{dynamic, nonvar\} \\
s(\{dynamic, list, nonvar, static\}) & \longrightarrow & \{dynamic, nonvar, static\} \\
s(\{dynamic, nonvar\}) & \longrightarrow & \{dynamic, nonvar\} \\
s(\{dynamic, nonvar, static\}) & \longrightarrow & \{dynamic, nonvar, static\} \\
v & \longrightarrow & \{dynamic\}
\end{array}
$$

The process of obtaining the set of deterministic types is fully automatic. They are derived from the programmer specified types. The deterministic types can then be used to propagate more precise binding types, describing which types of terms can arise at which argument position for a given predicate. The term

$[0, 0]$ that for the regular type description, could be interpreted as either static, dynamic, list or nonvar, can for the deterministic type description only be considered $\{dynamic, list, nonvar, static\}$; the informal interpretation would be that the term $[0, 0]$ is definitely a *list* and definitely *static* (in addition to being definitely *dynamic* and *nonvar*) and definitely not *var*. Another interpretation of the deterministic types for this particular example could be

- $\{dynamic, nonvar\}$ is equivalent to the type of a term that is non-ground, non-variable and non-list. In Figure 5.9 these elements would be in the subset denoted (e).

- $\{dynamic, list, nonvar\}$ is equivalent to the type of terms that are non-ground lists. These would be in the subset denoted (b).

- $\{dynamic, nonvar, static\}$ is equivalent to the type of terms that are ground non-lists. This would be the subset denoted (c).

- $\{dynamic, list, nonvar, static\}$ is equivalent to the type of terms that are ground lists. This would be the subset denoted (f).

In addition to this new "richer" set of binding types, two other issues must be resolved before an automated BTA can be constructed; the binding types must be propagated through the program to be specialised and some method must be established to ensure termination of the later specialisation step. The complete procedure is described in [58]. For the termination problem, this is handled using a convex hull abstraction of the argument size relationships, similar to the process described in Chapter 4.

The automated BTA has been integrated in the Logen [107] specialiser for logic programs. The specialiser is available at the URL

    http://www.stups.uni-duesseldorf.de/~pe/weblogen/

5.7   Implementation issues

The analyser based on deterministic regular types has two main components; the first component is the determinisation algorithm that takes the supplied type definition and produces a pre-interpretation and the second component calculates the least model of the supplied program for analysis and computes the least model with respect to the pre-interpretation derived by the first component.

Both an implementation of the textbook algorithm for determinising an NFTA (Algorithm 3 on page 81) and an implementation of the efficient determinisation

algorithm (Algorithm 5 on page 90) has been implemented. Both implementations follow the outlined algorithms.

The calculation of the least model can be computed based on, for example, John Gallagher's Bottom-Up Toolkit [69] and a logic program transformation technique called *abstract compilation* (described in the next section). Additionally a BDD and Datalog based approach to computing least models with respect to a pre-interpretation is described.

5.7.1 Abstract Compilation of a Pre-Interpretation

The idea of abstract compilation was introduced first by Debray and Warren [62]. Operations on the abstract domain are coded as logic programs and added (or compiled) directly to the target program, which is then executed according to standard concrete semantics. The reason for this technique is to avoid some of the overhead of interpreting the abstract operations.

Abstract compilation introduces a new binary predicate '→' representing the pre-interpretation directly into the program P to be analysed. P is transformed through an iterative process. In each iteration, each clause of the program is transformed by replacing non-variable terms occurring in the clause, of the form $f(X_1, \ldots, X_m)$ where $X_1, \ldots, X_m$ ($m \geq 0$) are variables, by a fresh variable U and adding the atom $f(X_1, \ldots, X_m) \rightarrow U$ to the clause body. We stop iterating when the only non-variables in the clause occur in the first argument of $\rightarrow$. The transformed program is denoted $\bar{P}$.

When a specific pre-interpretation J is added to $\bar{P}$, the result is a *domain program* for J, called $\bar{P}^J$. Clearly $\bar{P}^J$ have a different language than P, since the definition of '→' contains elements of the domain of the interpretation. It can easily be shown that the minimal Herbrand model of $\bar{P}^J$ (restricted to the original program predicates) is isomorphic to $\mathsf{M}^J[\![P]\!]$. An example of the domain program for *append* and the pre-interpretation for variable/non-variable is shown in Figure 5.10.

Computing the least model

The least model $\mathsf{M}^J[\![P]\!] = \mathsf{lfp}(T_P^J)$ is obtained by computing $\mathsf{lfp}(T_{\bar{P}j})$, and then restricting to the original predicates in P discarding the '→' predicates introduced by the abstract compilation.

5.7.2 Computing Models of Datalog Programs

Performing an analysis based on pre-interpretations is equivalent to computing the minimal Herbrand model of a (definite) Datalog program [147]. A definite Datalog

The append program:

$$append([\,], Xs, Xs) \leftarrow$$
$$append([X|Xs], Ys, [X|Zs]) \leftarrow append(Xs, Ys, Zs)$$

The append program transformed using abstract compilation:

$$
\begin{array}{rcl}
append(U, Xs, Xs) & \leftarrow & [\,] \rightarrow U \\
append(U, Ys, V) & \leftarrow & append(Xs, Ys, Zs), \\
& & [X|Xs] \rightarrow U, \\
& & [X|Zs] \rightarrow V
\end{array}
$$

A *var/nonvar* domain:

$$
D = \left\{
\begin{array}{rcl}
v & \rightarrow & var \\
[\,] & \rightarrow & nonvar \\
[nonvar|nonvar] & \rightarrow & nonvar \\
[var|nonvar] & \rightarrow & nonvar \\
[nonvar|var] & \rightarrow & nonvar \\
[var|var] & \rightarrow & nonvar
\end{array}
\right\}
$$

The model of the append program over the *var/nonvar* domain:

$$\{append(nonvar, nonvar, nonvar), append(nonvar, var, var),$$
$$append(nonvar, var, nonvar)\}$$

Figure 5.10: Example of a domain program for the *append* program, and the domain *var/nonvar*.

program is a set of Horn clauses containing no function symbols with arity greater than zero. The Herbrand models of such programs are finite.

In the abstract domain programs defined in Section 2.2.1, a pre-interpretation was represented by a set of facts of the form

$$(f(d_1, \ldots, d_n) \to d) \leftarrow true$$

Although there are function symbols occuring in such facts, we can easily represent the facts using a separate predicate for each function symbol; say pre_f is the relation corresponding to f. Then all atoms of form $f(d_1, \ldots, d_n) \to d$ would be represented as the function-free atom $pre_f(d_1, \ldots, d_n, d)$ instead. Since function symbols occur nowhere else in the abstract domain program, we are left with a Datalog program.

Efficient techniques for computing Datalog models have been studied extensively in research on deductive database systems [147], and indeed, many techniques (especially algorithms for computing joins) from the field of relational databases are also relevant. In the logic programming context, facts containing variables are also allowed; tabulation and subsumption techniques have been applied in a Datalog model evaluation system for program analysis [59].

The analysis method based on pre-interpretations is of course independent of which technique is used for computing the model of the Datalog program. Having transformed the analysis task to that of computing a Datalog program model, we are free to choose the best method available. Recent methods allow very large Datalog programs to be solved [152].

Computing Datalog models using BDDs

Here Binary Decision Diagrams (BDDs) and a tool for solving Datalog programs, based on BDDs are introduced.

Boolean functions

Definition 49 (Boolean Domain). *A Boolean domain B is a 2-element set, whose elements are interpreted as logical values, typically $B = \{0, 1\}$ or $B = \{false, true\}$. A Boolean variable ranges over a Boolean Domain.*

Definition 50 (Boolean Function). *A Boolean function is a function of the form $f : B \times \ldots \times B \mapsto B$, where B is a Boolean Domain.*

A Boolean function of arity n is typically written as a propositional formula over a set of variables $x_1, \ldots, x_n$.

Example 23. *Take the Boolean function* logical and *defined as* $\{\langle 0, 0 \rangle \mapsto 0, \langle 0, 1 \rangle \mapsto 0, \langle 1, 0 \rangle \mapsto 0, \langle 1, 1 \rangle \mapsto 1\}$. *This function will be written as* $x_1 \wedge x_2$. □

Definition 51 (Boolean Valued Function). *A function* $f : D_1 \times \ldots \times D_n \mapsto B$ *where* D_i, $i \in [1, \ldots, n]$ *is an arbitrary domain, and B is a Boolean domain, is called a* Boolean valued *function.*

Binary Decision Diagrams

Definition 52 (Binary Decision Diagram). *A Binary Decision Diagram (BDD) is a rooted Directed Acyclic Graph (DAG) that satisfies the following properties*

1. *the internal nodes of the graph are labeled by variables*

2. *the leaves are labelled by the Boolean constants* 0 *and* 1

3. *every internal node has exactly two children with edges to these labelled* 0 *and* 1

The BDD is called *ordered* if different variables occurs in the same order on all paths from the root. It is called a *Reduced Ordered* BDD (ROBDD) if any isomorphic subgraphs are merged and any node with isomorphic children are eliminated. Ordered Binary Decision Diagrams are a canonical and efficient way of representing and manipulating Boolean functions [28]. The ordering used on the enumerating variables determines how efficient the representation is. Some Boolean functions have BDDs of exponential size.

The domains that represent pre-interpretations are usually not Boolean domains. Boolean valued functions over these domains would not in general be Boolean functions. Only the Boolean functions have a direct representation as a BDD. However a Boolean valued function over a finite domain can be represented by a Boolean function that in turn has a BDD representation. This method is described next.

Let R^n be an n-ary relation over a finite domain D, with D containing m elements. Then suppose R^n must be represented as a Boolean function. The m elements in D can be encoded using $k = \lceil log_2(m) \rceil$ bits. So $n.k$ Boolean variables are introduced;

$$x_{1,1}, \ldots, x_{1,k}, x_{2,1}, \ldots, x_{n,1}, \ldots, x_{n,k}$$

A tuple $(d_1, \ldots, d_n) \in R^n$ is then a conjunction

$$x_{1,1} = b_{1,1} \wedge \ldots \wedge x_{n,k} = b_{n,k} \tag{5.1}$$

where $b_{i,1}, \ldots, b_{i,k}$ is the encoding of d_i for $1 \leq i \leq n$.

A relation $R : D_1 \times \ldots \times D_n$ over a finite domain D, can be represented as a Boolean valued function, $f : D_1 \times \ldots \times D_n \mapsto \{0, 1\}$, such that $(d_1, \ldots, d_n) \in R \Leftrightarrow f(d_1, \ldots, d_n) = 1$ and $(d_1, \ldots, d_n) \notin R \Leftrightarrow f(d_1, \ldots, d_n) = 0$. A finite relation can therefore also be represented as a disjunction of conjunctions of the form shown in Equation 5.1.

BDDs allow very large relations, translated in this way into Boolean formulas, to be represented compactly (though variable ordering is critical, and there are some relations that admit no compact representation).

The possibility of using Boolean functions to represent finite relations was exploited in model-checking [34].

BDD-Based Deductive DataBase - bddbddb

John Whaley has developed a BDD based solver for Datalog programs, called `bddbddb` [153]. This tool computes the model of a Datalog program, and provides facilities for querying Datalog programs. It is written in Java and can link to established BDD libraries using the Java Native Interface (JNI).

Any programs written in Datalog in this thesis or any experiment performed on Datalog program, is solved using `bddbddb` linked to the BuDDy package [110]. We wrote a front end to translate our abstract logic programs and pre-interpretations into the form required by `bddbddb`.

In a BDD-based evaluation of a Datalog program, the solution of each predicate is thus represented as a Boolean formula (in BDD form) and the relational operations required to compute the model can be translated into operations on BDDs. For example, if we are solving the conjunction

$$p(A, B), q(B, C)$$

we take the Boolean formulas representing the current solutions of p and q, say F_p and F_q and build a new BDD representing the formula

$$F_p \wedge F_q \wedge x_{2,1} = y_{1,1} \wedge \ldots \wedge x_{2,k} = y_{1,k}$$

where
$$x_{1,1}, \ldots, x_{1,k}, x_{2,1}, \ldots, x_{2,k} \text{ and}$$
$$y_{1,1}, \ldots, y_{1,k}, y_{2,1}, \ldots, y_{2,k}$$

are the Boolean variables representing the respective arguments of p and q.

Representing and manipulating Boolean formulas is a very active research field and there are other techniques besides BDDs that are competitive. In logic-program analyses, multi-headed clauses have demonstrated good performance when compared to BDDs, for example [90].

5.7.3 From Product Representations to Datalog

The determinisation algorithm in Section 5.5 returns transitions in product form. Though this saves computation, we still need to represent the product form as a Datalog program, so that we can exploit techniques such as BDD-based evaluation of the model (see Section 5.7.2).

Consider a product transition $f(\{a,b\},\{c,d,e\}) \rightarrow q$. As before, we can introduce a predicate for each function to replace the arrow relation, obtaining $pre_f(\{a,b\},\{c,d,e\},q)$. To represent this as a clause we could write the following.

$$pre_f(X,Y,q) \leftarrow member(X,[a,b]), member(Y,[c,d,e]).$$

To convert to Datalog we need only introduce a specialised *member* predicate for each set that occur as an argument in a product transition. In the above case we obtain:

$$
\begin{array}{ll}
pre_f(X,Y,q) \leftarrow m_1(X), m_2(Y). & m_2(c) \leftarrow true. \\
m_1(a) \leftarrow true. & m_2(d) \leftarrow true. \\
m_1(b) \leftarrow true. & m_2(e) \leftarrow true.
\end{array}
$$

As a further optimisation, if some product transition has for some argument a set containing all of the determinised states, we may simply replace that argument by an anonymous variable (a "don't care" argument). Also, singleton sets $\{q\}$ can be replaced by q instead of introducing a deterministic *member* call. For the transitions produced from Example 21, the set of determinised states was $\{\{any\}, \{any, list\}\}$. (We can write these states as constants q_1, q_2 respectively). The product transitions are

$$
\begin{array}{rcl}
[\{q_1,q_2\}|\{q_2\}] & \rightarrow & q_2 \\
[\{q_1,q_2\}|\{q_1\}] & \rightarrow & q_1 \\
f(\{q_1,q_2\},\{q_1,q_2\}) & \rightarrow & q_1 \\
[\,] & \rightarrow & q_2
\end{array}
$$

The Datalog program is thus

$$
\begin{array}{l}
pre_{cons}(\_,q_2,q_2) \leftarrow true. \\
pre_{cons}(\_,q_1,q_1) \leftarrow true. \\
pre_f(\_,\_,q_1) \leftarrow true. \\
pre_{nil}(q_2) \leftarrow true.
\end{array}
$$

5.8 Experiments

We now summarise the analysis procedure. The procedure takes two inputs: a program P to be analysed and a set of regular type definitions R expressing term properties of interest. The procedure then follows these steps.

1. Augment the types with a standard type *any* over the signature of the program, and determinise yielding transitions R_d in product form.

2. Transform P to an abstract domain program P_a (using flattened predicates pre_f to denote the pre-interpretation of function f as explained in Section 5.7.3).

3. Transform R_d to a suitable Datalog representation R_{dat}, again using the pre_f representation, together with the specialised *member* predicates for the product transitions (and optionally introduce don't care arguments).

4. Transform $P_a \cup R_{dat}$ to the syntax required by `bddbddb` and compute its least model.

All experiments report in this section has been carried out on a machine equipped with an Intel Xeon E5355 2.66 GHz quad core processor, 8 GB of memory and a 64-bit version of Debian Linux install. The determinisation algorithm is implemented in Ciao Prolog and the `bddbddb` tool is implemented in Java using the supplied JFactory BDD library.

5.8.1 Experiments on determinisation

Figure 5.11 shows a few experimental results just illustrating the effect of the determinisation algorithm. For each input FTA, the table shows the number of states Q and transitions Δ, followed by the number of states in the output DFTA, Q_d. Three measures of the set of transitions are shown. First the total number of transitions Δ_d, followed by the size of the set of product transitions generated by the algorithm $\Delta_{\prod}$. Thirdly we show the size of another set of product transitions Δ_{dc} that is generated by locating "don't care" arguments. The final column is the time in seconds to compute the product form Δ_{dc} (which is almost identical to the time to compute $\Delta_{\prod}$).

The most important observation is the significant reduction in size of $\Delta_{\prod}$ and Δ_{dc} compared to Δ_d. Note also that the set of states in the DFTA can actually be less than the set of states in the input FTA, as in the `dnf` example. This is because, as is typical in automatically generated FTAs, there are many equivalent states in the input, and this redundancy is removed in the DFTA.

The input FTAs are `chr`, a set of regular types for analysing a CHR transition system; `dnf`, the regular type inferred automatically by the abstract interpretation over DFTAs described in [75]; `mat1`, a set of types for an offline binding time analysis of a matrix transposition program; `mat2`, the regular types from Example 16 augmented by two extra function symbols; `ring`, the regular types describing states in the token-ring analysis problem (Section 5.6.2); `pic`, a set of regular types

	FTA		DFTA				
Name	Q	Δ	Q_d	Δ_d	$\Delta_{\prod}$	Δ_{dc}	msecs
chr	21	84	46	216192	262	252	20.9
dnf	104	791	57	6567	168	141	59.7
mat1	6	10	6	39	8	8	0.2
mat2	3	8	3	12	9	7	0.1
ring	5	12	5	30	14	11	0.1
pic	8	270	8	4989	274	268	1.7
aquarius	4	1866	5	9993536	4131	1760	67.7
chatparser	4	656	5	86803	695	433	6.7

Figure 5.11: Determinisation results for improved algorithm. Timing results are reported in milliseconds and they are obtained using the built-in timing features of Ciao Prolog.

expressing properties of a PIC processor emulator (described later in Chapter 6); `aquarius`, the Aquarius compiler described in [151] and `chatparser`, part of a set of benchmark programs for the Aquarius compiler.

Comparison with the Timbuk tool

Timbuk [77] is a publicly available toolkit for manipulating tree automata[2]. The tool is implemented in the OCaml programming language. Figure 5.12 on the following page shows timing results for Timbuks' implementation of a determinisation algorithm. According to the author of the Timbuk tool the implementation follows the textbook algorithm. The tool have no built-in option for timing the algorithm so the reported figure are rough measures.

5.9 Type Analysis Tool

A type analysis tool has been developed that uses the deterministic regular types. The type analysis tool includes three main type analysis engines:

Domain model: Given a program and a regular type, the tool computes the least model with respect to the pre-interpretation derived from the type. A BDD-based solver can be optionally used to compute the model, giving greater scalability (at the cost of some pre-processing).

---

[2] http://www.irisa.fr/lande/genet/timbuk/

Name	Timbuk
`chr`	stack overflow
`dnf`	stopped after 5 minutes
`mat1`	instantly
`mat2`	instantly
`ring`	instantly
`pic`	20 secs
`aquarius`	stack overflow
`chatparser`	7 seconds

Figure 5.12: Timing results for Timbuk's determinisation algorithm. If Timbuk returned a result in a second or less it is reported as *instantly*. If no result was reported after 5 minutes the tool was interrupted. Timbuk failed with a stack overflow error for some of the test cases.

Two existing type analysis tools that complements the first tool have additionally been added to the tool kit.

Well-typing: Given a program, the tool computes types and predicate signatures that are a well-typing for the program. This tool was developed by Bruynooghe, Gallagher and Van Humbeeck [25].

NFTA: Given a program, the tool computes a regular type (non-deterministic finite tree automaton) that represents an over-approximation of the least model of the program. This tool was developed by Gallagher and Puebla [75].

These tools are all goal-independent but we have implemented goal-dependent analyses using query-answer transformations (see Section 2.3).

5.9.1 Domain Model

The analysis is based on given regular types as described previously in this chapter. First we look at an example.

Example 24. *A Prolog program for transposing matrices.*

```
transpose(Xs,[]) :-
    nullrows(Xs).
transpose(Xs,[Y|Ys]) :-
    makerow(Xs,Y,Zs),
    transpose(Zs,Ys).
```

```
makerow([],[],[]).
makerow([[X|Xs]|Ys],[X|Xs1],[Xs|Zs]) :-
      makerow(Ys,Xs1,Zs).

nullrows([]).
nullrows([[]|Ns]) :-
      nullrows(Ns).
```

$\square$

As this program is intended to manipulate matrices of unknown type, we define the following types *matrix*, *row* and *dynamic*, expressed as an FTA using the notation selected for the analysis tool.

```
[ ] -> matrix.
[row|matrix] -> matrix.
[ ] -> row.
[dynamic|row] -> row.
```

Our intention is to analyse the *transpose* program in order to discover whether the program's arguments have the expected types. Note that the above FTA is not bottom-up deterministic since there are two transitions with left hand side equal to [ ]. Determinisation of this FTA together with the rules defining *dynamic* for the program's signature, yields the following states and type rules (assuming that the signature is $\{[\_|\_]^2, [\,]^0, 0^0, s^1\}$).

```
[ ] -> {dynamic,matrix,row}
[{dynamic,matrix,row}|{dynamic,matrix,row}] -> {dynamic,matrix,row}
[{dynamic,row}|{dynamic,matrix,row}] -> {dynamic,matrix,row}
[{dynamic}|{dynamic,matrix,row}] -> {dynamic,row}
[_|{dynamic,row}] -> {dynamic,row}
[_|{dynamic}] -> {dynamic}
0 -> {dynamic}
s({dynamic}) -> {dynamic}
```

These rules define an abstraction over the domain elements $\{\{dynamic, row\},$ $\{dynamic, matrix, row\}, \{dynamic\}\}$, which represent the following sets of terms.

- $\{dynamic, matrix, row\}$: the set of terms that are in all the types *matrix*, *row* and *dynamic*. $\{dynamic, matrix, row\}$ is equivalent to *matrix* since *matrix* is a subset of both *row* and *dynamic*.
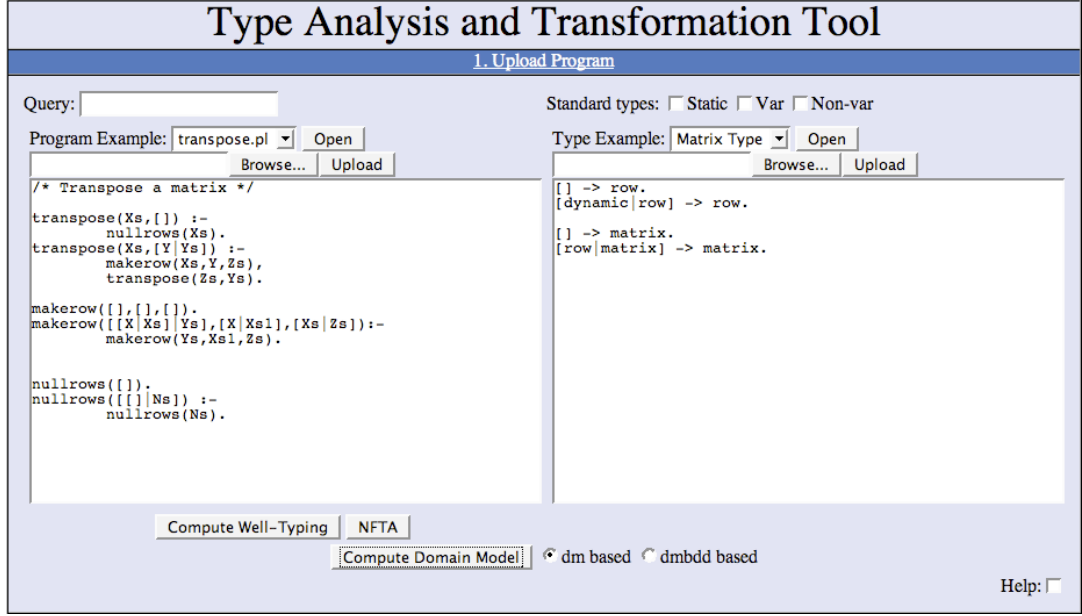
Figure 5.13: The main tool interface showing a program and type ready for analysis

- $\{dynamic, row\}$: the set of terms that are in the types *row* and *dynamic* but not in *matrix*, that is, it is the set of lists whose elements are not all lists.

- $\{dynamic\}$: the set of terms that are in *dynamic* but not in the other types.

These three types are disjoint and complete; every term is in exactly one of these types. We can now compute the minimal model of a program based on this DFTA. In the model of this program over the domain $\{\{dynamic, row\}, \{dynamic, matrix, row\}, \{dynamic\}\}$ defined above, the *transpose* predicate has the model

$$transpose(\{dynamic, matrix, row\}, \{dynamic, matrix, row\})$$

indicating that it can only succeed with both arguments *matrix*. Figure 5.13 displays a screenshot of the tool, showing the *transpose* program, and the regular type defining *matrix* and *row*.

Figure 5.14 shows the results of computing the domain model, for the *transpose* predicate. The model of the predicate appears when the mouse is moved over the symbol to the left of the head of a clause for *transpose*. Note that for brevity the domain elements are numbered and a key is given alongside the predicate's model. Also, the type *dynamic* is omitted since it intersects with every other type,
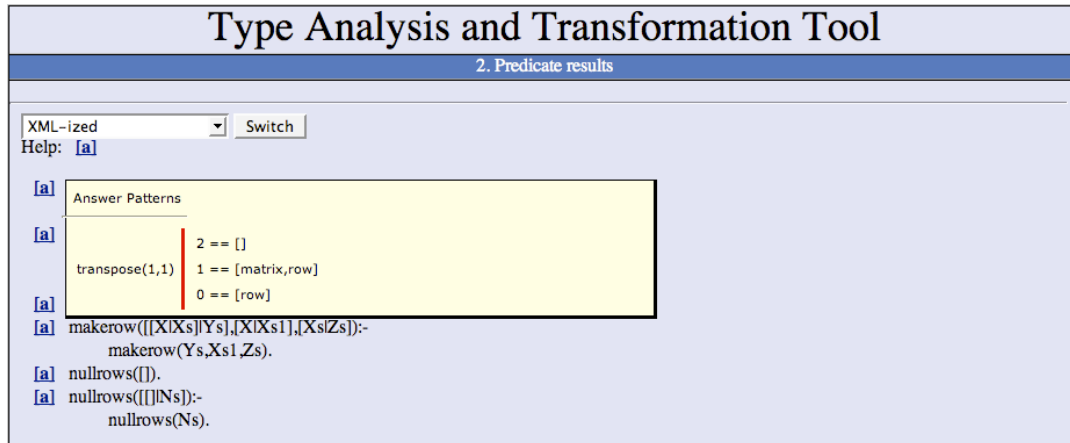
Figure 5.14: Displaying the model of the `transpose` predicate

hence the type [ ] mentioned in Figure 5.14 is in fact the type $\{dynamic\}$. If any predicate has an empty model, then the heads of its clauses are highlighted in red.

Modes as regular types

The type *dynamic* depends on the signature of the program. We call such types *contextual* types [76]. Other contextual types are those defining the set of ground terms (called *static*) or non-variables terms (called *nonvar*). We assume that the signature contains some constant $\$VAR$ that does not appear in any program or query. The rules for contextual types are generated automatically by the system for the given program's signature, and the user does not in fact see these rules at all. The rules defining *static* are all those of the form $f(static, \ldots, static) \rightarrow static$ for each function f in the signature apart from $\$VAR$, while the rules defining *nonvar* are all rules of the form $f(dynamic, \ldots, dynamic) \rightarrow nonvar$ for each function f in the signature apart from $\$VAR$. The rules for *dynamic* do include a rule $\$VAR \rightarrow dynamic$, and thus the types *static* and *nonvar* are not identical to each other or to *dynamic*. A type *var* can also be defined using the single type rule $\$VAR \rightarrow var$.

In the tool, the user can select one or more of the standard types *static*, *nonvar* and *var* and add them to the types to be used for analysis. (The type *dynamic* is always included automatically, to ensure that the types are complete).

Using these "standard" types, the tool can be used to perform the classic mode analyses described in Section 5.6.1; only e.g. g is now called *static*. The naive reverse program is one of the analysis tool's example programs. Selecting

Type Analysis and Transformation Tool

1. Upload Program

Query: [ ]

Standard types: ☐ Static ☐ Var ☐ Non-var

Program Example: transpose.pl ▾ Open

Type Example: Empty Type ▾ Open

Browse... Upload

Browse... Upload

```
/* Transpose a matrix */

transpose(Xs,[]) :-
        nullrows(Xs).
transpose(Xs,[Y|Ys]) :-
        makerow(Xs,Y,Zs),
        transpose(Zs,Ys).

makerow([],[],[]).
makerow([[X|Xs]|Ys],[X|Xs1],[Xs|Zs]):-
        makerow(Ys,Xs1,Zs).

nullrows([]).
nullrows([[]|Ns]) :-
        nullrows(Ns).
```

```
Type definitions
t1(X2) --> [X2|t1(X2)]; []
t3(X2) --> [t5(X2)|t3(X2)]; []
t4(X2) --> [t1(X2)|t4(X2)]; []
t5(X2) --> [X2|t5(X2)]; []

Signatures
transpose(t3(X2),t4(X2))
nullrows(t3(X2))
makerow(t3(X2),t1(X2),t3(X2))
```

Compute Well-Typing   NFTA

Convert to Regular Type

Compute Domain Model   ⦿ dm based ○ dmbdd based

Help: ☐

Figure 5.15: Displaying a well-typing of the `transpose` program

this program and the standard types *static*, *var* and *nonvar* and then computing the domain model, would yield the results for the g, v, any abstract model shown in Figure 5.6 on page 94. User defined types can be added manually allowing the user to combine the standard types with e.g. a type definition for lists so ground lists can be distinguished from non-ground lists, and list terms can be distinguished from non-list terms.

Computing a Well-Typing

This tool, developed by M. Bruynooghe, J. P. Gallagher and W. Van Humbeeck [25], derives from a supplied untyped program a typed program, or more precisely the type definition and predicates making the untyped program well-typed. The tool derives polymorphic types which are more expressive than monomorphic types. The types are automatically generated from scratch. Experimental evaluation of the tool have shown that for certain applications such as termination analysis, the derived types provides termination conditions as good as those obtained using user declared types.

The result of applying this tool to a program is displayed in the type window on the right hand side of the screen (an example is shown in Figure 5.15). The types are parametric. Note that the types inferred for the *transpose* program are in effect the types *matrix* and *row* defined earlier, except that there is a type

112

parameter representing the type of the elements of the rows. Recall that a well-typing does not necessarily represent a safe approximation of the success set of the given program. It only gives types that consistently indicate the way in which the predicates are actually called in the program. As we will discuss in Section 5.9.3 the well-typing can be checked to see whether it is also a safe approximation. Note that there is some duplication in the types; for example $t3$ and $t4$ are renamings of each other, as are $t1$ and $t5$. The determinisation algorithm described in this chapter could be used to identify and eliminate these duplicates. This has not been implemented though.

Computing a Non-Deterministic Regular Type Approximation

The third analysis method is the computation of a non-deterministic finite tree automaton that over-approximates the least model of the given program. This method is described in [75]. The generated types tend to be complex and difficult to read. The most interesting information is usually the emptiness of a type. As with the well-typing, the result is displayed in the type window, and can then be converted to a regular type and used to build a domain model. Conversion to regular type form in this case is simple, as the inferred types are already regular type rules. We just need to remove the rules defining the types of the predicates (which will be recomputed during the domain model construction).

5.9.2 Goal-Dependent Type Analysis

Transformations to allow goal-dependent analysis using a goal-independent analysis tool are well known (see Section 2.3). The common feature of these transformations is the definition of "query predicates" corresponding to the program predicates. The variant we use in the tool constructs a separate query predicate for each body literal in the program. A similar transformation was described in [73]. Programs derived from a query-answer transformation can be hard to read. To make the analysis results easier to read the tool will display the models of the query-predicates at the corresponding body calls in the original program. An example is shown in Figure 5.16. When the mouse is moved over the symbol to the left of each body call, the query patterns for that call are displayed, along with a key to the determinised types, as before.

5.9.3 From Descriptive to Prescriptive Types

As mentioned earlier, a descriptive analysis such as the well-typing or the NFTA analysis constructs types automatically. Apart from providing documentation about the program predicates, these types can be used for prescriptive analysis
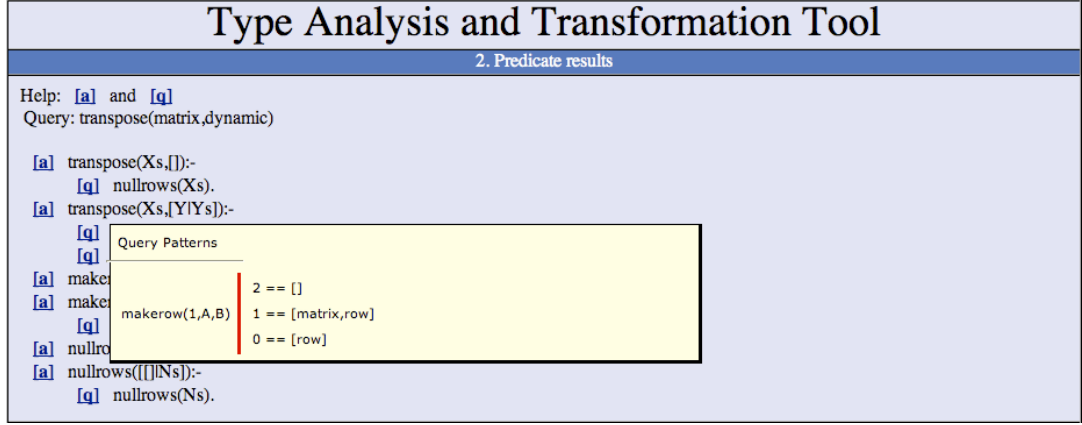
Figure 5.16: Displaying query patterns

where types have to be provided. In [25] the types were used to generate type-based norms for termination analysis. In the present context, the inferred types can be input to the domain model tool. The main reason for doing this is to generate more precise information about failure and dead code in the program. Consider the *naive reverse* program. The well-typing analysis informs us that both arguments of *reverse* are lists, but we do not know whether this is a safe approximation.

A goal-independent NFTA analysis of the same program gives the information that the second argument of *reverse* is approximated by *dynamic*. By taking the result of either of these analyses and using it as the regular type in the domain model tool, we can verify that the second argument of *reverse* is indeed approximated by the type *list*. (We would get the type presented with some system-generated type name such as $[\ ] \rightarrow t1, [dynamic|t1] \rightarrow t1$ which we would have to recognise as *list*. Recognition of standard regular types is already performed by the `CiaoPP` analysis tool [86] and might be added to our tool in the future). This allows us to detect as ill-typed any call to *reverse* whose second argument is typed by a non-list.

As can be seen in Figure 5.15, the tool offers the possibility of using the inferred well-typings or NFTA analysis results to construct regular types. In the case of well-typing rules this just involves replacing the parameters by *dynamic*, which involves a loss of precision. The NFTA types can be used directly with minor syntactic changes. The regular types can then be used to build a domain model, as in Section 5.9.1.

NFTA types tend to be large and complex with much redundancy in the form of multiple occurrences of the same type with different names. So far, the most likely uses of NFTA analysis is to find useless clauses, dead-code, failing calls and

such like. Determinisation and computation of a domain model can enhance the ability to search for such anomalies. An example is provided by the *tokenring.pl* program on the sample program menu, that is an implementation of the token ring transition system described in Section 5.6.2. An NFTA analysis yields types, but every predicate has a non-empty type. If the derived types are converted to a regular type and a domain model is computed, then the fact that *unsafe/1* has an empty type is detected. The BDD-based domain-model tool is required to get this result; the NFTA type rules are complex and the determinised automaton has 111 distinct states.

5.9.4 Features of the Implementation

The Type Analysis Toolkit consists of a back-end and a front-end. The back-end consists of the analysis programs themselves, and the front-end is a user friendly web-interface for those tools. The web-interface serves as a demonstration of the techniques developed in this chapter. It allows readers to verify the results on their own. And finally it allows for future comparison with other type analysis tools without the need for other researchers to install specific software packages on their own machines.

Back-end

There are currently three tools in the back-end: the domain-model analyser, the polymorphic type analyser and the NFTA analysis tool - all written in Prolog. The particular Prolog system we are using is the Ciao Prolog Development System[3].

Initially the back-end tools were developed independently and were intended to be executed within a Prolog environment. They have been modified to allow them to be compiled into two separate command line tools. It is also possible to use the Ciao-shell environment for running the back-end tools from the command line. This method does not require compilation of the tools, but it comes with a performance penalty. Table 5.1 shows the analysis time for a few select programs; the append program, Leuschel and Massart's model checker for CTL formulas [108], and the token ring program mentioned earlier in Section 5.9.3. The table compares a compiled version of the Domain Model (DM) program to a version of DM executed in the Ciao-shell environment. For larger programs the penalty of using Ciao-shell may be insignificant, but for smaller programs the compiled version is significantly faster.

The tools were also modified to read their input and write output to files to simplify the development of the web interface. The output from the polymorphic

---

[3]`http://clip.dia.fi.upm.es/Software/Ciao/`

Program	No. Clauses	Compiled DM	Ciao-shell DM
append.pl	2	0.4	3.7
tokenring.pl	20	0.9	3.5
ctl.pl	27	3.7	7.3

Table 5.1: This shows the analysis time in seconds for a few programs containing from 2 to 27 clauses with respect to the regular type definition of lists

type analyser is plain text. This output will be read by the domain model tool. The output from the domain model tool will be parsed by the front-end, so we decided to format this output in XML. The result from the domain model tool is a term containing the analysed program with clause heads and clause bodies annotated with the query and answer patterns. The terms containing the models resulting from the analysis can be quite large. This term is not a syntactically correct Prolog program, and we have therefore written our own Prolog module to handle the output of terms in an XML structure. Tools for outputting and analysing Prolog code in XML exist [140], but at the moment we only need to separate clauses and calls from annotations when presenting the analysis results in HTML.

Front-end

The invoking of the back-end tools and the presentation of the analysis results are handled by the front-end. The front-end has two main parts; a page with a form to be filled in by the user, where the user uploads, types in or selects one of the supplied example programs and types, and a presentation page where the analysis results are displayed. The front-end is built on the Apache webserver, the PHP scripting language and libxml and libxslt. The design of the pages is inspired by other online analysers for logic programming developed in the context of the Framework 5 ASAP project[4].

The PHP language allows us to create dynamic pages in which, for example, buttons can be removed from the input-form if they do not apply to the current input given. It also enables us to execute the command line analysis tools and run the output through XSLT. XSLT provides a convenient way of automatically transforming the XML output into HTML that can be displayed in a browser. Depending on whether the analysis is performed with or without a query, the appropriate XSL style sheet is applied to the analyser output.

---

http://www.clip.dia.fi.upm.es/ASAP/

[4] http://www.stups.uni-duesseldorf.de/~pe/weblogen/

http://www.stups.uni-duesseldorf.de/~asap/asap-online-demo/

Selection of Program and Analysis Method

An example of the input page is shown in Figure 5.13. On the left side of the screen the user can either paste in a program to be analysed, select one of the example programs or upload a local file containing a program. On the right side of the screen, the user can supply the type, again either from a local file, by selecting one of the provided type definitions or by typing directly into the panel. At the bottom of the screen are the options to either run the Polymorphic Type Analyser, the Domain Model Analyser or the NFTA analysis on the given program. In the case of the Domain Model, the Prolog implementation or the BDD-based tool `bddbddb` can be selected. If the Polymorphic Type Analyser or the NFTA analysis is used, the result is shown to the right. A new option to convert to regular type will appear.

Display of Analysis Results

An example of the output page is shown in Figure 5.14. The analysed program is shown annotated with answer patterns and if a query was supplied to the analysis, also a query pattern for the calls in the body of the program clauses.

Placing the mouse over either of the annotations will show a small window with the actual patterns.

Should a clause have an empty answer pattern it will be coloured red to indicate that it is dead code. If a query was given to the analyser, the code is considered dead with respect to that particular query pattern.

Calls in the body of a query having an empty call pattern are similarly highlighted in red. These are calls that are redundant; they and the calls to their right in the clause can safely be "sliced" from the program, since they are not invoked in the computation of the given query.

The analysis toolkit called TATTOO – Type Analysis and Transformation Tool – is available online. To try it out, visit the URL

$$\text{http://wagner.ruc.dk/Tattoo/}$$

Part II

Analysis of a PIC processor

Chapter 6

Analysing Abstract Machines via Logic Programming

In this chapter we look at constraint logic programming as a tool for analysing other languages, including descriptions of abstract machines and programs for such machines. A specific abstract machine will be selected as a case study, however the method described is not specific to this particular example.

The method described here relies on some existing techniques used in program analysis and specialisation, such as *meta-programming*, *partial evaluation* and *static analysis*. Previous chapters described symbolic and numerical analysis tools and techniques for logic programming. These will be applied here.

A meta-program is a program that has other programs as inputs. In this chapter programs will be manipulated, transformed and analysed, hence meta-programming becomes a central concept. In the context of logic programming, meta-programming has been studied and applied to a variety of languages, especially logic programming itself, since Prolog programs can be represented directly as terms in Prolog (e.g. see description of naive bottom-up interpreter for logic programs in Chapter 4). Using logic programming based meta-programs to study languages other than logic programming has also been studied in the past, but not to the same extent; examples such as [127, 133, 87] have used CLP to analyse imperative programming languages.

Partial evaluation is a program specialisation technique that can be used for such purposes as, but not limited to, generating more efficient programs and automatic generation (compilation) of interpreters. It is the latter property of partial evaluation we will make use of.

Finally static analysis, i.e. techniques for deriving information about a program's run-time behavior without actually executing the program, will be used not only to analyse the CLP program representing the original abstract machine program (which will be called the *object program*), but also to facilitate the specia-

lisation steps needed to derive this CLP program from the original object program. The general framework for static analysis, namely abstract interpretation, has previously been used in combination with partial evaluation, leading to more effective specialisation [93, 103, 129]. The general purpose of program analysis is to determine properties of the analysed program. We will be analysing CLP programs that represent programs written in a different language. However from the analyser's point of view the analysed program is "just" another CLP program that should be analysed as any other program would be analysed. In the context of abstract interpretation any of the abstract domains available for CLP analysis could potentially be applied to the derived CLP program.

6.1   Method Overview

The process of analysing an object program via CLP has two basic steps. Step 1 is to derive from an object program P written in a language L, and a meta-program M designed to "emulate" parts of or the complete semantics of L, a specialised program M_P. The object program and the specialised program can be considered isomorphic if there is a one to one mapping between program points of the object program P and program points of the specialised CLP program M_P. Then Step 2 of the process, applying an analyser to M_P, should provide results that would also hold for the object program P. Finally the analysis results obtained from analysing the derived program M_P must the related to the original object program P. An overview is shown in Figure 6.1.

Chapter Overview

- Section 6.2 describes the PIC microcontroller that will be used as a case study and how it is modeled in CLP.

- Section 6.3 describes how partial evaluation can be used to generate CLP programs that are equivalent to a given PIC program.

- Section 6.4 describes a Datalog based approach to flow analyses and how liveness analysis results can be obtained using a logic program based program transformation method.

- Section 6.5 applies the convex polyhedron analyser to the specialised emulator.

- Section 6.6 shows how an instrumentation of the emulator results in a parametric Worst Case Execution Time analysis.
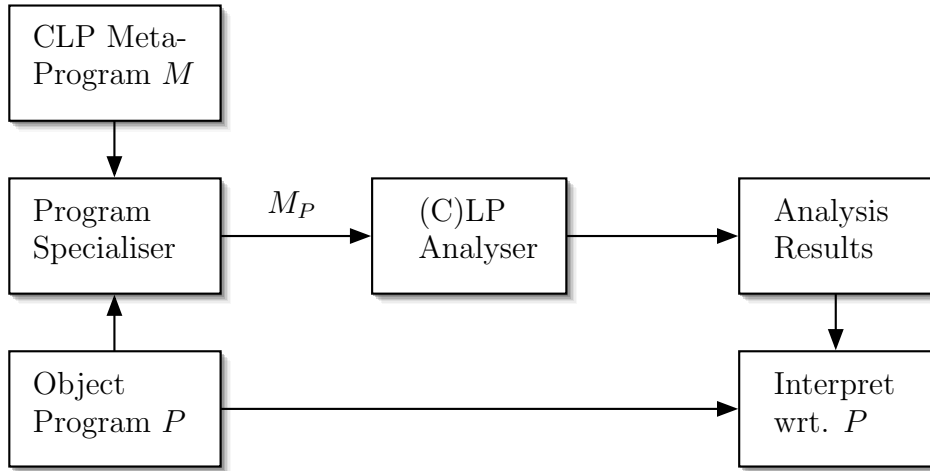
Figure 6.1: Overview of specialisation and analysis process

6.2 PIC Case Study

As a case study for an example abstract machine we have chosen a PIC microcontroller. PIC is the name of a family of microcontrollers manufactured by *Microchip Technology*. They are all based on a *Harvard architecture*, where the data memory is separated from program memory, in contrast to the *von Neumann architecture* where data and program is located in the same memory pool. Among other things the Harvard architecture does not allow for self modifying code. The different models of microcontrollers in the PIC family differs in the size of the data words (8 or 16 bits), the size of data and program memory and what additional instructions they may implement in addition to the basic set of PIC instructions; such additional instructions are e.g. a multiplication instruction. The PIC microcontrollers are used in a variety of devices. Lately the PIC family of microcontrollers have been used in wearable computing, due to its low power consumption [123].

The specific model chosen is the 16F84 [120], an 8 bit processor that has 35 single word instructions, 1024 words of program memory, 68 bytes of data memory, an accumulator (in PIC terms called a working register), an eight-level deep hardware stack and two I/O ports. Additionally there is 64 bytes of eeprom memory that can only be accessed through the hardware registers. The hardware registers contains an 8 bit timer, some status flags and serves as access to the I/O ports. The hardware registers are located at the low 15 registers ordered by memory

address. The data registers follows after the hardware registers in the memory addess space. From the programmer's point of view, there is no difference between the hardware registers and the data registers.

6.2.1   Test case programs

The Mobile and Wearable Computing Group at University of Bristol, United Kingdom, uses the PIC microcontroller for their wearable applications. They have provided us with three test case programs:

Compass This program provides an interface to a compass, detecting which direction the wearer is facing. This program has 141 instructions.

Accelerometer A program providing an interface to an accelerometer that can be used for e.g. counting the number of steps the wearer takes. This program has 215 instructions.

GPS This will let the PIC microcontroller interface to a GPS receiver, so the exact location of the wearer can be determined. This program has 400 instructions.

These 3 programs will be used primarily when evaluating the method described in this chapter.

6.2.2   Modeling a PIC microcontroller

The semantics of the PIC microcontroller is implemented in CLP by means of an emulator, in which the PIC is modelled as a state transition system. The state contains the values of the data registers, the accumulator, the program counter, the stack and the eeprom. The emulator is given an initial state, a program to emulate and possibly some environment data, for instance external input on the input port. Each instruction will, when executed on a given state, produce a new state. The emulator works by executing machine instructions from the supplied PIC program one at a time, each time altering the state according to the instruction at the current program point. The design of the emulator emphasises semantic clarity rather than efficiency.

Small step semantics

When the semantics of a program statement in one language is modelled in another language, it can be described using an operational semantics where the "valid" intepretation of a program statement is described in a sequence of computational steps. This operational semantics can be divided into the *small step semantics* and

the *big step semantics* [139, 126]. The small step semantics describes step-by-step how to evaluate an expression and arrives at a final state value. It gives a clear definition of the order of execution of individual computational steps and is therefore commonly found in the context of modeling concurrency or programs designed to run forever. It may describe the computational steps at an unnecessarily detailed level however.

The big step semantics describes an entire transition from expression to final state value. It describes how overall results are obtained, avoiding unnecessary steps. In some situations this may be a more natural way of describing an operational semantics for a given programming language.

Defining the operational semantics of a programming language of interest, by providing a state transitional system, can be traced back to [128]. This method has previously been used in studying relations between programs, e.g. determining *bisimulation* between state transition system. An informal description of bisimulation would be to say that two systems are bisimular if they match each other's moves.

Modeling the hardware registers of the PIC, e.g. such as capturing exact timing information, would require detailed description of each instruction. This makes the small step semantics more relevant for modeling our PIC case study.

6.2.3   A CLP emulator for PIC programs

The PIC emulator is implemented in Prolog. A predicate called `execute` contains the main loop that executes each instruction. For readability purposes the arguments containing state information have been shortened to just `StateIn` and `StateOut`. A state is a grouping of lists and values combined by the functor *state*; `state(Regs,PC,Acc,Stack)`, where the argument `Regs` contains the state of the data memory, the `PC` is the program counter, `Acc` is the accumulator and `Stack` is a list containing the present stack information.

```
execute(Prog,StateIn,Environment) :-
    fetchinst(Prog,PC,I,R1,R2),
    execInst(I,R1,R2,StateIn,StateT),
    simulatehw(StateT,StateOut,Environment,NewEnvironment),
    execute(Prog,StateOut,NewEnvironment).
```

There exists an `execInst` clause for each machine instruction in the PIC processors' instruction set. The argument named `Environment` contains external data such as values assigned to the processors' hardware input port. This argument can be left uninstantiated. A short explanation of the calls follows

- `fetchinst/5` will fetch the instruction from the program memory that the

125

program counter points to. `R1` and `R2` will be instantiated to the instructions' arguments.

- `execInst/5` will execute the fetched instruction with its arguments on the current machine state producing a new (temporary) machine state.

- `simulatehw/4` will simulate internal and external hardware behavior of the processor e.g. increasing the timer appropriately to the executed instruction and assigning data to the hardware input port. This will produce a new machine state and a new environment state.

A few examples are shown next.

Example 25. *The instruction `addwf` adds the content of the accumulator to a register. The first argument (here `Arg1`) is the register number and the second argument will, for this instruction, determine the destination of the result. If it is '0' the result is stored back in the accumulator, and if the argument is '1' the result is stored back in register `R1`. The instruction is only shown for the case where the second argument is '0'. An almost identical clause exists for the second case.*

```
execInst(addwf,Arg1,0,
        state(RegIn,Stack,PC,Acc),
        state(RegOut,Stack,PCOut,AccOut)) :-
  retrievedata(PC,RegIn,Ru1,Arg1,X),
  intAdd(Wt,X,Acc),
  reduceBits(Wt,AccOut),
  intShiftR(C,Wt,8),
  updateZeroBit(PC,Ru1,Rt,AccOut),
  updateCarryBit(PC,Rt,RegOut,C),
  PCOut is PC + 1.
```

A short description of the calls in the clause body is appropriate

- *The `retrievedata/5` call fetches the value of register given in `Arg1`*

- *`intAdd/3` adds the value to the accumulator*

- *`reduceBits/2` ensures the result is an 8 bit value*

- *`instShiftR/3` is in this case used to calculate the carry bit, but also implements the bit-shift-right operation*

- *`updateZeroBit/4` and `updateCarryBit/4` updates the zero bit and the carry bit in the machine status register*

126

$\square$

Using user defined predicates for arithmetic operations makes it easier to change how they are emulated. For now, no requirement is imposed on how the values of the element in the machine state are represented; most naturally it would be an integer value, but it could also be a list of 8 boolean values representing an 8 bit integer.

Instructions with no arithmetic or boolean operations are much simpler.

Example 26. *The* `goto` *instruction needs no clause body; the first argument* `R1` *is copied to the PC in the new state.*

```
execInst(goto,Arg1,_,state(R,S,PC,Acc),state(R,S,Arg1,Acc)).
```
$\square$

The implementation of the emulator in this fashion is not efficient, but it is however a generic approach. The instruction set and the state information can easily be changed to emulate other processors or abstract machines.

Register State

The data and hardware registers are stored in a single list. The precise content of this list depends on which approach is followed when the emulator is specialised and analysed. The content of this list can simply be a pair of the register number and the current value in the register, for example `[RegNr-Value|Regs]`.

6.3   Specialising the emulator

Writing generalised software usually means trading efficiency for clarity of code. In some cases program specialisation techniques are not used for efficiency reasons. Some techniques have other interesting properties; partial evaluation can for instance be used for compiler generation.

6.3.1   Partial Evaluation

Programs that are purposely written to be highly general are typically also highly parameterised. Applying the program to a set of similar problems would often result in some of the parameters being constant for all input problems, and some of the parameters would differ from problem to problem. Partial evaluation [83] is a source-to-source program transformation technique that exploits the knowledge of some parameters to a program being constant (*static*) for all sets of input data.

The input that can vary is also called *dynamic* input. The *static* input is generally speaking the input which is known before the program execution, while the *dynamic* input is the data that is not known until program execution.

If running the program on a full set of input is called a full evaluation of the program, running the program on a set of partially known input would then be called a partial evaluation of the program. The *static* input would be the known part of the input data, and the *dynamic* input would be the unknown part of the input.

Given a program P and a set of static input data S, a partial evaluation of P with respect to S would result in a new program P_S, the *residual program*, that for the same dynamic input D, would produce the same output as P with inputs S and D. The process is illustrated in Figure 6.2 below. A *partial evaluator* is a program that performs partial evaluation, and in this example an evaluator could for instance be an interpreter.

(a) Normal Evaluation   (b) Partial Evaluation   (c) Evaluation of Specialised Program
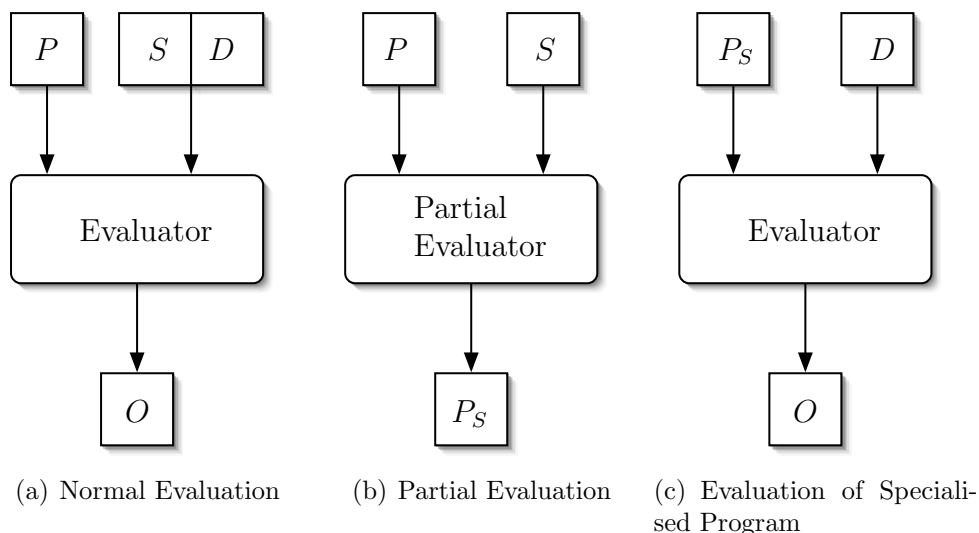
Figure 6.2: Partial Evaluation Overview. P = Program, S = Static input, D = Dynamic input, O = Output and P_S = residual program

A partial evaluator can be viewed as a mix between an *interpreter* and a *compiler*. It interprets code in P depending on static data and generates code in P_S for code in P depending on dynamic data.

Assuming we have a partial evaluator available, we can apply this to a simple program to illustrate what a resulting residual program could look like. A typical example used to illustrate partial evaluation, is the power function, $power(x, y) = x^y$, implemented as a recursive program. If y is even, the result is the square of

$x^{\frac{y}{2}}$, and if y is odd the result is $x * x^{y-1}$.

Example 27. *The power function could be implemented in Prolog in the following manner:*

```
power(_,0,1).
power(X,Y,Z) :-
      even(Y), Yh is Y / 2,
      power(X, Yh, Zr), Z is Zr * Zr.

power(X,Y,Z) :-
      odd(Y), Yp is Y - 1,
      power(X, Yp, Zr), Z is X * Zr.

even(X) :- X > 0, 0 is X mod 2.
odd(X) :- X > 0, 1 is X mod 2.
```

If we make Y static (known at specialisation time) and X and Z dynamic, and we partially evaluate[1] with respect to e.g. Y = 5, we obtain the following residual program:

```
/* power(A,5,B) :- power__5(A,B). */
power__5(A,B) :-
      C is A*1,
      D is C*C,
      E is D*D,
      B is A*E.
```

The recursive calls have been unfolded and all operations only dependent on static input, have been pre-computed. □

In the context of logic programming, partial evaluation is often referred to as *partial deduction* [100]. This is used to distinguish between partial evaluation for pure logic programming, where it is referred to as partial deduction, and logic programming with non-logical features such as asserts and cuts, where it is referred to as partial evaluation.

6.3.2 Strategies for Partial Evaluation

Under normal evaluation of a program, all input is available at every program point. During partial evaluation, all, some or none of the input will be available.

---

[1]In this example WebLogen, a partial evaluator for Prolog has been used.

How the partial evaluator decides which program constructs can be evaluated and which must be residualised, is a complex process. There are two main strategies for performing partial evaluation - *online* and *offline* partial evaluation. In online partial evaluation the decision between evaluating or residualising code in P is taken at specialisation time. In offline partial evaluation these decisions are made based on results obtained from a *Binding Time Analysis* (BTA) performed prior to specialisation.

To begin with, all partial evaluators were online evaluators [92]. When partial evaluation was investigated with regard to its use as a program generator tool, offline implementations were required to allow, for instance, for self application of partial evaluators; when a partial evaluation is applied to itself with an interpreter as static input, the result is a compiler. Applying partial evaluation to an interpreter with a source program for that interpreter as static input, it results in a new version of the source program, implemented in the language of the interpreter. For our purpose partial evaluation is used for specialisation of interpreters, hence offline partial evaluation will be the most relevant.

Online partial evaluation

Online partial evaluators make the control decisions during the specialisation phase. The control decisions will divide the program constructs into those that can be evaluated at specialisation time and those for which residual code must be generated. A program loop is a typical example of a construct that can result in infinite unfolding. Size reduction based on norm sizes is an example of a technique that can be applied to detect safe and unsafe unfoldings.

Offline partial evaluation

In this method of performing partial evaluation, the decision between evaluating program constructs or residualising those constructs is taken offline; meaning in a separate phase before actual specialisation occurs. There are two phases in offline partial evaluation.

1. **Binding-Time Analysis:** In this first phase of the process, the programmer supplies a specification of which parameters will be static and which will be dynamic. The partial evaluator will separate program constructs into two categories; those that can be fully evaluated and those that must be residualised. The specification is usually a number of annotations for the constructs in the program.

2. **Specialisation:** In this second phase, the programmer supplies the actual static input, and the partial evaluator will now follow the guidelines from
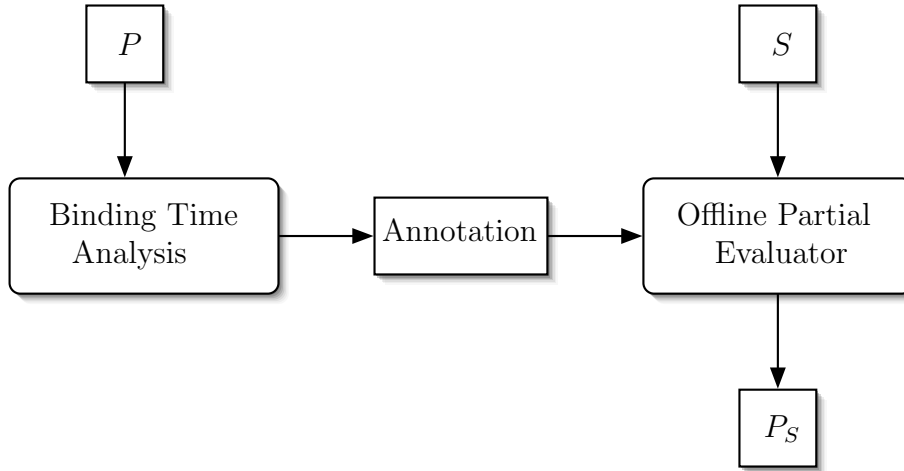
Figure 6.3: Offline Partial Evaluation Overview. P = Program, S = Static input and P_S = residual program

the annotated program, and based solely on these annotations, will separate the program constructs into those that can be fully evaluated and those for which code must be generated and included in the residualised program.

Figure 6.3 gives an overview of the process; after offline specialisation the residual program P_S can be evaluated as shown previously in Figure 6.2(c). With this type of partial evaluation the programmer has more control over which program constructs are residualised and which are eliminated through evaluation. The fact that offline partial evaluation does not depend on actual static input, but only an annotation describing which input *will* be static at specialisation time, can be an advantage of this technique over its online counterpart. Once the annotations have been generated for specialising P with respect to S_1, the same annotations can be used to specialise P with respect to S_2 providing S_1 and S_2 have the same specifications of static and dynamic input, but only differs in the actual values of the static input. This can greatly speed up the specialisation process if P must be specialised for multiple sets of input data.

Strictly speaking online partial evaluation is more powerful than offline partial evaluation for several reasons [82], one being that it makes control decisions based on the actual static input fed to the partial evaluator. In logic programming data can be partially instantiated. For instance a list can contain 3 uninstantiated elements, $[X, Y, Z]$, making the list partially static and partially dynamic. An online specialiser might decide that unfolding this list would be safe. For an offline specialiser the same list would be considered dynamic and therefore not safe to

unfold. Extending the binding types to a richer domain than the static/dynamic domain, as also described in Section 5.6.3, is one method that can be used to strengthen offline partial evaluation.

6.3.3 Partial evaluation of interpreters

Efficiency improvements is not the only application of partial evaluation. The three Futamura projections [68] can be used to

1. achieve compiling by specialising an interpreter

2. generate compilers by self-application

3. generate compiler generators by double self-application

It is the first Futamura projection, specialising interpreters, we will make use of. This is a key application of partial evaluation. In this situation the interpreter is the program that will be specialised, the object program is the static input and the call that begins the execution of the object program in the interpreter, is the dynamic input. The result is a version of the interpreter that is less general in the sense that it no longer interprets any object program, but only the object program it was specialised with respect to.

The specialised interpreter will in general be more efficient in terms of run-time behavior.

Particularly offline partial evaluation has proven to be very useful for specialising interpreters [104]. To specialise our PIC emulator we will use the Logen [107] offline partial evaluator for logic programs developed by Michael Leuschel and Jesper Jørgensen.

Specialising the PIC emulator

To use an offline partial evaluator to specialise the emulator described in Section 6.2.3 poses one problem that must be solved; it must be possible to determine the program counter (PC) statically. In all PIC instructions, with the exception of the `call` and `return` instructions implementing subroutines, the PC is calculated in the individual instructions. This calculation of the next PC is only possible during specialisation, if the PC is static. When a subroutine is invoked by a `call`-instruction, the program point following the `call`-instruction is pushed onto a stack. Upon exit from the subroutine by a `return`-instruction, the stack is popped and control is returned to the program point at the top of the stack. There may be several calls to the same subroutine at different program points. Hence the next instruction to be executed after a `return`-instruction is not in general known

at specialisation time. This can result in total loss of specialisation following a `return`-instruction.

For now we assume that a set of facts exists, that describes the control-flow of the program, i.e. which program points can follow a given program point. These facts are of the form `nextInstr(PC,PCnext)`. How these facts can be obtained from a Control Flow Analysis is described later in Section 6.4.

The emulator is specialised with respect to a given program and a set of control-flow facts. The PIC program, the control-flow facts and any environment data supplied are static inputs. In the `execute`-loop, everything is unfolded except the loop itself. Every `execInst` is unfolded completely; arithmetic operations can be annotated as *rescall*, which will preserve the call in the residual program. Arithmetic operations on elements in the machine state are generally marked *rescall*, with exceptions such as the program counter. In the later analysis step where a CLP analyser will be applied to the residual program, the analysis will be based on those rescall'ed arithmetic operations.

The `return`-instruction is unfolded with respect to the `nextInstr/2`-facts. The stack content will not be known at specialisation time, but the unfolding of the `nextInstr/2`-fact will ensure that control flow is returned to any possible instruction that could occur at the top of the stack, at that particular `return`-instruction. The implementation of the `return`-instruction is shown below.

```
execInst(return,_,_,
         state(RegIn,StackIn,PC,Acc),
         state(RegIn,StackOut,PCOut,Acc)) :-
    popstack(StackIn,StackOut,PCOut),
    nextInstr(PC,PCOut).
```

The result of this specialisation is a new Prolog program, with a numbered `execute`-predicate for every program point in the original PIC program. The control flow of the PIC program is now embedded in the calls from one `execute`-clause to the next. PIC instructions that can alter control flow of the program will have more than one version of a given numbered `execute`-clause. Instructions modifying the hardware status register (containing among other things, the carry and zero bit) will also have more than one version, depending on whether the results would set or not set a bit in the status register.

Example 28. *An example of such an `execute`-clause after specialisation is shown below; this particular clause corresponds to the `addwf` instruction whose `execInst`-implementation was shown earlier in Example 25.*

133

Program	No. Instr.	No. Clauses	Size (kb)	Specialisation Time (sec.)
Compass	141	199	40	13
Accelerometer	215	274	46	18
GPS	400	631	183	42

Table 6.1: Specialisation of PIC programs

```
execute__5(S,Q,O,M,K,I,G,E,D,B,C,_,F,H,J,L,N,P,R,A) :-
      T is Q+A,
      O is T>>8,
      T \== 0,
      _ is 24 /\ 251,
      U is 24 /\ 254,
      _ is _+1,
      execute__6(S,Q,O,M,K,I,G,E,D,B,C,U,F,H,J,L,N,P,R,T).
```

Two status bits can be modified, so 4 different versions are generated; control flow of the program can only pass to the following instruction, so all `execute__5s` *will continue with* `execute__6`. □

Resulting specialised programs

Some results acquired from specialising the three test case programs from Section 6.2.1 is shown in Table 6.1. The number of predicates in the specialised program is omitted, since there is a one-to-one mapping between the instructions in the object program and the predicates in the residual program. The number of clauses will depend on e.g. the number of instructions modifying control-flow. The table shows that this increase is in the range 27% to 60% for the example programs. The size of the residual program reflects how large the programs for later analysis will be. The timing results show how computationally expensive this step in the process is. The results were collected using the command line version of Logen on a Linux machine equipped with a 900MHz Intel Pentium III CPU and 256MB RAM using the command line tool "time" measuring the user CPU time.

An earlier version of the PIC emulator has been used as a benchmark for, and a demonstration of, the applicability of the Logen specialiser [105, 106].

The next section will describe a method for obtaining the control-flow facts used in the specialisation process.

6.4 Flow Analysis

A general approach to Data Flow Analysis (DFA) using logic programming is described in this section. Since the approach is based on general techniques and tools it is therefore not specific to a particular language or abstract machine, and the method described here can easily be carried out on other languages or abstract machines.

Expressing a program analysis declaratively in a logic programming language has a few advantages; the implementation of the analysis is greatly simplified and analyses described in a few lines of Datalog can take hundreds of lines to write in a traditional language. Among other things this minimises the risk of introducing errors in the analyser. When the analysis is expressed in a uniform manner it becomes easier to combine it with other analyses [153].

Overview

Given a language to analyse we annotate the instruction set with a set of facts describing their control flow behavior. The program to analyse with respect to is appended as another set of facts. Then we write rules to determine properties of that particular program.

The facts associated with the instructions, the program, and the rules, make up a Datalog program. Efficient methods exist for solving Datalog program such as the BBD-based method described in Section 5.7.2. We will use the same method here.

6.4.1 Data Flow Analysis

This section will present an overview of classical Data Flow Analysis concepts. The notation is based mainly on [99, 97, 134]. Data Flow Analysis (DFA) is in the area of static program analysis. It focuses on gathering information about the definition and use of data objects in a program, without executing the program itself. For imperative languages it provides information about possible program states that may occur at some program point during execution. A classical DFA is based on two things: an abstraction of the data transformations in the program, and the propagation of this information through the control flow graph of the statements in the program.

The obtained information can be used for a variety of purposes, for instance debugging, program optimisation and so on. Data Flow Analysis is frequently found in compilers where it can be used to generate faster code, save memory etc.

A number of properties about a given program can typically be detected using DFA. Properties can hold for program points and program paths, where a program

path is a possibly infinite list of program points having an entry program point and where each successive program point is immediately reachable from the previous program point in the list. These properties can be split into groups that depend on a program point's *history*, namely those program paths reaching it, and properties that depend on a program points *future*, which are program paths starting from that program point. Some properties will be valid if some property holds for *some* path in the history or the future of a given program point, making them *existentially* quantified properties, and some will only hold if the property holds for *all* paths, making them *universally* quantified properties.

Examples of properties that can be detected using DFA are:

- **Liveness:** Data is only live if some future path depends on its value. This property is typically used for program optimisation. Instructions in the program that calculate dead data can typically be eliminated from the program, and the memory that holds the dead object can be used for other (live) objects, thereby saving space.

- **Undefined values:** The use of data objects with no assigned value can lead to propagation of undefined values in the program, and bugs that can be hard to detect. This property can be used to warn the programmer of possible bugs in the program.

Flow graphs

Control Flow Analysis (CFA) is another static analysis method that is needed in DFA. The DFA will typically be performed on the Control Flow Graph (CFG) representation of the program. In this more abstract representation, each statement in the program has its own node, and the edges corresponds to the control transitions. There is a node marked *start* in the graph. When the CFG is used for Data Flow Analysis, every node will have a transfer function that specifies how data is propagated through the program. As with the DFA, CFA can be used to detect properties that can lead to program optimisation, debugging information etc. Examples of such properties are:

- **Dead Code:** Program points that can never be reached by any path in the program starting from the node marked 'start', are dead and can be eliminated from the program.

- **Worst Case Execution Time (WCET):** The programmer may wish to know whether parts of the code will complete within some time constraints. WCET can be used to give estimates of the execution time of parts of the program.

We will use the following notation and definitions of properties relating to the CFG:

Definition 53. $G = (N, E, s)$ *is a directed graph with nodes N, edges $E \subseteq N \times N$ and starting node $s \in N$.*

Definition 54. *The immediate predecessors of a node n are denoted and defined as*

$$Pred(n) \stackrel{def}{=} \{m | (m, n) \in E\}$$

Note: $Pred(s) = \emptyset$

Definition 55. *The immediate successors of a node n are denoted and defined as*

$$Succ(n) \stackrel{def}{=} \{m | (n, m) \in E\}$$

Definition 56. *A path p from node n_1 to n_k is a sequence of nodes such that $(n_i, n_{i+1}) \in E$ for $1 \leq i \leq k - 1$. $\mathcal{P}(n)$ denotes the set of all paths from s to n.*

Data Flow Framework

A Data Flow Framework, $\langle G, D, F, [\![.]\!], \bot \rangle$, consists of the following items

- $G = (N, E, s)$ a control flow graph

- D: a semi lattice $\langle \mathcal{D}, \sqsubseteq, \sqcap \rangle$ of abstract values describing program states

- $F \subseteq \{f | f : \mathcal{D} \to \mathcal{D}\}$ a set of functions

- $[\![.]\!] : N \to F$ are transfer functions for G. A transfer function associated with a particular node n will be denoted f_n.

- $\bot \in \mathcal{D}$: a description that holds for $s \in G$

D is intended to express the relevant data flow information. The *path semantics* of G is given by the function $[\![.]\!]$ which assigns meaning to every node $n \in N$ in terms of the transfer functions.

Definition 57 (Path semantics). *For every path $p = (n_1, ..., n_k) \in \mathcal{P}(n)$, the path semantics is defined recursively as a composition of transfer functions:*

$$[\![p]\!] = [\![(n_2, \ldots, n_k)]\!] \circ f_{n_1}$$

Two strategies exist for obtaining the abstract semantics. The *meet over all paths* (MOP) and the *maximal fixed point* (MFP) [94].

The MOP-strategy meets (or intersects) all information of all possible executions of program paths reaching a given program point of interest.

Definition 58 (Meet over all paths).

$$MOP(n) = \bigsqcap_{p \in \mathcal{P}(n)} [\![p]\!](\bot)$$

If there are loops or recursion in the program, $\mathcal{P}(n)$ can be infinite. If $\mathcal{P}$ and $\mathcal{D}$ are infinite, it is generally not feasible to solve MOP. Instead the MFP-strategy can be used. This strategy is in general a safe approximation of the MOP-strategy.

Definition 59 (Maximal Fixed Point).

$$MFP(s) = \bot$$

$$MFP(n) = \bigsqcap_{m \in Pred(n)} f_n(MFP(m)) \; for \; n \neq s$$

The MFP-semantics coincides with the MOP-semantics when the functions specifying the abstract semantics of statements are distributive - a result known as the Coincidence Theorem [99]. Solving a DFA problem can be regarded for practical purposes as computing an MFP.

In the following section we will describe our approach to performing Data Flow Analysis of abstract machines.

6.4.2 Datalog model

In this section we explain how an abstract machine can be modelled in a Datalog program and how adding Datalog rules will be sufficient to obtain a Data Flow Analysis Framework for this abstract machine.

In the Datalog program each instruction of the machine is given a unique number and for each instruction we annotate it with a set of facts that describe the behavior of that particular instruction. The instruction can behave differently depending on its arguments. In the following we assume that the instruction set allows for 0, 1 or 2 arguments. The facts are triples of the form `someFact(I,A1,A2)`, where `I` is the instruction number, `A1` and `A2` are the arguments. For the properties that do not depend on the arguments to the instructions, the associated facts are of the form `someFact(I)`.

We can categorize the facts such as those describing the control flow of the program, access to registers and timing information.

- **Control flow:** In the control flow category, the PIC instructions can be one of five different types:

138

– Straight line instruction: Upon execution of such an instruction, the control flow of the program continues with the next instruction in the program.

– Branch instruction: When a branch instruction is executed, either the following instruction in the program is executed or it will be skipped and the second following instruction will be executed.

– Goto instruction: Control flow of the program will continue with the instruction number given as an argument to the `goto`-instruction.

– Call instruction: Same as a `goto`-instruction, but it is distinguished from the gotos since the matching return instruction can redirect control flow of the program, to the instruction following the call.

– Return instruction: The instruction to be executed after a return instruction will depend on the call context of that return instruction.

- **Register access:** Instructions can access the registers or other parts of the machine state like the accumulator. It can either be a read or a write access. Some instructions will both read and subsequently write to a register, making it a both read and write instruction. Of course some instructions access neither registers nor the accumulator.

- **Timing information:** Execution of an instruction can take a different number of clock cycles, depending on the instruction. Annotating the instructions with precise timing information can be used later to calculate e.g. WCET.

All programs are assumed to start at program point 0, so we add that as a fact as well; `entryPoint(0)`.

The program is encoded as a set of facts of the form `pp(PC,I,A1,A2)`, where `PC` is the program memory location of that instruction, `I` is the instruction number and `A1` and `A2` are the arguments to that instruction.

Initially we do not allow for arithmetic expressions in our set of rules and facts, i.e. we restrict ourselves to pure Datalog. A simple increment operator will be need for the Control Flow Analysis described in Section 6.4.4. A set of increment facts are added to the program created so far. The increment operator in Datalog is defined as facts of the form `increment(0,1)`, `increment(1,2)` etc. up to the maximum number of instructions in the program.

6.4.3 Datalog rules

Once every instruction has been annotated with the facts described above, and the program has been appended, rules can be written to detect properties of the associated program.

Example 29. *Program point* `N` *contains a branch instruction if the instruction located in the program memory at location* `N` *contains an instruction that was categorized as a branch instruction. This translates into the rule*

```
branchInstr(N) :- pp(N,I,_,_), branchInst(I).
```
□

We will show how these rules can be reused when more complex rules are needed to describe the properties mentioned in the previous section.

6.4.4 Control Flow Analysis

The control flow facts can be used to reason about the control flow of the program. A *next instruction* procedure can be defined in terms of the facts from the control flow category. For each type of fact, a rule like the one shown in Example 29 is created. The *next instruction* procedure has a rule for each type of fact.

```
nextInstr(N1,N2) :-
                straightLineInstr(N1),
                increment(N1,N2).
nextInstr(N1,N2) :-
                branchInstr(N1),
                increment(N1,N2).
nextInstr(N1,N3) :-
                branchInstr(N1),
                increment(N1,N2),
                increment(N2,N3).
nextInstr(N1,R1) :-
                gotoInstr(N1,R1).
nextInstr(N1,R1) :-
                callInstr(N1,R1).
nextInstr(N1,N3) :-
                returnInstr(N1),
                calledFrom(N1,N2),
                increment(N2,N3).
```

The branch instruction has two rules. Conditions for branch instructions are not taken into account, and control flow can therefore either continue with the following or the second following instruction. Depending on which abstract machine is being modelled, e.g. branch instructions can behave differently. This particular behavior corresponds to the PIC microcontroller used as a case study.

The *called from* procedure determines the *call contexts* for a given instruction. A *call context* corresponds to the latest `call` instruction that was encountered in the control flow of the program. The *called from* procedure is defined in terms of

the facts from the control flow category and the following rules:

```
calledFrom(0,0).
calledFrom(N1,N2) :-
                increment(N3,N1),
                straightLineInstr(N3),
                calledFrom(N3,N2).
calledFrom(N1,N2) :-
                increment(N3,N1),
                branchInstr(N3),
                calledFrom(N3,N2).
calledFrom(N1,N2) :-
                increment(N3,N1),
                increment(N4,N3),
                branchInstr(N4),
                calledFrom(N4,N2).
calledFrom(N1,N2) :-
                gotoInstr(N3,N1),
                calledFrom(N3,N2).
calledFrom(R1,N1) :-
                callInstr(N1,R1).
calledFrom(N1,N2) :-
                increment(N3,N1),
                callInstr(N3,_),
                calledFrom(N3,N2).
```

All programs are assumed to start at instruction at memory location 0, so initially the call context is 0. Since a subroutine can be called from more than one place in the program, an instruction can belong to more than one call context. The last `calledFrom`-rule states that any instruction following a call instruction belong to the same call context as the call instruction does. This is similar to the first rule stating that any straight line instruction belong to the same call context as the instruction preceding it.

Dead code detection

Dead code are instructions in the program that will never be executed. A more optimal use of the program memory can be achieved by eliminating these instructions. An instruction is dead if it is not reachable from any trace in the program starting at program point 0.

First we define the relation `prevInstr(N,Np)` by reusing the previously defined relation `nextInstr(N,Nn)`. This rule in fact only increases readability and is

141

strictly not necessary.

```
prevInstr(N,Np) :-
                nextInstr(Np,N).
```

The dead instruction rule can now be expressed the following way:

```
reachable(0).
reachable(N1) :-
                prevInstr(N1,N2),
                reachable(N2).
unreachable(N) :-
                pp(N,_,_,_),
                !reachable(N).
```

We introduce the exclamation mark as *stratified negation* (see Section 2.4). The rule states that reachable instructions are those that can be traced back to program point 0, and the unreachable instructions are any instruction in the program that is not reachable.

Register Access

A register R is *defined* when it is assigned a value (written to) and *referenced* when the value in R is used for other computations (read). We write rules for finding instructions that read or write to a given register.

```
writeInstruction(R,N) :-
                pp(N,I,R,R2),
                writeInst(I,_,R2).
readInstruction(R,N) :-
                pp(N,I,R,R2),
                readInst(I,_,R2).
```

Program history and future

Writing rules for finding a program's history and future at a given program point n is simple enough. A program point m is in program point n's history *if* m is an immediate predecessor *or* program point l is an immediate predecessor of n *and* m is in the history of l. This can be described more formally:

$$m \in history(n) \Leftarrow m \in pred(n) \vee l \in pred(n) \wedge m \in history(l)$$

142

Writing this property in our rule notation, we get the following procedure:

```
history(N,M) :-
                prevInstr(N,M).
history(N,M) :-
                prevInstr(N,L),
                history(L,M).
```

The `future` relation can be defined in terms of the history - a program point m is in the future of n if n is in the history of m, more formally

$$m \in future(n) \Leftarrow n \in history(m)$$

Or in our rule notation, simply

```
future(N,M) :-
                history(M,N).
```

Determining whether a particular property $P(n)$ holds in a program point's future

$$P(n) \Leftarrow \exists m \in future(n), P(m)$$

would for example, for the property is a value written to a register at program point N possibly read later in the program, simply becomes the rule

```
possiblyUsed(R,N) :-
                future(N,M),
                readInstruction(R,M).
```

Program paths

Properties of interest must typically hold for some *path* in a program; e.g. a register will be live at program point n, if it is referenced at program point m in n's future *and* not defined in *some* path from n to m.

There is a difference between writing rules that must hold in either history or future, and rules that must hold for paths. If the program has a finite set of program points, history and future will themselves be finite sets. The set of paths on the other hand will be infinite if loops are allowed.

Bit vector problems

Bit vector problems are a group of important DFA problems that are found in most program optimisations used in practice. Each path in the history and/or future

of the given program point is reduced to a single bit value - does a given property hold for this path. The solution to the problem is obtained by applying either the conjunction or the disjunction as the "meet" operator over the bit vector. A previously mentioned property, *liveness*, is an example of a bit vector problem. This problem is a future dependent existentially quantified problem - it must hold for *some* path in the future. A general approach for writing rules for bit vector problems can be describe in the following way:

Let P be some property of a program point, N a program point of interest, and t a path (or trace) in the history of N. The general rule to determine if P holds at some point on some path in the history of N:

$$holdsSome(P, N) \Leftarrow P(N)$$
$$holdsSome(P, N) \Leftarrow prev(N, Np) \wedge holdsSome(P, Np)$$

The general rule to determine if P holds on every point on some path in the (reachable) history of N:

$$holdsAll(P, N) \Leftarrow entryPoint(N) \wedge P(N)$$
$$holdsAll(P, N) \Leftarrow prev(N, Np) \wedge P(N) \wedge holdsAll(P, Np)$$

Example 30. *The rules for detecting a write free path (a holdsAll) or a path with at least one read (holdsSome) from* N *to* M *can be expressed in the following way:*

```
writeFreePath(R,N,M) :-
                nextInstr(N,M),
                !writeInstruction(R,M).
writeFreePath(R,N,M) :-
                nextInstr(N,Ni),
                !writeInstruction(R,Ni),
                writeFreePath(R,Ni,M).
readPath(R,N,M) :-
                nextInstr(N,M),
                readInstruction(R,M).
readPath(R,N,M) :-
                nextInstr(N,Ni),
                readPath(R,Ni,M).
```

These rules states that there is a write free path from N *to* M*, if there exists a series of instructions from* N *to* M *such that all instructions on this path, are not write instructions - and there is a path from* N *to* M *with a reference of* R*, if some instruction on this path is a read instruction.* $\square$

Liveness

The liveness property can now be defined in terms of the previously defined rules.

```
live(R,N) :-
               future(N,M),
               readInstruction(R,M),
               writeFreePath(R,N,M).
```

This rule states that register `R` is live at program point `N`, if for any read instruction of `R` in the future of `N`, there is a path where `R` is not assigned a new value.

Uninitialised registers

Propagation of values from uninitialised data registers can lead to program bugs that can be hard to locate. This behavior can be defined similar to the liveness.

```
initialised(R,N) :-
               entryPoint(M),
               !writeFreePath(R,M,N).
```

A register is uninitialised at `N` if it is not initialised; or simply

```
uninitialised(R,N) :-
               !initialised(R,N).
```

Redundant code

If a register is assigned a well defined value at program point `N`, but this value is never referenced in any future program path, the data assignment at program point `N` is redundant and the program can possibly be optimised by eliminating `N`. Dead assignments can arise from e.g. register initialisations - if no reference of a register can possibly occur before any assignment of data to it, initialisation is

redundant.

```
nextWrite(R,N,Nw) :-
                future(N,Nw),
                writeInstruction(R,Nw),
                writeFreePath(R,N,Nw).
readBeforeWrite(R,N,Nw) :-
                nextWrite(R,N,Nw),
                readPath(R,N,Nw).
redundantAssignment(R,N) :-
                writeInstruction(R,N),
                !readBeforeWrite(R,N,_).
```

Datalog program

When the program to be analysed is merged with the facts for each instruction
of the abstract machine and the rules for each property of interest, they make up
a Datalog program. In this Datalog program, all of the elements of a traditional
Data Flow Analysis Framework are present. The Control Flow Graph is present in
the form of the `nextInstr/2`-procedure. Each result for this procedure represents
an edge in the graph. The nodes are the program points, and the starting node is
represented by the Datalog fact `entryPoint/1`. The abstraction of the Data Flow
(lattices and transfer functions) are present in the facts of each instruction and
rules for each property of interest.

How to obtain results from a Datalog program is described back in Section
5.7.2. The same BDD based procedure can be used for Property Programming.

Applying Property Programming to PIC case study

The `nextInstr/2` rule provides the missing CFG information needed for the offline
partial evaluation as described in Section 6.3.3.

Applying the dead code rules described in Section 6.4.4 to the three program
selected for the case study, provides the results shown in Table 6.2. The 34 re-
dundant instructions in the Accelerometer program turn out to be never called
subroutines copied from the GPS program. The two redundant instructions in
the GPS program are two `goto`-instructions that can never be reached. The test
results were collected on a Linux machine equipped with a 900MHz Intel Pentium
III CPU and 256MB RAM using the command line tool "time" measuring the user
CPU time. Both control flow and data flow rules are solved at the same time.

146

Program	No. Instr.	Dead Instr.	Analysis Time (sec.)
Compass	141	0	1.8
Accelerometer	215	34	1.5
GPS	400	2	3.0

Table 6.2: Dead code analysis of PIC programs

6.4.5 Register Remapping

The liveness information obtained by the Data Flow Analysis can be used to optimise the object program. This section gives a demonstration of how a logic programming implementation of a textbook graph algorithm can be used to optimise the memory allocation of the case study programs. Reallocating registers using a graph algorithm applied to the liveness data for a given program [30] is a well established method. This technique has later been refined [22, 23, 44].

A proper coloring of a graph is, in short, an assignment of a color to every node in the graph so that no two adjacent nodes have the same color. Adjacent meaning there exists an edge between the two nodes. The graph coloring problem is to color the graph using the least number of colors. The graph coloring problem was also on Karp's list of 21 NP-complete problems [95].

The output from the Property Programming based Data Flow Analysis is a set of facts, `live(R,N)`, denoting that register `R` is live at program point `N`. Based solely on these facts a graph coloring algorithm can provide a remapped memory allocation that will use at most the same number of data registers as in the analysed program.

The graph coloring is applied to an *inference* graph representation of the liveness data. Each register is assigned its own node in the graph. If register r_1 and r_2 are live at the same program point, i.e. r_1 infers with r_2, there is an edge (r_1, r_2) in the graph indicates that these two registers cannot be remapped to the same register. The resulting coloring of the graph yields a more optimal register allocation if it can be colored using a lower number of colors than the number of registers actually used in the analysed program.

Algorithm 6 shows a simple graph coloring algorithm. The node with the highest degree, i.e. most connected, is colored first. Following iterations will try to color the remaining nodes using a color already used. If that is not possible a new color is introduced. This procedure is reiterated over the uncolored part of the graph until all nodes are colored so no adjacent nodes have the same color. This algorithm is not guaranteed to provide a solution using the least possible number of colors.

Applying this procedure to the test case program for the PIC processor gives

147

Algorithm 6 Graph Coloring Algorithm

Input: Liveness inference graph $G = (Nodes, Edges)$
Output: Coloring of $Nodes$

colorGraph([ ],[ ],[ ]).
colorGraph(G, [($RemovedNode,ThisColor$)|$ColoredNodes$], $ColorsUsed$) ←
    assignDegrees($G,NodesDegrees$),
    removeLeastDegreeNode($G,NodesDegrees,ReducedGraph,RemovedNode$),
    colorGraph($ReducedGraph,ColoredNodes,ColorsToUse$),
    colorNode($G,RemovedNode,ColorsToUse,ThisColor$),
    addColorIfNew($ThisColor,ColorToUse,ColorsUsed$).

Program	Used Data Reg.	Remapped Reg.	Reduction pct.
Compass	3	2	33%
Accelerometer	5	3	40%
GPS	23	20	13%

Table 6.3: Remapping registers of PIC programs

the results shown in Table 6.3. For all test case programs there is a more optimal assignment of data memory compared to that assignment used by the programmer. The process of remapping registers can also be made transparent to the user once the instruction set has been annotated and data flow rules specified for a given microcontroller.

6.4.6 CFA/DFA Analyser Tool

Once the instruction set of the target machine has been annotated and rules written for the flow analyses, the procedure can be made fully automatic with no other user involvement than that of supplying a program for analysis. A web based front end has been constructed to demonstrate this. For a given PIC program, the dead code analysis from Section 6.4.4 and the register remapping from the previous section can be performed in one go.

First, the source PIC program is uploaded. Then some statistics are shown for the program, as shown in Figure 6.4 on the facing page. Then the program is shown with the dead code highlighted in red, as shown in Figure 6.5 on page 150.

A demonstration of tool is available online at

http://wagner.ruc.dk/PIC/Liveness/
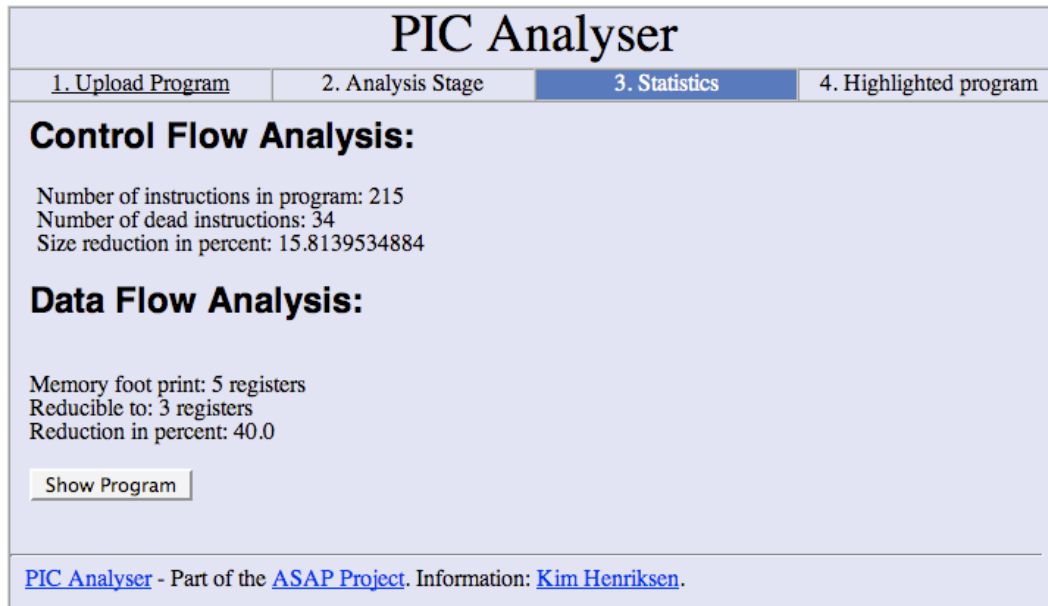
Figure 6.4: Statistics shown for the accelerometer test case program

6.4.7 Program Transformation Based Approach to Liveness Analysis

The specialised emulator resulting from the partial evaluation described in previous sections is an automatically generated program. Such automatically generated programs often contain redundant parts that can be eliminated by general purpose program transformations without affecting the correctness of the program. The clause shown in Example 28 on page 133 contains various redundancies including a large number of arguments for the predicate `execute__5` and some redundant operations in the clause body.

After specialisation of the emulator the semantics of the PIC program is now embedded in the `execute`-predicates of the specialised version of the emulator. The machine state is embedded in the arguments of the `execute`-predicates. To simplify later specialisation and analysis steps, the machine state can be reduced to only the live state. Eliminating dead elements of the state will not alter the semantics of the program.

The results from the Property Programming based technique could also be used to reduce the machine state by eliminating the registers containing dead data. If the `live/2`-facts were available at the partial evaluation step, the list containing the machine state could be unfolded so only the live registers were kept in the predicate head.

149

```
IN_CIR MOVF SER_RX,0
XORLW 'I' ; test if I symbol (inside)
BTFSS STATUS, 2
GOTO OUT_CIR

CALL RX_MASK ; receive filter mask from PC i.e. Long/Lat
LOOP1 CALL CALC_SP ; calc separation of mask and fix (gets fix aswell)
; test for inside
GOTO LOOP1
GOTO MAIN


OUT_CIR MOVF SER_RX,0
XORLW 'O' ; test if O symbol (outside)
BTFSS STATUS, 2
GOTO MAIN

CALL RX_MASK ; receive filter mask from PC i.e. Long/Lat
LOOP2 CALL CALC_SP ; calc separation of mask and fix (gets fix aswell)
; test for outside
GOTO LOOP2
GOTO MAIN


; Loop back and wait for the next string from the PC
```

PIC Analyser - Part of the ASAP Project. Information: Kim Henriksen.

Figure 6.5: Dead code highlighted for the GPS test case program

This section will describe a more general logic program transformation technique called *Redundant Argument Filtering* that, when applied to the specialised emulator, results in a new version of the specialised emulator, where only the live machine state is kept. The technique generally applies to any logic program and is not specific to our specialised emulators.

6.4.8 Liveness Analysis Using Redundant Argument Filtering

Leuschel and Sørensen [109] proposed a general logic program transformation technique called *Redundant Argument Filtering* (RAF). This program transformation removes predicate arguments that are never "used". Propagation of dependency information can either happen in a top-down fashion, from clause head to clause body, or in a bottom-up fashion from clause body to clause head. The top-down propagation is called Redundant Argument Filtering (RAF) and the bottom-up propagation is called Reverse Redundant Argument Filtering (FAR). Either one or both of the methods can be applied to a program. An optimal solution can be reached by iterating the procedure over a program, and in each iteration switching between the two techniques, until a fix point is reached.

The main motivation for the transformation was the simplification of programs produced by other transformations, in particular by conjunctive partial deduction [60]. We focus here on the transformation called Reverse Redundant Argument Filtering (Section 5 of [109]). In this section we show how the FAR algorithm is a generalisation of liveness analysis and can be applied to the specialised PIC emulator to eliminate dead registers at each program point. The result is a simplified program for later program analysis. The filtered program will be both smaller with regards to lines of code and the arity of the predicates in the program. The complexity of later analyses applied to this filtered program may depend on both predicate arity and code size. Leuschel and Sørensen [109] reported an average code size reduction of 20%. For the specialised PIC emulator this reduction will be shown to be more than 50%. Particularly the reduction of the arity of the predicates is significant.

Correct Erasures

The FAR algorithm computes a correct *erasure* for a given logic program P; an erasure is a set of predicate argument positions that can be eliminated without affecting the computed answers for any goal.

Definition 60 (Erasure). *Let P be a program. An erasure is a set of tuples (p, k) with $p/n \in \Pi_P$ and $1 \leq k \leq n$.*

Suppose an erasure contains $\{(p, 1), (p, 3), (p, 4)\}$. Then this means that the first, third and fourth arguments of predicate p are to be removed. Given a program P and erasure E, denote by $P|E$ the result of striking out the arguments in E from all occurrences of the predicates in P.

Example 31. *In the case of the erasure $\{(p, 1), (p, 3), (p, 4)\}$ an occurrence of an atomic formula for p such as $p(t_1, t_2, t_3, t_4, t_5)$, would be replaced by $p(t_2, t_5)$.* $\square$

An erasure E is *correct* for program P if the following property holds. For every computation of P with a goal G, the answers and finite failures are the same (for the non-erased arguments) as the computation of $P|E$ with $G|E$ (the result of applying the erasure to G).

Algorithm for Computing a Correct Erasure (FAR)

The algorithm presented by Leuschel and Sørensen successively identifies arguments that are "needed". Initially, the erasure for P is the set of all argument positions of predicates in P. On each iteration, arguments are removed from the erasure until a correct erasure is computed.

It is provable that if conditions 1-3 checked in the loop are false for all argument positions then the corresponding erasure is correct.

Algorithm 7 FAR Algorithm

Input: program P
Output: a correct erasure E for P
Initialise:
  $i = 0$;
  $E_0 =$ the set of all predicate arguments;
while there exists a $(p, k) \in E_i$
  and a clause $p(t_1, \ldots, t_n) \leftarrow B$ in P such that
    1. t_k is not a variable; or
    2. t_k is a variable occurring more
        than once in $p(t_1, \ldots, t_n)$; or
    3. t_k is a variable occurring in $B|E_i$
do $E_{i+1} = E_i \setminus \{(p, k)\}$; $i = i + 1$;
return E_i

Liveness Analysis using the FAR algorithm

We now show that the FAR algorithm can be used as a liveness analyser. In fact it is more general than a liveness analyser; as will be seen, it can remove redundancies other than dead variables. To do this we build a straightforward translation from control flow graphs and logic programs called control flow programs. Then we show that the classical liveness analysis on control flow graphs [1] is mimicked by the action of the FAR algorithm on control flow programs. Note that the flow programs and flow graphs considered in this section are not part of the PIC case study; they are just defined in order to show that the FAR algorithm can perform liveness analysis.

We take flow graphs to consist of a set of nodes (program points) and directed edges labelled either by an assignment statement $x = e$ or a boolean expression b. Conditional branches are represented by edges labelled with a boolean expression, and control flows along that edge if the condition evaluates to *true*. Control flow graphs with branch instructions and goto statements could be defined instead, without altering anything essential in the procedure below.

Let $x_1, \ldots, x_m$ be the set of all variables appearing in the graph (i.e. in assignment statements or in boolean expressions). For each node j define a unique predicate p_j with m arguments. The control flow program resulting from a given graph is defined to be the set of clauses of the following form.

1. $p_i(x_1, \ldots, x_m) \leftarrow x'_l = e, p_j(x_1, \ldots, x'_l, \ldots x_m)$, where $x_l = e$ is an assignment statement on the edge (i, j) and x'_l is a variable different from $x_1, \ldots, x_m$.

2. $p_i(x_1, \ldots, x_m) \leftarrow b, p_j(x_1, \ldots, x_m)$, where b is a boolean expression on the edge (i, j).

Let 0 be the entry node of the flow graph. We observe that there is a computation in the flow program that calls some sequence of predicates $p_0, p_{i_1}, p_{i_2}, p_{i_3}, \ldots$ if and only if there is a legal computation in the flow graph passing through nodes $0, i_1, i_2, i_3, \ldots$. The arguments of a call to predicate p_i represent the state of the variables in the program at node i in the corresponding flow graph execution.

Now consider the application of the FAR algorithm to this program. Initially, E_0 contains every pair (p_j, k) where j is a node and $1 \leq k \leq m$. (We assume that the arguments of the equality predicate and the boolean expression predicates that appear in the program are not erased.) We argue informally that the iterations of the FAR algorithm acting on the flow program correspond to the iterations of the classical liveness algorithm. Consider an edge (i, j) labelled by an assignment statement $x_l = e$. Define, as is usual, $use(i) = vars(e)$ and $def(i) = \{x_l\}$. If the edge is labelled by a boolean expression b then $use(i) = vars(b)$ and $def(i) = \{\}$.

Consider some erasure; let the set of arguments that are not erased from p_i be called $in(i)$. Let the set of arguments that are not erased from all predicates p_j such that there is an edge (i, j) be called $out(i)$. Then we can show that the FAR algorithm solves the classic data flow equations for liveness, namely $in(i) = use(i) \cup (out(i) - def(i))$, and $out(i) = \bigcup_{j \in Succ(i)} in(j)$, where $Succ(i)$ is the set of immediate successors (see Definition 55). The algorithm iterates starting from initial values $in(i) = out(i) = \{\}$ in the classic liveness analysis; this corresponds to the fact that the initial erasure consists of all predicate arguments. When the FAR algorithm terminates, the sets $in(i)$ contain the set of arguments that are *not* erased from predicate p_i, that is, the live variables at node i. A detailed proof could argue that the FAR loop terminates exactly when the above data flow equations are satisfied.

Claim 1. *Given a control flow graph containing variables $x_1, \ldots, x_m$ and the corresponding control flow program as defined above. Then according to the classical liveness analysis, a variable x_j is live at a given node i of the control flow graph $(x_j \in in(i))$ iff the FAR algorithm returns an erasure that does not include (p_i, j).*

Application of Redundant Argument Filtering

The specialised emulator described in Section 6.3 has a similar structure to the flow programs described above. The state of the registers is held in the arguments of the `execute` predicates, and there is one version of the `execute` predicate for each PIC program point. Application of the FAR algorithm to the specialised emulator results in the erasure of all register arguments from say `execute`$_i$ that are dead at the program point corresponding to `execute`$_i$. The number of registers live at

a point is often a small fraction of the total set of registers, and hence the FAR algorithm yields a program that can be much more efficient to analyse, without losing any essential information about the program.

In fact, redundant argument filtering does more than just remove dead arguments. It also allows some operations in the body of the `execute` clauses to be eliminated as they definitely succeed. Example 32 shows an example of the clause for the same predicate `execute__5` shown in Example 28 on page 133. Note that the number of arguments of `execute__5` has been drastically reduced and the clause body is simpler.

Example 32. *An example of an* `execute`*-clause after specialisation and redundant argument filtering is shown below.*

```
execute__5(B,A) :-
    C is B+A,
    0 is C>>8,
    C \== 0,
    D is 24 /\ 254,
    execute__6(D,B,C).
```

$\square$

Experiments with case study programs

The FAR algorithm has been applied to the specialised versions of the case study programs. Table 6.4 shows the reduction in the arity of the predicates and the time it took to run the FAR algorithm. The `execute`-predicates of a specialised emulator all have the same arity. During specialisation the list of registers is unfolded up to the last register used at some point in the program. The size of the machine state will vary from program to program but will remain static for a given program. The average live arity in the filtered program is calculated by summing the arity of the predicates in a given program and dividing it by the number of predicates in that program. The reduction is more than 50% for all test case programs.

The reduction in program size is due to both the reduction in arity and the removal of some calls in the clause bodies. Table 6.5 shows the program size before and after filtering for the test case programs. This reduction is also substantial.

154

Program	Arity	Avg. live arity	Reduction	Specialisation Time (sec.)
Compass	29	3.5	88%	2.3
Accelerometer	24	3.2	87%	1.5
GPS	46	18	61%	18.6

Table 6.4: Liveness transformation PIC programs

Program	Specialised emulator (kb)	Transformed emulator (kb)	Reduction
Compass	40	16	60%
Accelerometer	46	19	59%
GPS	183	94	49%

Table 6.5: Transformation reduction of PIC programs

6.5 Convex Polyhedron Analysis applied to specialised emulators

The program resulting from the specialisation of the emulator as described up until now is a logic program equivalent to the initially supplied PIC program, implementing the same (or a subset of its) semantics. Existing analysis tools and techniques for logic programs can now be applied to the specialised emulator, to reason about the PIC program. An example of such a tool is the abstract interpretation based pre-processor CiaoPP [86], a global program analysis, source to source transformation and optimisation tool for logic programs.

The specialised emulators contain only numerical data; exceptions to this rule would be if the emulator had been instrumented with e.g. read/write access patterns as described in Section 6.6.1. The obvious choice is therefore to apply a numerical analyser to the specialised emulators. In this section we apply the Convex Polyhedron Analyser, developed earlier, to the specialised emulator. These programs contain non-linear arithmetic operations, like boolean AND, OR and NOT, that must be given a linear approximation.

The Convex Polyhedron Analyser gives an approximation of the argument value relationships of the predicates in a program. In the specialised emulator each instruction in the PIC object program has an equivalent predicate in the residual logic program. Each element of the abstract machine state, including the data registers, has its own argument in each predicate. Approximating the arguments of the specialised emulator results in an approximation of the machine state with

155

respect to the object program.

The results from the analysis of the logic program must be presented in a manner that enables the PIC programmer to interpret these results with regard to the PIC program. The problem of relating results from the analysis of the (C)LP program to the "imperative" object program is simplified by ensuring that the specialised residual program is isomorphic to the object program.

The output of this analysis will be a set of linear constraints between the registers of the PIC microcontroller, for each program point in the object program. These constraints can aid the PIC programmer in discovering properties of the analysed programs, such as possible bugs due to overflows, redundant code due to static branch conditions etc.

6.5.1 Query-Answer transformation

The specialised emulator is a Prolog program where data flow is propagated in a top-down fashion, and similarly, the execution strategy of the program is in a top-down manner. Furthermore, many PIC programs are not intended to terminate and "succeed" - they simply run forever. The individual calls to the `execute`-predicates are not expected to result in an answer unless an error occurs. A bottom-up analyser, such as the Convex Polyhedron Analyser described in Chapter 4, provides sound information about the set of all possible *answers* obtained in a top-down evaluation of the program. Programs where predicate calls can have no answers, may have an empty model and bottom-up analyses may return no useful information.

Query-answer transformation of logic program, as also described in Section 2.3, provides a way to use a bottom-up analysis tool to return information about the computations themselves, in particular, on the set of calls to each predicate in the program [136]. The query-answer transformation is a generic logic program transformation and therefore also applies to our specialised emulators with no further modifications needed to the specialised emulators or the query-answer transformation tool.

Example 33. *We illustrate the query-answer transformation for the specialised emulator clauses. Take the clause shown in Example 32 on page 154. If we are interested in obtaining information about the calls to* `execute__6(D,B,C)` *the query-answer transformation would result in the following clause which is an "inverted"*

version of the original `execute` *clause.*

```
execute__6_query(D,B,C) :-
  execute__5_query(B,A),
  C is B+A,
  0 is C>>8,
  C \== 0,
  D is 24 /\ 254.
```

☐

All PIC programs start at program memory location 0. Partially evaluating the emulator produces an `execute__0` predicate for this first instruction in the program. The query-answer transformation will produce an `execute__0_query` predicate for this first instruction in the program. When provided with some initially called goal as a fact, and this would then typically be called `execute__0_query` for the first instruction in the program, the bottom-up analyser generates a model for each query predicate.

6.5.2 Connecting results from analyser to object program

The object program is transformed in several stages, where each stage may rename registers, transform predicate heads either unfolding lists or eliminating redundant arguments. The analysis output is specified with respect to the specialised emulator, but the output should be presented with respect to the object program. Each program transformation step must supply some information specifying what previous predicate head has been transformed to the current predicate head. The Logen partial evaluator supplies comments in the residual program stating the exact predicate including its arguments, that through unfolding resulted in a particular predicate in the residual program. The redundant arguments filtering was modified to supply similar comments in the resulting filtered program. The procedure to translate analysis results from the Convex Polyhedron Analyser (CPA) to the object program involves the following steps:

1. The CPA renames the arguments of the predicates. Each argument position of each query-predicate in the output is mapped to that same argument position of the `execute`-predicate in the FAR filtered program.

2. Each argument name of the FAR filtered predicates is mapped to its argument position in the residual program from the Logen specialiser.

3. The name of the arguments of the predicates in the residual program is mapped to its register number in the PIC emulator. Accumulator, PC and

other machine state elements can be identified by their argument position.

4. The register numbers are located in the assembled PIC program to identify what name that particular register has been given by the programmer. The constraints from the CPA can now be renamed to register numbers used in the PIC program.

5. The PC identified in the residual program can be used to find the exact instruction in the PIC program that the `execute`-predicate implements. The renamed constraints for that `execute`-predicate are presented next to the PIC instruction in the assembler program.

Example 34. *Figure 6.6 illustrates the process for the instruction previously used in Example 28. The results for the `execute__5_query`-predicate from the Convex Polyhedron Analyser is mapped to the `exeute__5`-predicate from the Reverse Redundant Agument Filtering which in turn is mapped to the `execute`-predicate in the PIC emulator. Argument `A` of the `query`-predicate is mapped to register number 16 which is located in the assembled PIC program and indentified to be named "SUM". Argument `B` is identified to be the accumulator. The instruction is identified to be number 4 in the object program and translates to "`addwf SUM,0`" with the constraints "`[1*SUM=64,1*Acc=64]`".* □

6.5.3 Integer grid polyhedra

Only integer values can occur in the clause heads of the specialised emulators, and all arithmetic operations in the clause bodies yields integer results for integer operands, e.g. division, that could result in a rational number, does not occur in any clause bodies. The n-dimensional polyhedra are therefore implicitly restricted to n-dimensional integer grids, e.g. the 2 dimensional polyhedron $\mathcal{P}$, shown in Figure 6.7 on page 160 is restricted to the intersection of that polyhedron with the 2-dimensional integer grid $\mathbb{Z}^2$, $\mathcal{P}_{\mathbb{Z}} = \mathcal{P} \cap \mathbb{Z}^2$. Projection of a 2-dimensional polyhedron over the rational (or real) numbers onto a single dimension results in an interval that is the most precise approximation. Projecting the rational polyhedron shown in Figure 6.7 onto the x-axis would result in the accurate approximation $X > 0, X \in \mathbb{Q}$. Projecting the same polyhedron again, only this time intersected with the integer grid, onto the x-axis would ideally result in the accurate approximation $2 * X + 1 \geq 0, X \in \mathbb{N}$. Using a polyhedral analyser for the rational domain to analyse programs over the integer domain may lose precision during projection. For the example polyhedron shown here, projection may result in the interval $X > 0$ implicitly with $X \in \mathbb{N}$ which may be a safe approximation, but not a precise approximation, of the odd numbers.

Output:
```
addwf SUM,0      [1*SUM=64,1*Acc=64]
```

ASM:
```
00000010 00010 SUM EQU 10H
...
        0004 0710 00023 addwf SUM,0
```

PE: (Dec to Hex)
```
/* execute([...],[...,14-P,15-Q,16-R,17-S,18-T],4,A) :-
        execute__5(T,R,P,N,L,J,H,F,D,B,C,E,G,I,K,M,O,Q,S,A). */
```
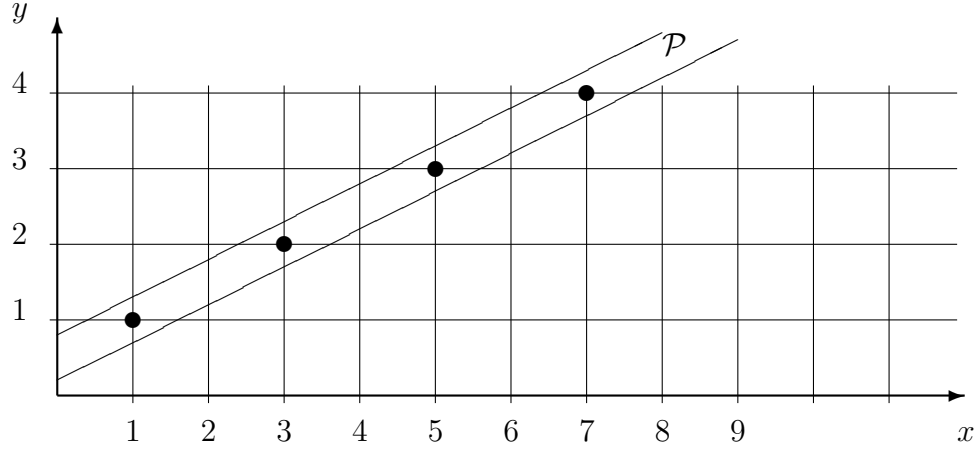
FAR:
```
/* execute__5(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T):-execute__5(B,T) */
```

CPA:
```
execute__5_query(A,B) :- [1*A=64,1*B=64]
```

Figure 6.6: Translate results from analysis of specialised emulator to results for source object program. Translation is a reversal of the specialisation process, so the figure is read "bottom-up".

project $\mathcal{P} \cap \mathbb{Z}^2$ onto x
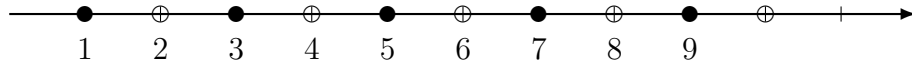
Figure 6.7: Projecting the 2-dimensional integer grid polyhedron $\mathcal{P}$ onto the x-axis is approximated by the set of odd numbers.

When our analyser (over the rational domain) is applied to the specialised emulators, no explicit conversion to the integer domain takes place. This may, as shown above, result in a loss of precision.

The specialised emulators contains a few boolean operators (over integers) for which some linear approximation must be given. These are described next.

6.5.4 Linear approximation

The example code shown in Example 33 contains two non-linear arithmetic operations, the bit shift '>>' and the boolean AND '/\'. Approximation for these two operators plus other boolean operators that the emulator may use such as OR and

NOT (bit-wise negation) must be added to the Convex Polyhedron Analyser. The linearisation procedure for the analyser, described in Section 4.3.4, is extended to include approximation of boolean operators. These operators are not typically found in CLP programs.

Bit-wise negation

The approximation of the unary operator NOT depends on the maximum integer size of the abstract machine we are modelling. In our case study, the 8 bit PIC micro-controller the maximum integer value is 255. This bit-wise negation operator can be given an exact linear approximation. The Prolog term X is \Y, where '\' is the Prolog syntax for bit-wise negation, can be approximated by the expression $X = 255 - Y$.

Boolean AND

For the AND operator it is not possible to give an exact numerical approximation if either of the operands is not a constant. The results however can never be greater than smaller one of the two operands, and never less than zero, assuming the abstract machine only allows for unsigned integers. The Prolog term X is Y /\ Z can be approximated by the constraints $X \leq Y \wedge X \leq Z \wedge X \geq 0$.

Boolean OR

Similarly with the OR operator. If either of the operands is not a a constant, an exact approximation cannot be given. The result of an OR operation however can never be greater than the sum of the operands and never smaller than the largest operand. The Prolog term X is Y \/ Z can be approximated by the constraints $X \leq Y + Z \wedge X \geq Y \wedge X \geq Z$.

Bit shift

The PIC processor has a bit shift instruction for left shifting of bits in a data register, and an operation for right shifting of bits. Bits can only be shifted one position at a time. Left-shifting bits one position equals multiplication by 2 - this case has already covered under the arithmetic operations. Right-shifting is division by 2 and rounding down to nearest integer.

These approximations are not very precise. As shown in Example 35 the constraints on registers whose values are the result of a series of boolean operations, are correct but not precise. The resulting set of constraints from the analyser is

renamed and shown along the PIC object program following the procedure described previously in Section 6.5.2. In this particular example no special features of the analyser, such as delayed widening or widening up-to, has been enabled. This is also evident from the lack of a lower bound for the CNT or SUM register. The following sections will demonstrate how narrowing, widening up-to and delayed widening for the Convex Polyhedral domain can be used to recover bounds and improve precision of the analysis results for the PIC programs.

Example 35. *This example is a small PIC program containing a loop of 10 iterations in which a register is incremented by two in each iteration. This is implemented with an* OR *and an* ADD *instruction. The instruction* decfsz *decrements* CNT *and skips the following instruction if the result is* 0.

Running it through the analyser produces the constraints shown below. The constraints are shown for the live machine state (data registers and accumulator) prior to the execution of the listed instruction. Acc *is the accumulator - its value is 1 from instruction 5 and to the end of the program. The precise value of* SUM *after the loop terminates, is 20. The analysis shows it to be less than or equal to 20; a correct result but not an exact result. The loss of precision is a result of the inexact approximation of the* OR*-operator.*

```
 1:    movlw D'10'
 2:    movwf CNT    [1*Acc=10]
 3:    movlw D'0'   [1*CNT=10]
 4:    movwf SUM    [1*CNT=10,1*Acc=0]
 5:    movlw D'1'   [1*SUM=0,1*CNT=10]
 6:  loop1
 7:    iorwf SUM,1  [-1*CNT>= -10,-1*SUM+ -2*CNT>= -20,1*Acc=1]
 8:    addwf SUM,1  [-1*CNT>= -10,-1*SUM+ -2*CNT>= -21,1*Acc=1]
 9:    decfsz CNT   [-1*CNT>= -10,-1*SUM+ -2*CNT>= -22,1*Acc=1]
10:     goto loop1  [-1*CNT>= -9,-1*SUM+ -2*CNT>= -20,1*Acc=1]
11:     goto MAIN   [-1*SUM>= -20]
```

Replacing the iorwf*-instruction with an* addwf*-instruction that has the same effect as the* iorwf*-instruction for this particular program, and has an exact linear approximation, results in the following constraints. The result after the loop terminates is now an exact result.*

162

```
1:    movlw D'10'
2:    movwf CNT     [1*Acc=10]
3:    movlw D'0'    [1*CNT=10]
4:    movwf SUM     [1*CNT=10,1*Acc=0]
5:    movlw D'1'    [1*SUM=0,1*CNT=10]
6:  loop1
7:    addwf SUM,1   [1*SUM>=0,1*SUM+2*CNT=20,1*Acc=1]
8:    addwf SUM,1   [1*SUM>=1,1*SUM+2*CNT=21,1*Acc=1]
9:    decfsz CNT    [1*SUM>=2,1*SUM+2*CNT=22,1*Acc=1]
10:     goto loop1  [1*SUM>=2,1*SUM+2*CNT=20,1*Acc=1]
11:     goto MAIN   [1*SUM=20]
```

□

6.5.5   Widening

In the numerical domain arithmetic operations have no upper or lower bounds. For infinite chains of operations some mechanism must be implemented to ensure or accelerate the convergence of the fix point computations. This is what the polyhedral widening operator ensures. Various widening operators are provided by the Convex Polyhedron Analyser. In our specialised emulators loops can occur - even non terminating loops. Widening will ensure termination of the analysis, but at the cost of precision. Typically this will be evident in a lack of either upper or lower bounds.

Example 36. *Take for example the small PIC program shown below; a simple loop of 10 iterations, in which 10 is added to a register in each iteration of the loop (when the loop terminates the register contains the value 100).*

```
1:    movlw D'0'
2:    movwf SUM     [1*Acc=0]
3:    movlw D'10'   [1*SUM=0]
4:    movwf CNT     [1*SUM=0,1*Acc=10]
5:  loop1
6:    addwf SUM,1   [1*SUM>=0,1*SUM+10*CNT=100,1*Acc=10]
7:    decfsz CNT    [1*SUM>=10,1*SUM+10*CNT=110,1*Acc=10]
8:    goto loop1    [1*SUM>=10,1*SUM+10*CNT=100,1*Acc=10]
9:    goto MAIN
```

The register containing the loop counter (CNT), if isolated in the constraints, lacks the lower bound '1' inside the loop. Apart from that, the register containing the sum of 10s, has the right relation with the loop counter. Isolating SUM in the

constraints for instruction 7 yields the following equation: $SUM = 110 - 10 * CNT$, for $CNT \in [1, \ldots, 10] \Rightarrow SUM \in [10, 20, 30, \ldots, 100]$ or the exact values SUM will be assigned in the loop.

Analysing the same program with either of the following widening techniques enabled - widening up-to, delayed widening for at least 10 iterations or simple narrowing - gives the correct lower bound.

```
1:     movlw D'0'
2:     movwf SUM     [1*Acc=0]
3:     movlw D'10'   [1*SUM=0]
4:     movwf CNT      [1*SUM=0,1*Acc=10]
5:  loop1
6:     addwf SUM,1   [1*SUM>=0,-1*SUM> -100,1*SUM+10*CNT=100,1*Acc=10]
7:     decfsz CNT    [-1*SUM> -110,1*SUM>=10,1*SUM+10*CNT=110,1*Acc=10]
8:     goto loop1    [-1*SUM> -100,1*SUM>=10,1*SUM+10*CNT=100,1*Acc=10]
9:     goto MAIN
```
□

For this particular example PIC-program any of the three improved widening techniques will suffice. This is not true in general. The next section will compare the techniques for the specialised emulators.

6.5.6   Convex Polyhedron Analysis of PIC programs

The structure of the specialised emulators resembles that of an *imperative* program. Generally speaking an imperative program is characterised by sequences of computations, where each step in a sequence changes the program state. In the specialised emulator a step in a computation sequence is equivalent to a predicate. We will consider the specialised emulators to be imperative style constraint logic program. The limited size of the stack on the PIC microcontroller limits the possibilities for the PIC programmer to write recursive programs. Loops in the PIC object program would be translated to loops in the imperative style CLP program.

The Convex Polyhedron Analyser has already been applied to a handful of declarative style CLP programs in Chapter 4. The imperative style CLP programs justifies a new evaluation of the analyser for these particular programs. The programs that were used to evaluate the analyser in Chapter 4 were small programs that resulted in a set of constraints that were small enough to present in this thesis. As Table 6.5 on page 155 shows, the specialised emulators are significantly larger programs, from 16kb and up, whose resulting set of constraints from the analysis will be too large to present here.

As a rule of thumb a higher number of constraints represents a more precise approximation for a predicate. Below are the results for a single predicate of the *compass*-program. With standard widening the approximation is

$$\texttt{execute\_\_100\_query}(\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}) :\text{-} [1 * \texttt{C} >= 0]$$

Analysing it with delayed widening produces the following result for the same predicate

$$\texttt{execute\_\_100\_query}(\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}) :\text{-} [-1 * \texttt{C} >= -255, 1 * \texttt{C} >= 0, -1 * \texttt{A} >= -1]$$

In this case the larger set of constraints also results in the smaller, and therefore better, approximating polyhedron. If the two sets of constraints were e.g. $[A < 100]$ and $[A < 1]$ then of course the latter set of constraints results in the smaller polyhedron, though the sets have the same size. For easy comparison of the analysis results for larger programs, we will simply use the size of the sets of constraints to find the better approximation.

Delayed Widening

One of the methods for improving precision of the CPA is the delayed widening technique, where the application of the widening operator is postponed for a number of iterations of the fix point computations. The particular structure of the specialised emulators results in only those query-predicates, whose predecessor in the call graph has been assigned an approximating polyhedron, can themselves be assigned an approximating polyhedron. The minimum number of iterations of the fix point computations to complete before all query-predicates have been assigned an approximating polyhedron, in a worst case situation, is equal to to the height of the minimal spanning tree of the call graph starting at the entry call for the program. If the height of this tree is greater than the number of iterations to delay, widening at some program points may not benefit from the delayed widening. Table 6.6 on the next page shows the results of analysing the compass case study program, with different number of iterations to delay widening with. The third column shows how many iterations the analyser requires to reach a fix point. The last column shows the number of constraints found for the program - a higher number generally means more precise approximation. Delaying for any number shorter than the length of the program - 141 instructions in this particular program - results in little gain in precision of the approximation.

For large programs this may mean a high number of iterations to delay widening. One alternative to the delayed widening strategy has been explored. Instead of starting widening after a certain number of iterations, widening is only applied every second, third etc. iteration. The results of this strategy is shown in Table 6.7.

Program	Delayed Widening (iterations)	Number of Iterations to fix point	Resulting Constraints (Number of)
Compass	0	158	200
Compass	5	158	200
Compass	10	158	200
Compass	30	158	200
Compass	100	178	223
Compass	140	208	289
Compass	150	212	350
Compass	160	212	350

Table 6.6: Delayed Widening for PIC programs

This technique does not result in more precise approximations or fewer fix point iterations overall compared to delaying widening for a large number of iterations. And delaying for more iterations does not necessarily result in more precision than delaying for fewer iterations.

A final solution would be to let the delaying of the widening apply individually to each widening point. A widening point is not considered until after a certain number of fix point iterations. This number is equal to the length of some path in the call-graph, from entry to the particular widening point. If widening should be delayed for d iterations, and p is a widening point and the shortest path from entry to p is of length l, then widening only applies to p after $l + d$ iterations. Table 6.8 shows the results of following this procedure. After 9 iterations there are no improvements to be gained from delaying widening. This happens to be the result for all three test case programs. These three PIC programs all contain loops of exactly 8 iterations. These loops typically occur when the programs reads input one bit at a time. This suggests that the number of iterations that widening should be delayed, should be determined by the sizes of the static loops in a given program.

Widen up-to and narrowing

Table 6.9 shows the results of applying the CPA to the specialised case study PIC programs. For the compass program the best result is achieved using a combination delayed widening, widen up-to and narrowing. The most significant gain appears to come from delayed widening and widen up-to. Narrowing contributes only few extra constraints. For the accelerometer program only widen up-to shows a small gain in precision.

Program	Iterations between widening	Number of Iterations to fix point	Resulting Constraints (Number of)
Compass	0	158	200
Compass	1	158	220
Compass	2	203	261
Compass	3	178	316
Compass	4	170	268
Compass	5	204	310
Compass	6	207	246
Compass	7	208	306
Compass	8	215	281
Compass	9	207	268
Compass	10	186	301

Table 6.7: Delayed Widening for PIC programs

Program	Delayed Widening (iterations)	Number of Iterations to fix point	Resulting Constraints (Number of)
Compass	7	208	227
Compass	8	208	266
Compass	9	215	354
Compass	10	215	354
Compass	20	215	354

Table 6.8: Delaying individually per widening point

Program	Delayed Widening (iterations)	Simple Narrowing (iterations)	Widen up-to	Resulting Constraints (Number of)
Compass				200
Compass		200		209
Compass			√	302
Compass	10			354
Compass	10	200		357
Compass	10		√	362
Compass	10	200	√	363
Accel.				420
Accel.		300		420
Accel.	10			420
Accel.			√	426
Accel.	10	300	√	426
GPS				6319

Table 6.9: Widening for PIC programs

Analysis of the GPS program is only possible without using improved widening techniques. This analysis takes just over 1 minute to complete. Analysing with delayed widening or widen up-to does not complete within 15 minutes. Narrowing results in the Ciao Prolog process allocating more than 1 GB memory. Asserting facts allocates memory but it does not appear to free this memory when the facts are retracted. This GPS program has a much larger live state than the two others, as Table 6.4 also shows. The approximating polyhedra are of similarly large dimensions. For this particular program "aggressive" widening is needed to accelerate the fix point computations at the cost of precision.

Selecting Widening Points

Section 4.3.5 described how the set of widening points for a program could be calculated. An algorithm selecting widening points based the number of loops that the individual program points cuts, was suggested. The results reported so far in this chapter on delayed widening, narrowing and widen up-to were all using the cut-loop algorithm. This section compares the results of this algorithm to the feedback-edge algorithm suggested by both Bourdoncle and Cousot [21, 46]. Table 6.10 shows the compass program analysed using both the feedback-edge detection of widening points, that finds 22 program points to widen at, and the cut-loop detection that finds 12 widening points. Results are showed using

Delay Widen	Narrow	Widen up-to	Feedback-edge 22 ∇-points	Cut-Loop 12 ∇-points
			200	200
	200		209	209
		$\checkmark$	295	302
10			340	354
10		$\checkmark$	355	357
10	200		357	362
10	200	$\checkmark$	363	363

Table 6.10: Feedback-edge results compared to Cut-Loop results

different configurations of the widening operator. Using fewer widening points improves precision for some combinations of widening strategies. However for this particular program the number of widening points selected makes no difference for the resulting precision if either "least" or "most" optimal widening strategy is used.

6.6   Instrumented emulator

The emulator can be "instrumented" with additional information not present in the PIC processor. This additional information can be used for program analysis purposes.

6.6.1   Register Access Patterns

The register list (see Section 6.2.3) could be extended with additional information that keeps a history of previous values and operations performed on the register. Each element in the register list could have the following structure, `RegNr-(Vlist,RWlist)`, where `Vlist` is a list of previous values the register has been assigned, with the current value in the head of the list. The `RWlist` is a list of `r(PCw,PCr)`-terms and `w(PCw)`-terms signifying that the register has been read or written. For each instance where the register has been accessed the program counter (PC) is stored in the terms and denotes at which program point a register had been read, and at what program point the read value had been written. For example a write operation at program point 22 would add the term `w(22)` to the head of the `RWlist` , and read operation to the same register at the next program point would add the term `r(22,23)` to the head of the `RWlist`. This procedure can for instance be used to determine access patterns of the data registers. A

169

register would have an illegal access pattern if it was read before it was initialised thereby propagating undefined values in the program, possibly indicating a bug. These access patterns can be defined using regular types rules, for instance a read-before-write type could be defined by the rules

$$
\begin{aligned}
rbw &\leftarrow [r]; [rw|rbw] \\
rw &\leftarrow r; w
\end{aligned}
$$

Keeping a list of values stored in the register can be used to detect for instance constant registers. If the same value is written to a register repeatedly is may reveal redundant computations in the program. This approach to analysing the PIC microcontroller is described in [85].

6.6.2 Worst Case Execution Time

The results from the convex polyhedron analyser applied to the specialised emulators provided some information about the PIC programs. There are other program analyses that depends on this information, specifically accurate approximations of upper and lower bounds on values in a program. An example of such an analysis is a Worst Case Execution Time (WCET) analysis. This section will briefly illustrate how the PIC analyser can be extended to include execution time analysis for a PIC program.

The PIC microcontroller has a variable clock frequency. The slower the clock frequence is, the lower the power consumption will be. For a wearable application power consumption is important. Wearable software would generally need to communicate with other devices. And device to device communication typically requires some precise timing constraints to be met. The WCET analysis can aid the programmer in determining whether a program meets these timing constraints.

A WCET analysis will give an estimate of how much time a given computational task will take to complete on a given hardware platform. The estimate is a safe approximation and is typically stated either in (fractions of) seconds or number of clock cycles. The clock frequency of the hardware platform will then determine the actual time to complete the task. The results of the WCET analysis can then be used for e.g. verification purposes for real time systems where tasks must be guaranteed to complete within some time frame. If the software fails to meet these timing constraints the results may have grave consequences such as loss of life.

WCET is a static analysis where abstract interpretation has been applied successfully [145]. A WCET analyser would typically contain the following components [65]

- A Control Flow Analysis of the object program

- A Value Analysis computing intervals of data memory, registers etc.

- A Loop Bound Analysis determining upper and lower bounds on the number of iterations a given loop can be executed

Hardware specific analyses may be required, such as cache analysis, depending on the hardware platform being modelled.

Parametric WCET

A *parametric* WCET allows parameters in the expression approximating the WCET. The execution time of a program may depend on (dynamic) values that will not be known prior to running the program. The WCET can be parameterised by these unknown values. Automatically discovering these relations between the parameters and the execution time, can provide insight into how these parameters affect the execution-time behaviour of the program.

Our analyser already provides the elements required for a WCET analysis. The CFG is embedded in the specialised emulator. The Convex Polyhedral Analyser computes the Value Analysis and implicitly also the loop bounds. In other words no additional analyses are required in our case. Only the execution time of each program point is missing to complete the WCET analysis.

Instrumenting the emulator with timing information

A loop counter is added to the emulator shown previously in Section 6.2.3. The counter will be increased by the number of clock cycles that the current PIC instruction takes to complete. The new emulator loop is shown below.

```
execute(Prog,StateIn,Clock,Environment) :-
    fetchinst(Prog,PC,I,R1,R2),
    execInst(I,R1,R2,StateIn,StateT,ClockTicks),
    simulatehw(StateT,StateOut,Environment,NewEnvironment),
    NewClock is Clock + ClockTicks,
    execute(Prog,StateOut,NewClock,NewEnvironment).
```

An additional argument is required for each emulated instruction. Example 26 on page 127 showed the emulation of the `goto`-instruction. With the added clock parameter it now looks like this:

```
execInst(goto,Arg1,_, state(R,S,PC,Acc),state(R,S,Arg1,Acc),1).
```

This procedure is similar to that outlined by Cousot and Cousot in Example 15 in [54]:

> "Observe that the analysis discovers relations between variables that never appear within the same command. Incidentally, this fact can be used to prove automatically the termination of loops: a new counter is added to the program for each loop which is initialized to zero and incremented by one within the loop body. The analysis will relate its value to that of the other variables of the program. If the value of the counter is bounded on loop exit, then termination is automatically proved."

If the worst case execution time is bounded then similarly termination of the program is proved. We count not the number of loop iterations but the number of clock cycles executed for every instruction in the program.

Examples where the convex polyhedron analyser is applied to the instrumented emulator is shown next.

A loop can have a "static" or "dynamic" number of iterations and similarly the length (in instructions) of a loop iteration can be static or dynamic. Two examples will show that a static number of iterations and loop length yields exact execution time approximation and a dynamic number of iterations and dynamic loop size results in a safe approximation that is still precise.

Example 37. *A PIC program containing a simple loop was shown in Example 36. Re-analysing the program with global clock enabled and none of the improved widening methods enabled, produces the following constraints (for brevity only constraints on the clock are reported).*

```
1:    movlw D'0'    [1*Clk=1]
2:    movwf SUM     [1*Clk=2]
3:    movlw D'10'   1*Clk=3]
4:    movwf CNT     [1*Clk=4]
5: loop1
6:    addwf SUM,1   [1*Clk>=5,4*CNT+1*Clk=45,2*SUM+ -5*Clk= -25]
7:    decfsz CNT    [2*SUM+ -5*Clk= -10]
8:    goto loop1    [2*SUM+ -5*Clk= -15]
9:    nop           [1*Clk=44]
```

Each loop takes 4 clock cycles, there are 10 iterations and 4 clock cycles are spent initialising the registers. The correct result '44' is found. The execution time of this example program does not depend any input values, hence no parameterised

expression of the execution time is given. The next example will show a program where WCET can be stated as a parameterised expression.

Example 38. *A modified version of the program shown in Example 36 is listed below. The number of loops to execute is now read from the input port. With external input considered dynamic, the result can be any integer in the interval* $[0, \ldots, 255]$. *The value read is restricted to the interval* $[1, \ldots, 16]$ *by applying boolean* AND *with the value '15' and adding '1'.*

In this example a conditioned call to a subroutine is added to the loop. The subroutine will be executed depending on whether bit 3 is set in the register 'SUM' - an otherwise useless condition. Analysing this program gives the following constraints on the clock:

```
1:      movlw D'0'       [1*Clk=1]
2:      movwf SUM        [1*Clk=2]
3:      movf INPUT,0     [1*Clk=3]
4:      andlw D'15'      [1*Clk=4]
5:      addlw D'1'       [1*Clk=5]
6:      movwf CNT        [1*Clk=6]
7:      movwf LOOPSIZE   [1*Clk=7]
8:      movlw D'10'      [1*Clk=8]
9:   loop1
10:     addwf SUM,1      [-6*LOOPSZ+6*CNT+1*Clk>=9,
                              10*LOOPSZ+ -10*CNT+ -1*Clk>= -9]
11:     btfsc SUM,3      [-6*LOOPSZ+6*CNT+1*Clk>=10,
                              10*LOOPSZ+ -10*CNT+ -1*Clk>= -10]
12:     call oddloop     [-6*LOOPSZ+6*CNT+1*Clk>=11,
                              10*LOOPSZ+ -10*CNT+ -1*Clk>= -11]
13:     decfsz CNT       [-6*LOOPSZ+6*CNT+1*Clk>=12,
                              10*LOOPSZ+ -6*CNT+ -1*Clk>= -16,
                              10*LOOPSZ+ -10*CNT+ -1*Clk>= -16]
14:     goto loop1       [1*SUM>=10,1*SUM+ -1*Clk>= -7,
                              -3*SUM+5*Clk>=35]
15:     goto MAIN        [-6*LOOPSZ+1*Clk>=8,10*LOOPSZ+ -1*Clk>= -8]
16:     oddloop
17:     nop              [-6*LOOPSZ+6*CNT+1*Clk>=13,
                              10*LOOPSZ+ -10*CNT+ -1*Clk>= -13]
18:     return           [-6*LOOPSZ+6*CNT+1*Clk>=14,
                              10*LOOPSZ+ -10*CNT+ -1*Clk>= -14]
19:  MAIN
```

If the longest loop path is taken then each loop iteration takes 10 clock cycles to complete. There can be up to 16 iterations of the loop. Finally there are 10 clock

cycles spent initialising and exiting the loop. The constraints found on the clock after loop exit is

```
[-1*Clk>= -170,1*Clk>=16]
```

The upper bound approximation of the clock, $16*10+10 = 170$, is both safe and precise. The shortest path through the program is one loop iteration (of 6 clock cycles) where the subroutine is not called, taking 16 clock cycles to complete including setting up the registers initially.

Projecting the constraints onto `Clk` *and* `LOOPSIZE` *gives the following parameterised expression for the execution time*

```
[-6*LOOPSZ+1*Clk>=10,10*LOOPSZ+ -1*Clk>= -10,1*LOOPSZ>=1,-1*LOOPSZ>= -16]
```

Here the sizes of the shortest and longest path through the loop are indirectly stated in the constraints. The number of clock cycles used to set up the loop can also be extracted from these constraints.

For the PIC test case programs the timing critical part is the subroutine sending (or reading) a byte from the I/O port attached to a serial interface. The subroutine transmitting a byte, one bit at a time, is shown in Figure 6.8. The value NUMBIT determines the number of bits to transmit (a constant of 8). An additional stop bit will be appended to the bit stream. Analysing this subroutine results in the constraints $[-1*Clk > -124, 1*Clk >= 16]$ for the global clock. Each iteration of the loop will take on average $(124-5)/9 = 13,2$ clock cycles - worst case is 9 iterations and 5 clock cycles are spent setting up the registers initially. Assuming the expected bit rate on the output port is 4800 bps then we can calculate the slowest clock frequency that will meet the timing constraints. At $4800bps$ the delay between bits must be $4800^{-1} = 208\mu s$. Each clock cycle should then take $\frac{208}{13,2} = 15,75\mu s$. This translates to a clock frequency of $15,75\mu s^{-1} = 63,5kHz$. In other words, any clock frequency slower than this cannot transmit at $4800bps$.

If the clock frequency is known then the delay between loop iteration required to meet the desired bitrate can be calculated. At $1MHz$ a clock cycle is exactly $1\mu s$. At $4800bps$, each loop iteration should take $208\mu s$. Executing a loop iteration at this frequency takes about $13\mu s$ leaving an artificial delay of $195\mu s$ to be introduced in each iteration of the loop to meet the timing requirements for the desired bit rate.

6.7   PIC Analysis Tool

The PIC analysis tool have been made available online. This illustrates that the whole procedure can be fully automated with no other involvement from the user

```
XMIT
   MOVWF SER_TX ; Data shifter
   MOVLW NUMBIT+1 ; Get the bit count + 1 for start bit
   MOVWF BITCNT ; Preset data bit (down) counter

; Send the start bit

   TXLOW ; Set start bit level
   GOTO XMITC ; Wait for start element to go

; Set the transmit data level from the carry and wait for an element

XMITA
   RRF SER_TX,1 ; Clock shift register RIGHT through carry
   SKPNC ; If data (carry) is '0', skip
   GOTO XMITB
   TXLOW ; Data is '0'
   GOTO XMITC
XMITB
   TXHI ; Data is '1'
XMITC
; WAITEM ; Wait for the element to go

; Count the elements as they are sent

   TSTF BITCNT ; Zero if just sent the stop bit
   SKPNZ ; Skip next if bit count is not zero
   RETURN ; Exit from XMIT

   DECFSZ BITCNT,1 ; Dec. bit count, skip if zero
   GOTO XMITA ; Loop until all bits are sent

; Bit count has zeroed, send the stop bit

   TXHI ; Set stop bit
   GOTO XMITC ; Wait for the stop bit to go
```

Figure 6.8: PIC code for serial transmission of single byte

PIC Analyser

1. Upload Program	2. Analysis Stage	3. Statistics	4. Highlighted program

Displaying analysis

```
    LIST P=16F84A
    __config 3FFAh
    STATUS equ 3H
    INPUT equ 4H
    SUM equ 10H
    CNT equ SUM + 1
    ORG 00
0:  nop
1:  movlw D'0'      1*INPUT=0,1*Clk=1
2:  movwf SUM       1*Acc=0,1*INPUT=0,1*Clk=2
3:  movf INPUT,0    1*INPUT=0,1*SUM=0,1*Clk=3
4:  andlw D'15'     1*Acc=0,1*SUM=0,1*Clk=4
5:  addlw D'1'      -1*Acc>= -15,1*SUM=0,1*Clk=5
6:  movwf CNT       -1*Acc>= -16,1*SUM=0,1*Clk=6
7:  movlw D'10'     -1*CNT>= -16,1*SUM=0,1*Clk=7
    loop1
8:  addwf SUM,1     -1*Clk> -72,1*Clk>=8,-4*CNT+ -1*Clk>= -72,2*SUM+ -5*Clk= -40,1*Acc=10
9:  decfsz CNT      -1*Clk> -73,1*Clk>=9,-4*CNT+ -1*Clk>= -73,2*SUM+ -5*Clk= -25,1*Acc=10
10: goto loop1      1*Clk>=10,-4*CNT+ -1*Clk>= -70,1*CNT>0,2*SUM+ -5*Clk= -30,1*Acc=10

11: nop             -1*Clk>= -71,1*Clk>=11
```

Figure 6.9: Example output from analysis of a PIC program

than simply supplying a program for analysis. An example output of the analysis is shown in Figure 6.9. The diagram in Figure 6.10 on the facing page shows:

- The data that must be supplied by the user (only the PIC program).

- The elements of the analyser that must be written specifically for our particular case study PIC microcontroller.

- The set of tools used to transform and analyse the supplied PIC program.

- A web-based front-end making the analysis tool available online.
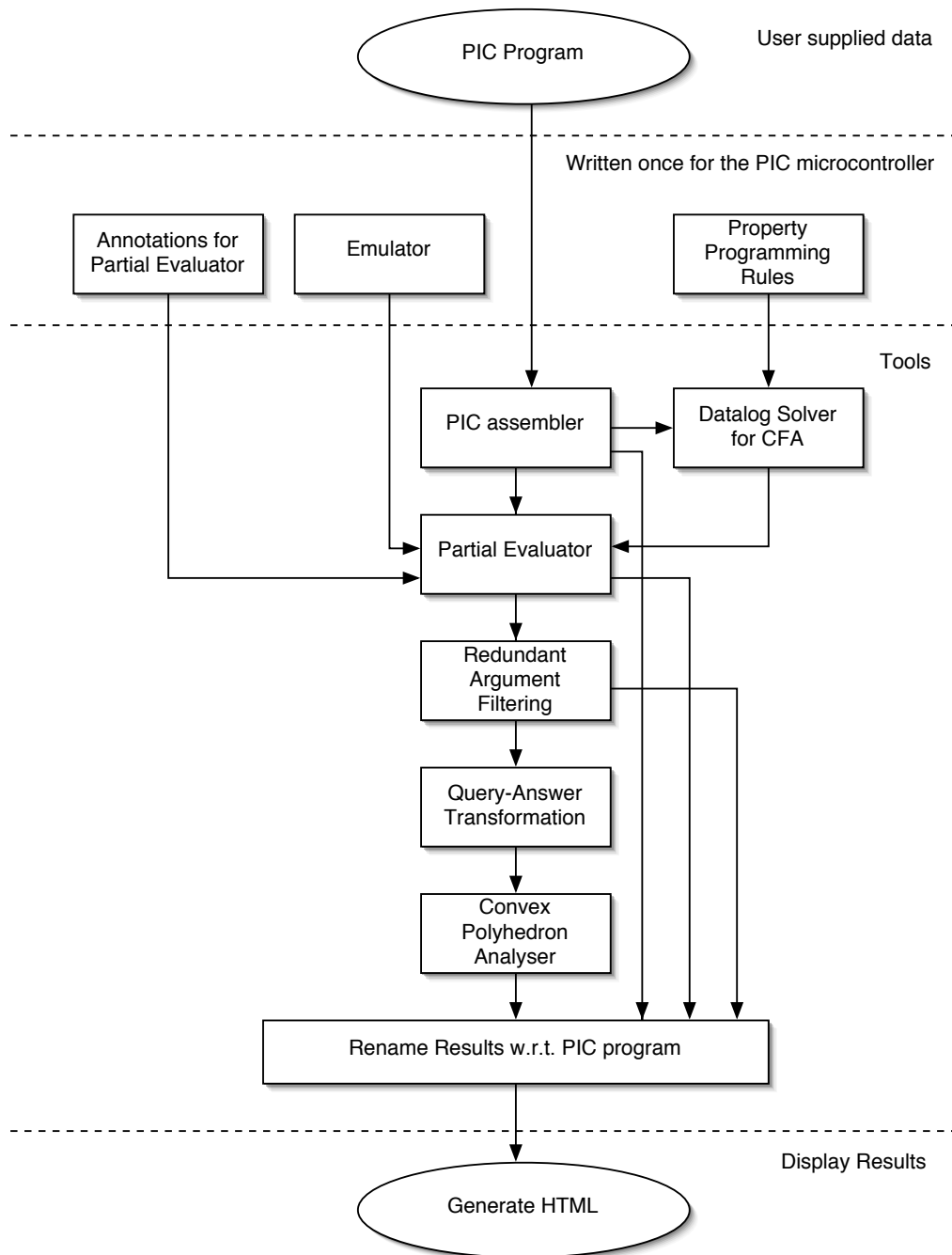
To try out the analysis too, visit the URL

```
http://wagner.ruc.dk/PIC/ConvexHull/
```

Figure 6.10: Structure of web-based front-end to PIC analyser

177

Chapter 7

Related work

The main topic that was covered in this thesis was the application of abstract interpretation based analysis tools to embedded systems and software such as that used in wearable computing. In this chapter we review related work. The analysis tools developed in this thesis were developed independently of the overall goal. We therefore also review related work for these tools.

Abstract Interpretation of embedded software

There are related projects applying abstract interpretation based frameworks to embedded software to aid programmers in producing reliable and efficient software. The Hoist [130] project explores two abstract domains; the interval domain which is a less powerful domain than the polyhedral domain but with computationally better performance, and a bit wise domain that can potentially give more precise approximations than the polyhedral domain for code containing boolean arithmetic. In Hoist the analyser is applied to an emulator of the target machine. Hoist can use an existing emulator to automatically construct abstract transfer functions. It derives the transfer functions by brute force, i.e. trying all possible input combinations for a given instruction, hence at the moment it is limited to 8 bit processors. Deriving a single transfer function can take a day's time. The derived transfer functions can then be used in an abstract interpretation based analyser. Relying on existing emulators would make it difficult to instrument the emulator to obtain WCET analysis results, just to give an example. It does not capture numerical relations between data registers which would be required to obtain parametric WCET results.

The aiT [65] tool is specialised for timing validation of embedded software. It applies abstract interpretation based analysers over the interval domain to approximate register values and loop bounds. It is not a fully automatic WCET analyser since some user input may be required, e.g. manually specifying safe approximation

of loop bounds. The tool can handle more advanced features of modern processors, such as caches and pipelines, to give more precise timing analysis results. The tool has successfully been applied to real-time critical embedded software such as the flight control systems of the Airbus A380[1].

Program analysis using logic based meta-programs

Using logic programming to analyse other programming languages has been studied before [87]. In [59] program analyses formulated declaratively were used for groundness analysis of logic programs and strictness analysis of functional programs. Semantic properties of the target language were expressed as logic programs. XSB, a table-based logic programming system was used to compute the models of these logic programs.

Closely related to property programming is the work by Whaley and Lam [152]. They analysed Java byte code where properties of objects defined in the Java code were stated as Datalog rules. BDDs were used to provide an efficient means of evaluating the Datalog programs. Data Flow Analysis of imperative programs using logic based approaches has previously been investigated in [131, 137].

Recently .NET intermediate code (similar to Java's byte code) have been analysed and transformed using an extension of logic programming called path logic programming [64].

Saha and Ramakrishnan [138] formulated incremental and demand driven program analyses (e.g. pointer analysis) as logic based rules. This was used for analysis of C-source code.

Polyhedral analysis

Analysis of argument size relationships of constraint logic programs has previously been investigated. Argument size analysers are typically applied in connection with termination analysis [14, 118, 119, 150]. Polyhedral analysers have been applied by Benoy and King [12], and Bagnara [4] applied a restricted form of polyhedra limited to rectangular bounding boxes corresponding to the interval domain.

Improved widening operators and widening strategies is an active research area that has been explored in e.g. [5, 142, 78].

Polyhedral analysis have previously been used for finding parametric WCET results [111, 79]. Authors claims this work is the first fully automatic parametric WCET analyser. The method applies abstract interpretation over the polyhedral domain to automatically detect bound on loop iterations. Standard polyhedral widening is used to accelerate fix point computations.

---

[1]`http://www.absint.com/releases/050427.htm`

Type analysis and tree automata

Type based analysis of logic programming is an established field [26, 40, 39, 2, 31, 67]. Much work exists in the field of type inference for logic programming [32, 10, 25, 66, 89, 25].

Pre-interpretation based analyses were introduced in the early 1990s [20, 19, 70]. Tree Automata are increasingly being used in the field of static analysis, abstract interpretation, logic programming etc. [75, 122, 43, 84, 17]. It is a well known property of Tree Automata that any Finite Tree Automaton (FTA) can be turned into a (bottom-up) Deterministic Finite Tree Automaton (DFTA). It is also well known that this transformation can in the worst case result in an exponential increase in the number of states and transitions. Efficient ways of representing automata state spaces has previously been studies [17] including the use of BDDs for this purpose [113].

Chapter 8

Conclusion

An approach to analysing programs written for small microcontrollers used in areas such as wearable computing has been described. The method is based on general program analysis and transformation techniques. The analysis tool has been applied to an example microcontroller architecture and test case programs. Analysis results obtained using this tool includes information about dead code in the programs, more optimal memory allocation and timing information.

The method is based on transforming the object program into a logic program equivalent. This is accomplished by constructing an "interpreter" for the microcontroller and subsequently applying partial evaluation to this interpreter with the object program as static input. This approach is based on an established and well researched topic. For our purpose it has the advantage that the emulator can be instrumented with additional information not part of the underlying language's semantics. For instance adding information about how many clock cycles each instruction takes to complete resulted in a Worst Case Execution Time analysis for the object program. A control flow analysis of the object program was required for offline partial evaluation of the emulator. A Datalog based approach for obtaining flow analyses was described.

An advantage of our approach of relying on logic programming as meta-programs for analysis, is the relative ease of developing a similar tool for a new language or microcontroller. New analysis tools developed for logic programs, which are emerging constantly, would also straightforwardly apply to our target platforms. The analysis tools applied to the transformed programs are mainly abstract interpretation based. Combining the analyses may lead to precision gains compared to applying them individually - this property is known as *the reduced product domain* [53].

The programmer using the analysis tool need not be aware that it relies on logic programming. The prototype implementation of the tool translates and presents the analysis results back to the object program transparently to the user.

Analysis tools for logic programs

Some tools required for our analysis approach are readily available for logic programs, such as partial evaluators and polyhedral programming libraries. Additionally, abstract interpretation based program analysers for logic programs can easily be constructed following well described method. Two analysis tool for logic programs were constructed; a convex polyhedron analyser and a type analysis tool. The analysis tools have been made available via web interfaces. This allows easy demonstration of the capabilities of the tools, convenient experimentation with the tools etc.

Convex polyhedron analyser

The convex polyhedron analyser for constraint logic programs is based on an existing polyhedral library. This has the advantage that as precision and efficiency of the polyhedral operations improve, such progress is easily added to our tool when they are implemented in the programming library. The analyser implements a number of techniques for improving precision of the analysis including a simple narrowing operation for convex polyhedra. A novel technique of computing widening points is implemented that provides a small set of program points to widen on compared to the classical feedback edge method. The precision is equal to those previously reported for convex polyhedral analysers for CLP.

Deterministic regular types

A method for automatically deriving pre-interpretations from regular type specifications was described. It is based on Finite Tree Automata techniques. Determinising the FTA description of the regular types results in a pre-interpretation. An efficient algorithm for determinsing the FTAs was described. It was furthermore described how an analysis based on the derived pre-interpretations can be computed using a BDD-based Datalog solver.

The deterministic type method has been used for propagating binding types for offline partial evaluation, among other applications.

Further work

The polyhedral domain is not suited for approximating programs making use of boolean operations. Other abstractions can be applied to the specialised emulator to improve precision of the approximation. This could, for example, be a bit-size domain where registers are assigned a type based on which bit is the most significant bit in the value contained in the register. For instance, the result of an `OR` operation between two registers would be the largest bit-size of the two operands,

and similarly bit shifting can be given more precise approximations in this domain. This symbolic abstraction could complement the numerical abstraction of the polyhedral domain to produce more precise results. The pre-interpretations based on regular type descriptions could possibly be used for this purpose.

Backwards analysis is another common analysis technique used on logic programs that could be applied to the specialised emulator to detect, for instance, preconditions that guarantees that certain parts of the code, such as error handling routines, are never executed.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] A. Aiken and T. K. Lakshman. Directional Type Checking of Logic Programs. In B. Le Charlier, editor, *Proc. First International Static Analysis Symposium, SAS'94*, volume 864 of *Springer-Verlag Lecture Notes in Computer Science*, 1994.

[3] K. R. Apt and M. H. van Emden. Contributions to the Theory of Logic Programming. *J. ACM*, 29(3):841–862, 1982.

[4] R. Bagnara. A Hierarchy of Constraint Systems for Data-Flow Analysis of Constraint Logic-Based Languages. *Science of Computer Programming*, 30(1–2):119–155, January 1998.

[5] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise Widening Operators for Convex Polyhedra. *Science of Computer Programming*, 2005.

[6] R. Bagnara, P. M. Hill, and E. Zaffanella. A New Encoding and Implementation of Not Necessarily Closed Convex Polyhedra. In M. Leuschel, S. Gruner, and S. L. Presti, editors, *Proceedings of the 3rd Workshop on Automated Verification of Critical Systems, AVoCS 2003*, volume DSSE-TR-2003-2 of *Technical Report*, pages 161 – 176. University of Southampton, 2003.

[7] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. 2006.

[8] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, pages 213–229, London, UK, 2002. Springer-Verlag.

[9] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic Sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM Press.

[10] R. Barbuti and R. Giacobazzi. A Bottom-Up Polymorphic Type Inference in Logic Programming. *Science of Computer Programming*, 19:281–313, 1992.

[11] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Diego, California*, 1987.

[12] F. Benoy and A. King. Inferring Argument Size Relationships with CLP(R). In J. P. Gallagher, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'96)*, volume 1207 of *Springer-Verlag Lecture Notes in Computer Science*, pages 204–223, August 1996.

[13] F. Benoy, A. King, and F. Mesnard. Computing Convex Hulls with a Linear Solver. *Theory and Practice of Logic Programming*, 5:259–271, unknown 2005. (Theory and Practice of Logic Programming was formally known as The Journal of Logic Programming, see http://www.cwi.nl/projects/alp/Welcome/appeal.html).

[14] P. M. Benoy. *Polyhedral Domains for Abstract Interpretation in Logic Programming*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK, January 2002.

[15] R. Berghammer and A. Fronk. Exact Computation of Minimum Feedback Vertex Sets with Relational Algebra. *Fundam. Inf.*, 70(4):301–316, 2006.

[16] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, invited chapter. In T. Mogensen, D. Schmidt, and I. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pages 85–108. Springer-Verlag, Oct. 2002.

[17] J. Börstler, U. Möncke, and R. Wilhelm. Table Compression for Tree Automata. *ACM Transactions on Programming Languages and Systems*, 13(3):295–314, 1991.

[18] A. Bossi, G. M., G. Levi, and M. Martelli. The *s*-semantics Approach: Theory and Applications. *J. Logic Programming*, Special Issue: Ten Years of Logic Programming, Vols.19-20:149–197, 1994.

[19] D. Boulanger and M. Bruynooghe. A Systematic Construction of Abstract Domains. In B. Le Charlier, editor, *Proc. First International Static Analysis Symposium, SAS'94*, volume 864 of *Springer-Verlag Lecture Notes in Computer Science*, pages 61–77, 1994.

[20] D. Boulanger, M. Bruynooghe, and M. Denecker. Abstracting *s*-semantics Using a Model-Theoretic Approach. In M. Hermenegildo and J. Penjam, editors, *Proc. 6*th *International Symposium on Programming Language Implementation and Logic Programming, PLILP'94*, volume 844 of *Springer-Verlag Lecture Notes in Computer Science*, pages 432–446, 1994.

[21] F. Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *Formal Methods in Programming and their Applications*, volume 735 of *Springer-Verlag Lecture Notes in Computer Science*, pages 123–141, 1993.

[22] P. Briggs. Register Allocation via Graph Coloring. Technical Report TR92-183, 24, 1998.

[23] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[24] M. Bruynooghe. A Practical Framework for Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10(2):91–124, 1991.

[25] M. Bruynooghe, J. P. Gallagher, and W. Van Humbeeck. Inference of Well-Typings for Logic Programs with Application to Termination Analysis. In C. Hankin and I. Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2005.

[26] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inferencing. In R. Kowalski and K. Bowen, editors, *Proceedings of ICLP/SLP*, pages 669–683. MIT Press, 1988.

[27] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: Towards the global optimization of prolog programs. In *SLP*, pages 192–204, 1987.

[28] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[29] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog system. Reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. Available from http://www.clip.dia.fi.upm.es/.

[30] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[31] W. Charatonik. Directional Type Checking for Logic Programs: Beyond Discriminative Types. In G. Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000*, volume 1782 of *Springer-Verlag Lecture Notes in Computer Science*, pages 72–87, 2000.

[32] W. Charatonik and A. Podelski. Directional Type Inference for Logic Programs. In G. Levi, editor, *Proceedings of the International Symposium on Static Analysis (SAS'98), Pisa, September 14 - 16, 1998*, volume 1503 of *Springer LNCS*, pages 278–294, 1998.

[33] K. Clark. Predicate Logic as A Computational Formalism. Technical Report DOC 79/59, Imperial College, London, Department of Computing, 1979.

[34] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[35] M. Codish. *Abstract Interpretation of Sequential and Concurrent Logic Programs*. PhD thesis, The Weizmann Institute of Science, 1991.

[36] M. Codish. Efficient Goal Directed Bottom-Up Evaluation of Logic Programs. *Journal of Logic Programming*, 38(3):355–370, 1999.

[37] M. Codish, D. Dams, and E. Yardeni. Bottom-Up Abstract Interpretation of Logic Programs. *Journal of Theoretical Computer Science*, 124:93–125, 1994.

[38] M. Codish and B. Demoen. Analysing Logic Programs Using "Prop"-ositional Logic Programs and a Magic Wand. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming, Vancouver*. MIT Press, 1993.

[39] M. Codish and B. Demoen. Deriving Type Dependencies for Logic Programs Using Multiple Incarnations of Prop. In B. Le Charlier, editor, *Proceedings of SAS'94, Namur, Belgium*, volume 864 of *Springer-Verlag Lecture Notes in Computer Science*, pages 281–296, 1994.

190

[40] M. Codish and V. Lagoon. Type Dependencies for Logic Programs using ACI-Unification. *Theoretical Computer Science*, 238(1-2):131–159, 2000.

[41] M. Codish and H. Søndergaard. Meta-Circular Abstract Interpretation in Prolog. In T. Mogensen, D. Schmidt, and I. H. Sudburough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 109–134. Springer-Verlag, 2002.

[42] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications.* http://www.grappa.univ-lille3.fr/tata, 2002.

[43] H. Comon, D. Kozen, H. Seidl, and M. Vardi. Applications of Tree Automata in Rewriting, Logic and Programming. Schloß Dagstuhl Seminar 9743, http://www.informatik.uni-trier.de/˜seidl/Trees.html, October 20-24, 1997.

[44] K. D. Cooper, A. Dasgupta, and J. Eckhardt. Revisiting Graph Coloring Register Allocation: A Study of the Chaitin -Briggs and Callahan-Koblenz Algorithms. In *LCPC 18: International Workshop on Languages and Compilers for Parallel Computers*. LNCS, 2005.

[45] P. Cousot. Asynchronous Iterative Methods for Solving a Fixed Point System of Monotone Equations in a Complete Lattice. Res. rep. R.R. 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, Sep. 1977. 15 p.

[46] P. Cousot. Semantic Foundations of Program Analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[47] P. Cousot. Abstract Interpretation. *ACM Computing Surveys, Symposium on Models of Programming Languages and Computation*, 28(2):324–328, 1996.

[48] P. Cousot. Abstract Interpretation: Achievements and Perspectives. In *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*. Scuola Superiore G. Reiss Romoli, July 31 – August 6 2000.

[49] P. Cousot. Abstract Interpretation Based Formal Methods and Future Challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[50] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[51] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, 1977.

[52] P. Cousot and R. Cousot. Automatic Synthesis of Optimal Invariant Assertions: Mathematical Foundations. In *ACM Symposium on Artificial Intelligence & Programming Languages*, Rochester, NY, ACM SIGPLAN Not. 12(8):1–12, Aug. 1977.

[53] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Conference Record of the 6th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio,Texas*, pages 269–282. ACM Press, New York, U.S.A., 1979.

[54] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming Leuven,Belgium*, volume 631 of *Springer-Verlag Lecture Notes in Computer Science*, pages 269–295, 1992.

[55] P. Cousot and R. Cousot. Combination of abstractions in the ASTRÉE Static Analyzer. In *Eighth annual IBM Programming Language Day*, Hawthorne, NY, USA, May 7th 2007.

[56] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min, D. Monniaux, and X. Rival. Combination of Abstractions in the ASTRÉE Static Analyzer. In M. Okada and I. Satoh, editors, *Eleventh Annual Asian Computing Science Conference (ASIAN'06)*, pages 1–24, Tokyo, Japan, LNCS ????, Dec. 6–8 2006. Springer, Berlin. (to appear).

[57] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[58] S.-J. Craig, J. P. Gallagher, M. Leuschel, and K. S. Henriksen. Fully Automatic Binding Time Analysis for Prolog. In S. Etalle, editor, *Pre-Proceedings, 14th International Workshop on Logic-Based Program Synthesis and Transformation, LOPSTR 2004, Verona, August 2004*, pages 61–70, 2004.

192

[59] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems: A Case Study. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 17–126, May 1996.

[60] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive Partial Deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming*, 41:231–277, November 1999.

[61] S. Debray and R. Ramakrishnan. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming*, 18:149–176, 1994.

[62] S. Debray and D. Warren. Automatic Mode Inference for Logic Programs. *Journal of Logic Programming*, 5(3):207–229, 1988.

[63] S. Decorte, D. D. Schreye, and H. Vandecasteele. Constraint-Based Termination Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 21(6):1137–1195, 1999.

[64] S. Drape, O. de Moor, and G. Sittampalam. Transforming the .NET Intermediate Language using Path Logic Programming. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 133–144, New York, NY, USA, 2002. ACM Press.

[65] C. Ferdinand. Worst Case Execution Time Prediction by Static Program Analysis. *ipdps*, 03:125a, 2004.

[66] T. Frühwirth. Type Inference by Program Transformation and Partial Evaluation. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*. MIT Press, 1989.

[67] T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic Programs as Types for Logic Programs. In *Proceedings of the IEEE Symposium on Logic in Computer Science, Amsterdam*, July 1991.

[68] Y. Futamura. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[69] J. P. Gallagher. A Bottom-Up Analysis Toolkit. In *Proceedings of the Workshop on Analysis of Logic Languages (WAILL); Eilat, Israel; (also Technical*

193

Report CSTR-95-016, Department of Computer Science, University of Bristol, July 1995), June 1995.

[70] J. P. Gallagher, D. Boulanger, and H. Sağlam. Practical Model-Based Analysis of Definite Logic Programs. Technical Report CSTR-95-11, University of Bristol, Department of Computer Science, 1995.

[71] J. P. Gallagher, D. Boulanger, and H. Sağlam. Practical Model-Based Static Analysis for Definite Logic Programs. In J. W. Lloyd, editor, *Proc. of International Logic Programming Symposium*, pages 351–365. MIT Press, 1995.

[72] J. P. Gallagher and D. de Waal. Regular Approximations of Logic Programs and Their Uses. Technical Report CSTR-92-06, University of Bristol, March 1992.

[73] J. P. Gallagher and D. de Waal. Deletion of Redundant Unary Type Predicates from Logic Programs. In K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, Workshops in Computing, pages 151–167. Springer-Verlag, 1993.

[74] J. P. Gallagher, K. S. Henriksen, and G. Banda. Techniques for Scaling Up Analyses Based on Pre-Interpretations. In M. Gabbrielli and G. Gupta, editors, *Proceedings of the 21st International Conference on Logic Programming, ICLP'2005*, volume 3668 of *Springer-Verlag Lecture Notes in Computer Science*, pages 280–296, 2005.

[75] J. P. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages (PADL'02)*, LNCS, January 2002.

[76] J. P. Gallagher, G. Puebla, and E. Albert. Converting One Type-Based Abstract Domain to Another. In P. M. Hill, editor, *LOPSTR*, volume 3901 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2005.

[77] T. Genet and V. V. T. Tong. Reachability Analysis of Term Rewriting Systems with Timbuk. In R. Nieuwenhuis and A. Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 695–706. Springer, 2001.

[78] D. Gopan and T. W. Reps. Lookahead Widening. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 452–466. Springer, 2006.

[79] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis using Abstract Execution. In *The 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, Rio de Janeiro, Brazil, December 2006.

[80] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of Real-Time Systems using Linear Relation Analysis. *Formal Methods in System Design: An International Journal*, 11(2):157–185, August 1997.

[81] M. Hanus. Horn Clause Programs with Polymorphic Types: Semantics and Resolution. In *TAPSOFT '89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2*, pages 225–240, London, UK, 1989. Springer-Verlag.

[82] J. Hatcliff. An Introduction to Online and Offline Partial Evaluation using a Simple Flowchart Language. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, June 1998.

[83] J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors. *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*, volume 1706 of *Lecture Notes in Computer Science*. Springer, 1999.

[84] N. Heintze. Using Bottom-Up Tree Automaton to Solve Definite Set Constraints. Unpublished. Presentation at Schloß Dagstuhl Seminar 9743, http://www.informatik.uni-trier.de/~seidl/Trees.html, 1997.

[85] K. S. Henriksen and J. P. Gallagher. Analysis and Specialisation of a PIC Processor. In *Proceedings of the 2004 IEEE Conference on Systems, Man and Cybernetics*, The Hague, Netherlands, October 10-13 2004.

[86] M. V. Hermenegildo, F. Bueno, G. Puebla, and P. López. Program Analysis, Debugging, and Optimization Using the Ciao System Preprocessor. In D. De Schreye, editor, *Proceedings of ICLP 1999: International Conference on Logic Programming, Las Cruces, New Mexico, USA*, pages 52–66. MIT Press, 1999.

[87] P. Hill and J. P. Gallagher. *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter Meta-Programming in Logic Programming, pages 421–498. Oxford University Press, 1998.

[88] P. M. Hill and R. W. Topor. A Semantics for Typed Logic Programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.

195

[89] K. Horiuchi and T. Kanamori. Polymorphic Type Inference in Prolog by Abstract Interpretation. In *Proc. 6th Conference on Logic Programming*, volume 315 of *Springer-Verlag Lecture Notes in Computer Science*, pages 195–214, 1987.

[90] J. M. Howe and A. King. Positive Boolean Functions as Multiheaded Clauses. In P. Codognet, editor, *International Conference on Logic Programming*, volume 2237 of *LNCS*, pages 120–134, November 2001.

[91] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119, New York, NY, USA, 1987. ACM Press.

[92] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Software Generation*. Prentice Hall, 1993.

[93] N. D. Jones. Combining Abstract Interpretation and Partial Evaluation. In P. Van Hentenryck, editor, *Symposium on Static Analysis (SAS'97)*, volume 1302 of *Springer-Verlag Lecture Notes in Computer Science*, pages 396–405, 1997.

[94] J. B. Kam and J. D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Inf.*, 7:305–317, 1977.

[95] R. M. Karp. Reducibility Among Combinatorial Problems. pages 85–103, 1972.

[96] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[97] U. P. Khedker and D. M. Dhamdhere. A Generalized Theory of Bit Vector Data Flow Analysis. *ACM Transactions on Programming Languages and Systems*, 16(5):1472–1511, September 1994.

[98] A. King, K. Shen, and F. Benoy. Lower-Bound Time-Complexity Analysis of Logic Programs. In *ILPS '97: Proceedings of the 1997 international symposium on Logic programming*, pages 261–275, Cambridge, MA, USA, 1997. MIT Press.

[99] J. Knoop and B. Steffen. The Interprocedural Coincidence Theorem. In *Computational Complexity*, pages 125–140, 1992.

[100] H. J. Komorowski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *META*, volume 649 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 1992.

[101] B. Le Charlier and P. Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical report, Providence, RI, USA, 1992.

[102] M. Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, Department of Computer Science, K.U. Leuven, 1997.

[103] M. Leuschel. A Framework for the Integration of Partial Evaluation and Abstract Interpretation of Logic Programs. *ACM Trans. Program. Lang. Syst.*, 26(3):413–463, 2004.

[104] M. Leuschel, S.-J. Craig, M. Bruynooghe, and W. Vanhoof. Specialising Interpreters Using Offline Partial Deduction. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, volume 3049 of *Lecture Notes in Computer Science*, pages 340–375. Springer, 2004.

[105] M. Leuschel, S.-J. Craig, and D. Elphick. Supervising Offline Partial Evaluation of Logic Programs using Online Techniques. *Proceedings LOPSTR'06*, pages 43–59, 2006.

[106] M. Leuschel, D. Elphick, M. Varea, S.-J. Craig, and M. Fontaine. The Ecce and Logen Partial Evaluators and their Web Interfaces. In F. T. John Hatcliff, editor, *Proceedings PEPM 06*, pages 88–94. IBM Press, Januar 2006.

[107] M. Leuschel and J. Jørgensen. Efficient Specialisation in Prolog Using the Hand-Written Compiler Generator LOGEN. *Elec. Notes Theor. Comp. Sci.*, 30(2), 1999.

[108] M. Leuschel and T. Massart. Infinite State Model Checking by Abstract Interpretation and Program Specialisation. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'99)*, volume 1817 of *Springer-Verlag Lecture Notes in Computer Science*, pages 63–82, April 2000.

[109] M. Leuschel and M. H. Sorensen. Redundant Argument Filtering of Logic Programs. In *Logic Program Synthesis and Transformation*, pages 83–103, 1996.

[110] J. Lind-Nielsen. BuDDy, a binary decision diagram package, 2004. http://sourceforge.net/projects/buddy/.

[111] B. Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In J. Gustafsson, editor, *Proc. Third International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 77–80, Porto, July 2003.

[112] J. W. Lloyd. *Foundations of Logic Programming: 2nd Edition*. Springer-Verlag, 1987.

[113] M. Mach and F. Plasil. Addressing State Explosion in Behavior Protocol Verification. In *SNPD*, pages 327–333. IEEE Computer Society, 2004.

[114] K. Marriott and H. Søndergaard. Bottom-Up Abstract Interpretation of Logic Programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming, Washington*, August 1988.

[115] K. Marriott and H. Søndergaard. Semantics-Based Dataflow Analysis of Logic Programs. In G. X. Ritter, editor, *Information Processing 89*, pages 601–606. North-Holland, 1989.

[116] C. S. Mellish. Abstract Interpretation of Prolog Programs. In *Proceedings on Third International Conference on Logic Programming*, pages 463–474, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[117] C. S. Mellish. Using Specialisation to Reconstruct Two Mode Inference Systems. Technical report, University of Edinburgh, March 1990.

[118] F. Mesnard and R. Bagnara. cTI: A Constraint-Based Termination Inference Tool for ISO-Prolog. *Theory Pract. Log. Program.*, 5(1-2):243–257, 2005.

[119] F. Mesnard and J.-G. Ganascia. CLP(Q) for Proving Interargument Relations. In *META-92: Proceedings of the 3rd International Workshop on Meta-Programming in Logic*, pages 308–320, London, UK, 1992. Springer-Verlag.

[120] Microchip. PIC16F84A Data Sheet.

[121] P. Mishra. Towards a Theory of Types in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, 1984.

[122] D. Monniaux. Abstracting Cryptographic Protocols with Tree Automata. *Sci. Comput. Program.*, 47(2-3):177–202, 2003.

[123] A. Moss and H. Muller. Efficient Code Generation for a Domain Specific Language. In *Generative Programming and Component Engineering*, pages 47–62. Springer Verlag, September 2005.

[124] P. Naur. Checking of Operand Types in ALGOL Compilers. *BIT Numerical Mathematics*, 5(3):151–163, 1965.

[125] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, December 2004.

[126] H. R. Nielson and F. Nielson. *Semantics With Applications: a Formal Introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.

[127] J. Peralta, J. P. Gallagher, and H. Sağlam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In G. Levi, editor, *Static Analysis. 5th International Symposium, SAS'98, Pisa*, volume 1503 of *Springer-Verlag Lecture Notes in Computer Science*, pages 246–261, 1998.

[128] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[129] G. Puebla, M. Hermenegildo, and J. P. Gallagher. An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework. In O. Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, Technical report BRICS-NS-99-1, University of Aarhus, pages 75–84, San Antonio, Texas, Jan. 1999.

[130] J. Regehr and A. Reid. HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 133–143, New York, NY, USA, 2004. ACM Press.

[131] T. W. Reps. Demand Interprocedural Program Analysis Using Logic Databases. In *Workshop on Programming with Logic Databases (Book), ILPS*, pages 163–196, 1993.

[132] J. Rohmer, R. Lescoeur, and J. M. Kerisit. The Alexander Method-A Technique for the Processing of Recursive Axioms in Deductive Databases. *New Gen. Comput.*, 4(3):273–285, 1986.

[133] B. J. Ross. The Partial Evaluation of Imperative Programs Using Prolog. In *META*, pages 341–363, 1988.

[134] A. Rountev, B. G. Ryder, and W. Landi. Data-Flow Analysis of Program Fragments. In *ESEC / SIGSOFT FSE*, pages 235–252, 1999.

[135] A. Roychoudhury, K. N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In S. Graf and M. I. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems, 6th Int. Conf., TACAS 2000*, volume 1785 of *Springer-Verlag Lecture Notes in Computer Science*, pages 172–187, 2000.

[136] S. K. D. S. and R. Ramakrishnan. Abstract Interpretation of Logic Programs using Magic Transformations. *J. Log. Program.*, 18(2):149–176, 1994.

[137] S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A Logic-Based Approach to Data Flow Analysis Problem. In *PLILP '90: Proceedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming*, pages 277–292, London, UK, 1990. Springer-Verlag.

[138] D. Saha and C. R. Ramakrishnan. Incremental and Demand-Driven Points-to Analysis using Logic Programming. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 117–128, New York, NY, USA, 2005. ACM Press.

[139] D. A. Schmidt. Abstract Interpretation of Small-Step Semantics. In *Selected papers from the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, pages 76–99, London, UK, 1997. Springer-Verlag.

[140] D. Seipel, M. Hopfner, and B. D. Heumesser. Analyzing and Visualising Prolog Programs based on XML Representations. In F. Mesnard and A. Serebrenik, editors, *WLPE*, volume CW371 of *Report*, pages 31–45. Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee (Belgium), 2003.

[141] H. Seki. On the Power of Alexander Templates. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 150–159, New York, NY, USA, 1989. ACM Press.

[142] A. Simon and A. King. Widening Polyhedra with Landmarks. In N. Kobayashi, editor, *Fourth Asian Symposium on Programming Languages and Systems*, volume 4279 of *Lecture Notes in Computer Science*, pages 166–182. Springer Verlag, November 2006. See also http://www.springer.de/comp/lncs/index.html.

[143] Z. Somogyi, F. Henderson, and T. Conway. Mercury: An Efficient Purely Declarative Logic Programming Language, 1995.

[144] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. 1955.

[145] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software. *dsn*, 00:625, 2003.

[146] J. Ullman. Implementation of Logical Query Languages for Databases. *ACM Transactions on Database Systems*, 10(3), 1985.

[147] J. Ullman. *Principles of Knowledge and Database Systems; Volume 1*. Computer Science Press, 1988.

[148] J. D. Ullman and A. Van Gelder. Efficient Tests for Top-Down Termination of Logical Rules. *J. ACM*, 35(2):345–373, 1988.

[149] M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *J. ACM*, 23(4):733–742, 1976.

[150] A. Van Gelder. Deriving Constraints Among Argument Sizes in Logic Programs. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems; Nashville, Tennessee*, pages 47–60, 1990.

[151] P. van Roy and A. M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1):54–68, 1992.

[152] J. Whaley and M. S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis using Binary Decision Diagrams. In W. Pugh and C. Chambers, editors, *PLDI*, pages 131–144. ACM, 2004.

[153] J. Whaley, C. Unkel, and M. S. Lam. A BDD-Based Deductive Database for Program Analysis, 2004. `http://bddbddb.sourceforge.net/`.

[154] E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Journal of Logic Programming*, 10(2):125–154, 1990.

RECENT RESEARCH REPORTS

#116 Marco Baroni, Alessandro Lenci, and Magnus Sahlgren, editors. *Proceedings of the 2007 Workshop on Contextual Information in Semantic Space Models: Beyond Words and Documents*, Roskilde, Denmark, August 2007.

#115 Paolo Bouquet, Jérôme Euzenat, Chiara Ghidini, Deborah L. McGuinness, Valeria de Paiva, Luciano Serafini, Pavel Shvaiko, and Holger Wache, editors. *Proceedings of the 2007 workshop on Contexts and Ontologies Representation and Reasoning (C&O:RR-2007)*, Roskilde, Denmark, August 2007.

#114 Bich-Liên Doan, Joemon Jose, and Massimo Melucci, editors. *Proceedings of the 2nd International Workshop on Context-Based Information Retrieval*, Roskilde, Denmark, August 2007.

#113 Henning Christiansen and Jørgen Villadsen, editors. *Proceedings of the 4th International Workshop on Constraints and Language Processing (CSLP 2007)*, Roskilde, Denmark, August 2007.

#112 Anders Kofod-Petersen, Jörg Cassens, David B. Leake, and Stefan Schulz, editors. *Proceedings of the 4th International Workshop on Modeling and Reasoning in Context (MRC 2007) with Special Session on the Role of Contextualization in Human Tasks (CHUT)*, Roskilde, Denmark, August 2007.

#111 Ioannis Hatzilygeroudis, Alvaro Ortigosa, and Maria D. Rodriguez-Moreno, editors. *Proceedings of the 2007 workshop on REpresentation models and Techniques for Improving e-Learning: Bringing Context into the Web-based Education (ReTIeL'07)*, Roskilde, Denmark, August 2007.

#110 Markus Rohde. *Integrated Organization and Technology Development (OTD) and the Impact of Socio-Cultural Concepts — A CSCW Perspective*. PhD thesis, Roskilde University, Roskilde, Denmark, 2007.

#109 Keld Helsgaun. An effective implementation of k-opt moves for the Lin-Kernighan TSP heuristic. 2006, Roskilde University, Roskilde, Denmark.

#108 Pernille Bjørn. *Virtual Project Teams — Distant Collaborative Practice and Groupware Adaptation*. PhD thesis, Roskilde University, Roskilde, Denmark, 2006.

#107 Henrik Bulskov Styltsvig. *Ontology-based Information Retrieval*. PhD thesis, Roskilde University, Roskilde, Denmark, 2006.

#106 Rasmus Knappe. *Measures of Semantic Similarity and Relatedness for Use in Ontology-based information Retrieval*. PhD thesis, Roskilde University, Roskilde, Denmark, 2006.

#105 Davide Martinenghi. *Advanced Techniques for Efficient Data Integrity Checking*. PhD thesis, Roskilde University, Roskilde, Denmark, 2005.