TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

Interprocedural Polynomial Invariants

Dipl-Inf. Univ. Michael Petter

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Francisco J. Esparza Estaun
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Helmut Seidl
	2. Univ.-Prof. Dr. Markus Müller-Olm, Westfälische Wilhelms-Universität Münster

Die Dissertation wurde am 12.01.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 13.09.2010 angenommen.

Abstract

This thesis describes techniques for static analysis of *polynomial equalities* in *interprocedural* programs. It elaborates on approaches for analysing polynomial equalities over different domains as well as techniques to apply polynomial analysis to infer interprocedurally valid equalities of uninterpreted terms.

This work is organised in three major theoretical parts, followed by a practical part. In the first part forward and backward frameworks for inferring polynomial equalities are presented and extended to deal with procedure calls. It is shown how both approaches make use of *polynomial ideals* as abstraction of program states. Since the performance of operations on polynomial ideals is highly dependent on the way the ideals are represented in detail, the most crucial representation issues are highlighted here. In the second part, values of variables are treated as integers modulo a power of two which coincides with the treatment of integers in most current architectures. This part provides methods for inferring polynomial invariants modulo 2^w. The third part is dedicated to equalities of uninterpreted terms, so called *Herbrand equalities*. It is inspected in how far polynomial ideals can be applied to interprocedurally infer *Herbrand equalities*. This gives rise to a novel subclass of the general problem which can be analysed precisely. In the practical part, I finally elaborate on my implementation of a framework for performing interprocedural program analyses on real-world code. I give an overview over the architecture of the analyser's fixpoint iteration engine, target architecture plugin system and a short description how to implement other analyses in this framework.

Zusammenfassung

In dieser Arbeit geht es um Techniken der *statischen Analyse* von polynomiellen Gleichungen in *interprozeduralen* Programmen. Dabei werden sowohl Ansätze zur Analyse von polynomiellen Gleichungen für verschiedene Wertebereiche als auch Techniken zur Anwendung von Polynomanalysen zur Herleitung von gültigen Gleichungen über uninterpretierten Termen in Programmen mit Prozeduraufrufen thematisiert.

Diese Arbeit ist aufgegliedert in drei große theoretische Teile, denen ein praktischer Teil folgt. Im ersten Teil werden die Grundsysteme zur Herleitung von Polynomgleichungen sowohl rückwärts wie auch vorwärts vorgestellt und um das Konzept von Prozeduraufrufen angereichert. Es wird verdeutlicht, wie beide Ansätze sich auf *Polynomideale* abstützen um Programmzustände zu abstrahieren. Da die Laufzeit der einzelnen Vorgängen auf den Polynomidealen davon abhängt wie die Ideale repräsentiert werden werden die wichtigsten Darstellungsaspekte hier vorgestellt. Im zweiten Teil wird dazu übergegangen, Werte von Variablen als Ganzzahlen Modulo einer Zweierpotenz zu betrachten, was genau der Behandlung von Ganzzahlen in den meisten aktuell verwendeten Rechnerarchitekturen entspricht. In diesem Teil werden die Werkzeuge bereitgestellt, um Polynominvarianten modulo 2^w herzuleiten. Der dritte Teil ist Gleichungen von uninterpretierten Termen, sogenannten *Herbrand Gleichungen* gewidmet. Es wird untersucht, inwieweit sich Polynomideale dazu eignen, bei der Herleitung von Herbrand Gleichungen in Programmen mit Prozeduren zu helfen. Dabei ergibt sich eine neue Unterklasse des allgemeinen Problems, die präzise analysiert werden kann. Im praktischen Teil geht es um die Implementierung eines Frameworks zur interprozeduralen Analyse von praktisch relevantem Code. Dabei wird ein Überblick gegeben über die Architektur des Gleichungslösers, über das Pluginsystem für verschiedene Maschinenarchitekturen und eine Beschreibung, wie neue Analysen in dieses Framework integriert werden können.

Publications

Some of the results of this work have been presented in the following publications:

[1] Markus Müller-Olm, Michael Petter and Helmut Seidl. Interprocedurally Analyzing Polynomial Identities. In *23rd International Symposium on Theoretical Aspects of Computer Science (***STACS***)*, February 2006.

[2] Andrea Flexeder, Michael Petter, Helmut Seidl. Analysing All Polynomial Equations in $\mathbb{Z}_{2^w}$ In *15th International Static Analysis Symposium (***SAS***)*, July 2008.

Acknowledgement

I would like to thank

Andrea Flexeder, **Bogdan Mihaila** and **Helmut Seidl**

for their efforts in proofreading, commenting, discussing and collaborating in order to accomplish this thesis.

Contents

1 Introduction **1**

2 Interprocedural polynomial invariants **5**

2.1 A primer on polynomials 5
2.1.1 Terms and term orders 5
2.1.2 Monomials and polynomials 6
2.1.3 Polynomial reduction 7
2.1.4 Polynomial ideals and Gröbner bases 7
2.1.5 Finding Gröbner bases with Buchbergers algorithm 8
2.1.6 Basic ideal algorithms 9
2.2 Program analysis basics 11
2.2.1 Control flow graphs 11
2.2.2 Semantics of programs 12
2.3 Analysing polynomials in $\mathbb{Q}$ 14
2.3.1 General setup 15
2.3.2 Forward analysis framework 17
2.3.3 Backward analysis framework 20
2.3.4 Optimistic procedure effect tabulation 24
2.3.5 Interprocedural analysis with transition invariants 25
2.3.6 Interprocedural analysis with WP transformers 27
2.3.7 Equality guards 30
2.3.8 Local variables 32
2.3.9 Conclusion 35
2.4 Analysing polynomials in $\mathbb{Z}_{2^w}$ 36
2.4.1 Related work 37
2.4.2 Concrete semantics 38
2.4.3 The ring of polynomials in $\mathbb{Z}_{2^w}$ 39
2.4.4 Verifying polynomial relations in $\mathbb{Z}_{2^w}$ 42
2.4.5 Computing with Ideals over $\mathbb{Z}_{2^w}[\mathbf{X}]$ 44
2.4.6 Inferring all polynomials 46
2.4.7 Implementation of modular analysis 48
2.4.8 Summary and prospects 50

3 Interprocedural Herbrand equaltities **51**
3.1 An introduction to Herbrand programs 53
3.1.1 Herbrand equalities in general 53
3.1.2 Herbrand programs 56
3.2 Analysing Herbrand equalities encoded as polynomials 60
3.2.1 Herbrand equalities as polynomial ideals 60
3.2.2 Herbrand programs as polynomial programs 66
3.2.3 Representing procedure effects on Herbrand equalities 69
3.2.4 Interprocedurally inferring Herbrand equalities 72
3.3 Herbrand constants 79
3.4 Local variables 86
3.5 Conclusion 89

4 Implementing a framework for program analyses **91**
4.1 Usage scenarios 91
4.2 Demands on tool support 92
4.3 Developing VoTUM 96
4.3.1 Ideas for implementation 96
4.3.2 Technical Realisation 99
4.3.3 Example analysis 112
4.4 Further Improvements 118

5 Conclusion **119**

Bibliography **120**

Chapter 1

Introduction

Since programming languages are Turing complete, it is impossible to decide for all programs whether a given non-trivial semantic property is valid or not. Nonetheless, by abstracting away certain features of programs, we achieve at least approximate results for properties of programs, possibly missing all properties that are valid in reality. However, often specially restricted program classes can be given, for which the approximate methods provide exact results. The methods, shown in this thesis are based on this concept. In particular, we concentrate on the following topics:

Polynomial invariants in $\mathbb{Q}$

Here, we consider analyses of polynomial identities between integer variables such as $\mathbf{x}_1 \cdot \mathbf{x}_2 - 2\mathbf{x}_3 = 0$. We describe current approaches and clarify their completeness properties. We further concentrate on an extension of our approach based on weakest precondition computations to programs with procedures, with or without local variables and equality guards.

Polynomial invariants in $\mathbb{Z}_{2^w}$

In this part, we present methods for checking and inferring all valid polynomial relations in $\mathbb{Z}_{2^w}$ in programs. In contrast to the infinite field $\mathbb{Q}$, $\mathbb{Z}_{2^w}$ is finite and hence allows for finitely many polynomial functions only. For this topic we show, that checking the validity of a polynomial invariant over $\mathbb{Z}_{2^w}$ is, though decidable, only *PSPACE*-complete. Apart from the impracticable algorithm for the theoretical upper bound, we present a feasible algorithm for verifying polynomial invariants over $\mathbb{Z}_{2^w}$ which runs in polynomial time if the number of program variables is bounded by a constant. In this case, we also obtain a polynomial-time algorithm for inferring all polynomial relations. In practise, our approach provides us with a feasible algorithm to infer all polynomial invariants up to a low degree.

Herbrand equalities

This part deals with analyses of identities between program variables based on uninterpreted terms, such as $a(b(\mathbf{x}_1, c), \mathbf{x}_2) = a(\mathbf{x}_2, b(b(c, c), c))$. We first introduce a general framework for analysing preconditions of Herbrand equalities in programs with procedure calls. However, lacking a concise general representation of procedure effects, the main message here consists in finding special instances of the general problem, which permit an effective analysis. Our key contribution for this topic then consists in presenting two such special instances.

First, we introduce, how to encode Herbrand terms and equalities as polynomials and ideals. This encoding permits to reduce the problem of verifying a Herbrand equality to the problem of verifying a polynomial. Based on this discovery, we get the possibility to implement procedure call effects same as in the case of polynomial equalities. This yields special restrictions on the class of programs which can be analysed exactly with this approach, leading to the result, that via polynomial encoding, we can infer all Herbrand equalities in programs with assignments, that do have not more than one variable on the right-hand-side of their assignments.

Second, we provide a solution to the problem of inferring all constant Herbrand equalities for a program point by the observation, that representing a procedures effect by tabulating only its effect on generic equalities $\mathbf{x}_i \doteq \bullet$ suffices to exactly perform weakest precondition transformations.

Practical implementation

The topic of the last part of the thesis is about the implementation of a framework for performing interprocedural analyses, called VoTUM. The experiences that we made during the practical implementation of the program analyses from the other parts lead to the perception of several needs which arose during the implementation of the prototypes for the analyses. This comprises rather unique aspects as being able to visualise different steps of the fixpoint iteration on the abstracted program states and debugging the data structures which belong to these implementations during the run of an analysis. Further, as most analyses developed in this thesis share common principles like the view of programs as graphs, with common structural characteristics and semantics. Here the idea is to save development time by sharing these common features between all analysis implementations. All this lead to the implementation of a specific framework for developing and performing analyses of programs as control flow graphs via abstract interpretation.

Outline

Chapter 2 starts with an introduction to polynomials and ideals, introducing basic notations and helper functions to deal with the structure of polynomials, as well as standard algorithms for ideals. After this, we define the model of programs, to which we refer our abstract interpretation based analyses in a generic way. This is followed by

a section, summarising the different approaches to analyse polynomial equalities over fields, concentrating on $\mathbb{Q}$. The last section of this chapter finally exposes the differences arising from interpreting program semantics not over $\mathbb{Q}$ but over the residue class ring $\mathbb{Z}_{2^w}$.
Chapter 3 starts with an introduction to Herbrand interpretation and defining the semantics of Herbrand programs, as a basis for the definition of a framework for interprocedural analysis of Herbrand equalities. The next section then treats the encoding of Herbrand equalities as polynomials, and elaborates on the consequences of implementing procedure effects for Herbrand equalities with preconditions of polynomial templates. The following section treats the topic of inferring all Herbrand constants by means tabulating the effect of the precondition transformers. Finally, this chapter ends with the addition of local variables to the framework.
Chapter 4 starts with a collection of the different uses, which are crucial for the a practical framework for program analyses during the development of program analyses. Then, we continue with a more condensed summary of the key requirements deduced from the use cases. This is followed by the concrete description of the solution to these requirements, consisting of a rather general overview of the concepts as well as of the technical realisation and the interaction of the involved components. Finally, we sketch further improvement plans of features that are still missing in the implementation.

Chapter 2

Interprocedural polynomial invariants

2.1 A primer on polynomials

Throughout this thesis, polynomials and polynomial rings are core components for most of the presented analyses. In the following, we therefore discuss a few general foundations of polynomials as well as a number of algorithms, which are important for the proceeding sections. For a more thorough insight into polynomials and polynomial ideals, we recommend a look into [BW93], whose remarks are the base for this chapter.

2.1.1 Terms and term orders

We emanate from a set of variables $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_k\}$ of arity n. A product of variables $\mathbf{x}_1^{\nu_1} \cdot \dots \cdot \mathbf{x}_k^{\nu_k}$ with integer exponents $\nu_i \geq 0$ is called a *term* in the variables from $\mathbf{X}$. In particular, the special term $\mathbf{x}_1^0 \cdot \dots \cdot \mathbf{x}_k^0$ is called 1. By $\mathbf{T}[\mathbf{X}]$ we denote the set of all terms in the variables from $\mathbf{X}$. Multiplication on terms can be performed by adding the corresponding exponents of the two terms. Divisibility of two terms is defined corresponding to the natural partial order on their exponent tuple:

$$\mathbf{x}_1^{\nu_1} \cdot \dots \cdot \mathbf{x}_k^{\nu_k} \mid \mathbf{x}_1^{\mu_1} \cdot \dots \cdot \mathbf{x}_k^{\mu_k} \Leftrightarrow (\nu_1, \dots, \nu_k) \leq (\mu_1, \dots, \mu_k)$$

The following algorithms though require a complete order on terms, which has the following properties (c.f. [BW93]): A *term order* is a total order on $\mathbf{T}[\mathbf{X}]$ which satisfies the conditions

- $1 \leq t$ for all $t \in \mathbf{T}[\mathbf{X}]$
- $t_1 \leq t_2$ implies $t_1 \cdot s \leq t_2 \cdot s$ for all $s, t_1, t_2 \in \mathbf{T}[\mathbf{X}]$

Candidates for term orders, which are suitable for purposes of our analyses are:

(i) $\mathbf{x}_1^{\nu_1} \cdot \dots \cdot \mathbf{x}_k^{\nu_k} \;\leq\; \mathbf{x}_1^{\mu_1} \cdot \dots \cdot \mathbf{x}_k^{\mu_k}$ iff $(\nu_1, \dots, \nu_k) = (\mu_1, \dots, \mu_k)$, or there exists an i, with $\nu_j = \mu_j$ for $j < i$ and $\nu_i \leq \mu_i$. This order is called *lexicographical* order on $\mathbf{T}[\mathbf{X}]$.

(ii) $\mathbf{x}_1^{\nu_1} \cdot \dots \cdot \mathbf{x}_k^{\nu_k} \;\leq\; \mathbf{x}_1^{\mu_1} \cdot \dots \cdot \mathbf{x}_k^{\mu_k}$ iff $(\nu_1, \dots, \nu_k) = (\mu_1, \dots, \mu_k)$, or there exists an i, with $\nu_j = \mu_j$ for $j > i$ and $\nu_i \leq \mu_i$. This order is called *inverse lexicographical* order on $\mathbf{T}[\mathbf{X}]$.

(iii) $\mathbf{x}_1^{\nu_1} \cdot \dots \cdot \mathbf{x}_k^{\nu_k} \leq \mathbf{x}_1^{\mu_1} \cdot \dots \cdot \mathbf{x}_k^{\mu_k}$ iff $\sum_{i=1}^{k} \nu_i < \sum_{i=1}^{k} \mu_i$ or $\sum_{i=1}^{k} \nu_i = \sum_{i=1}^{k} \mu_i$ and $\mathbf{x}_1^{\nu_1} \cdot \dots \cdot \mathbf{x}_k^{\nu_k} \leq' \mathbf{x}_1^{\mu_1} \cdot \dots \cdot \mathbf{x}_k^{\mu_k}$ with $\leq'$ a term order. This term order composed with a lexicographical term order is called *total degree lexicographical* order.

For sets of variables $\mathbf{U} \subseteq \mathbf{X}$, we define the property $\ll$ with respect to a term order $\leq$: We say that $\mathbf{U} \ll \mathbf{X} \setminus \mathbf{U}$ if $s < t$ for all $s \in T(\mathbf{U})$ and $t \in T(\mathbf{X} \setminus \mathbf{U})$. Note that typical term orders, that fulfil this property are lexicographic term orders, where all variables $\in \mathbf{U}$ are less then the other variables $\in \mathbf{X} \setminus \mathbf{U}$.

2.1.2 Monomials and polynomials

So far, we did only consider the syntax of terms. Semantically, we permit a variable from $\mathbf{X}$ to take a value from an arbitrary ring $\mathcal{R}$. Additionally, now we also consider coefficients for terms from the ring $\mathcal{R}$. Therefore, we introduce an n-ary coefficient function a which maps tuples of exponents $(\nu_1, \dots, \nu_k)$ to values from $\mathcal{R}$. Then, we call a product of the form

$$a_{(\nu_1,\dots,\nu_k)} \cdot \mathbf{x}_1^{\nu_1} \cdot \dots \cdot \mathbf{x}_k^{\nu_k}$$

a *monomial*. The set of monomials in $\mathbf{X}$ over a ring $\mathcal{R}$ is denoted by $M[\mathcal{R}, \mathbf{X}]$. We obtain a preorder $\preceq$ on monomials by extending their term order according to $t_1 \leq t_2 \Leftrightarrow a_1 \cdot t_1 \preceq a_2 \cdot t_2$. By abuse of notation, we will also write $\leq$ for the preorder on $M[\mathcal{R}, \mathbf{X}]$. Multiplication of monomials can be performed by multiplying their coefficients, respectively terms separately, e.g. $a_1 t_1 \cdot a_2 t_2 = (a_1 a_2)(t_1 t_2)$.

Polynomials p, multivariate in $\mathbf{X}$, can be written as a sum of multivariate monomials:

$$p = \sum_{(\nu_1,\dots,\nu_k) \in \mathbb{N}_0^k} a_{(\nu_1,\dots,\nu_k)} \cdot \mathbf{x}_1^{\nu_1} \cdot \dots \cdot \mathbf{x}_k^{\nu_k}$$

Given a term order $\leq$, we can represent a polynomial p uniquely by $\sum_{m_i \in M[\mathcal{R},\mathbf{X}]} m_i$ with $m_i > m_{i+1}$. Furthermore, we obtain a unique representation for any polynomial p by sorting and then combining all monomials with similar terms. We call polynomials of this form *normalised*. In $p = a_* t^* + \sum_{m_i \in M[\mathcal{R},\mathbf{X}]} m_i$ with $a_* t^* > m_i$ we call a_* the *head coefficient*, t^* the *head term* and $a_* t^*$ the *head monomial* of p. For short, we write $a_* = HC(p)$, $t^* = HT(p)$ and $a_* t^* = HM(p)$. Additionally, we denote by $T(p)$ the set of terms, that occur in p with a non-zero coefficient, where as $\mathbf{Vars}(p)$ yields the set of variables, which occur in p.

As a polynomial is an ordered sequence of monomials, we can introduce a preorder $\preceq$ on polynomials:
$\sum_{m_i \in T[\mathbf{X}]} a_i m_i \preceq \sum_{n_i \in T[\mathbf{X}]} b_i n_i$ iff $m_i = n_i$, or there exists a $j > i$ with $m_i = n_i$ and $m_j \preceq n_j$.

We denote the ring of all polynomials over a ring $\mathcal{R}$, multivariate in $\mathbf{X}$, by $\mathcal{R}[\mathbf{X}]$. In the course of this thesis we treat the polynomial rings $\mathbb{Q}[\mathbf{X}]$ and $\mathbb{Z}_{2^w}[\mathbf{X}]$.

2.1.3 Polynomial reduction

Multiplication of multivariate polynomials can be done quite straightforward. What is a little more challenging is the concept of long division of multivariate polynomials, which we call *reduction*. For this, we assume a fixed term order $\preceq$.

We say that a polynomial $p = a \cdot t + \sum_{t_i \neq t} a_i t_i$ can be reduced to r modulo q, eliminating t, if there exists an s with $s \cdot HT(q) = t$ and

$$r = p - \frac{a}{HC(q)} \cdot s \cdot q$$

We denote this relation with $p \xrightarrow[q]{} r\ [t]$. In most cases, it is sufficient to express, that there exists a t, for which p can be reduced by q to r, which is expressed by $p \xrightarrow[q]{} r$. We can extend this definition to cover also division by a set of polynomials as follows: p reduces to r modulo Q if there exists a $q \in Q$ such that $p \xrightarrow[q]{} r$. The notation for this is $p \xrightarrow[Q]{} r$. We call a polynomial p, for which there exists no r such that $p \xrightarrow[Q]{} r$ *irreducible* modulo Q. An irreducible polynomial r is the normal form of a polynomial p, if it satisfies $p \xrightarrow[Q]{*} r$, where $\xrightarrow[Q]{*}$ is the reflexive-transitive closure of $\xrightarrow[Q]{}$. Note, that normal forms are not necessarily unique modulo a set of polynomials Q. We call $p \xrightarrow[q]{} r\ [t]$ a *top-reduction*, if $t = HT(p)$, and p *top-reducible*.

2.1.4 Polynomial ideals and Gröbner bases

In this place, we recall that the set $\mathcal{R}[\mathbf{X}]$ of all polynomials forms a *commutative ring*. A non-empty subset I of a commutative ring $\mathcal{R}$ satisfying the two conditions:

(i) $a + b \in I$ whenever $a, b \in I$ (closure under sum) and

(ii) $r \cdot a \in I$ whenever $r \in \mathcal{R}$ and $a \in I$ (closure under product with arbitrary ring elements)

is called an *ideal*.

Example 1 . Let p, p_1, p_2, r *be polynomials over some ring. Clearly, if* $p(x) = 0$ *is valid then also* $r(x) \cdot p(x) = 0$ *for arbitrary polynomials* r*. Also, if* $p_1(x) = 0$ *and* $p_2(x) = 0$ *are valid then also* $p_1(x) + p_2(x) = 0$ *is valid. Thus, the set of polynomials* p *for which* $p(x) = 0$ *is valid forms a* polynomial ideal.

Ideals (in particular those in polynomial rings) enjoy interesting and useful properties. For a subset $G \subseteq \mathcal{R}[\mathbf{X}]$, the least ideal I containing G is given by

$$I = \{r_1 g_1 + \ldots + r_k g_k \mid n \geq 0, r_i \in \mathcal{R}, g_i \in G\}$$

or for short $I = \langle G \rangle$. In this case, G is also called a set of *generators* of I.

In this context, it is especially interesting to recall, that if computing over fields $\mathcal{F}$, we also have a statement, that all polynomial ideals are even finitely generated:

Theorem 1 .[Hilbert, 1888] *Every ideal* $I \subseteq \mathcal{F}[\mathbf{X}]$ *of a commutative polynomial ring in finitely many variables* $\mathbf{X}$ *over a field* $\mathcal{F}$ *is finitely generated, i.e.,* $I = \langle G \rangle$ *for a finite subset* $G \subseteq \mathcal{F}[\mathbf{X}]$. □

Considering membership of an ideal, we find that

$$p \xrightarrow[R]{*} 0 \implies p \in \langle R \rangle$$

Note, that the opposite direction does not necessarily hold.
Example 2 . Consider the polynomial $p = \mathbf{x} + 1$ and the set $Q = \{\mathbf{x}, \mathbf{x} + 1\}$. Then the normal form of p modulo Q is not uniquely determined, depending on which polynomial is used for reduction:

$$p \xrightarrow[\mathbf{x}]{} 1 \text{ and } p \xrightarrow[\mathbf{x}+1]{} 0$$

Anyway, there are special constraints on generator sets, for which every normal form is unique. This is especially futile, when trying to prove membership in an ideal:
Example 3 . Consider the set of polynomials $S = \{\mathbf{xy} + 1, \mathbf{yz} + 1\}$ and the polynomial $p = \mathbf{z} - \mathbf{x}$. We find, that $p = \mathbf{z}(\mathbf{xy} + 1) - \mathbf{x}(\mathbf{yz} + 1) \in \langle S \rangle$. Nevertheless p is in normal form modulo S, no matter which term order.

However, there exist sets of polynomials, which allow for polynomial reduction to uniquely determine membership. We call such a finite generator system $G \subseteq \mathcal{R}[\mathbf{X}]$ a *Gröbner base* w.r.t. a fixed term order $\leq$, iff $p \xrightarrow[G]{*} 0$ for all $p \in \langle G \rangle$.

Theorem 2 . (Gröbner Bases [BW93] 5.41) *Let I be an ideal of $\mathcal{R}[\mathbf{X}]$. Then there exists a Gröbner base G of I w.r.t. $\leq$.*

Example 4 . Let us consider example 3 again.

By replacing the polynomial $\mathbf{yz} + 1$ by p in the set $S = \{\mathbf{xy} + 1, \mathbf{yz} + 1\}$, we obtain a Gröbner base $G = \{\mathbf{xy} + 1, -\mathbf{x} + \mathbf{z}\}$.

This approach can also be ported to m-tuples of polynomials from $\mathcal{F}[\mathbf{X}]$, which form a polynomial module: $\mathcal{F}[\mathbf{X}]^m = \mathcal{F}[\mathbf{X}] \times \ldots \times \mathcal{F}[\mathbf{X}]$. Within modules, addition can be performed component wise, as well as multiplication with a scalar from $\mathcal{F}[\mathbf{X}]$. Here, the goal is to compute a normal form of a tuple $\in \mathcal{F}[\mathbf{X}]^m$ modulo a polynomial module $\langle 2^{\mathcal{F}[\mathbf{X}]^m} \rangle$.

2.1.5 Finding Gröbner bases with Buchbergers algorithm

The mere existence of a finite generator base for every polynomial ideal does not help in finding unique normal forms of polynomials modulo a given ideal. Thus, it's important to have a way to compute a Gröbner base for the ideal, generated by a set of polynomials.

In order to construct a Gröbner base for an ideal, generated by a set of polynomial generators incorporates generating *syzygy polynomials*. Let $p_1, p_2 \in \mathcal{F}[\mathbf{X}]$ with $a_i = HC(p_i)$ and $s_i \cdot HT(p_i) = \text{lcm}(HT(p_1), HT(p_2))$. Syzygy polynomials $spol()$ for a pair of polynomials are defined as $\text{spol}(p_1, p_2) = a_2 s_1 p_1 - a_1 s_2 p_2$.

We find, that a Gröbner base G for the ideal $\langle S \rangle$ should reduce all S-polynomials of two arbitrary polynomials $\in G$ to zero:

$$\forall g_1, g_2 \in G : \text{spol}(g_1, g_2) \xrightarrow[G]{*} 0$$

Thus, we conclude that we have to add all S-polynomials to a set of polynomials S to obtain a Gröbner base for the ideal generated by S. This is exactly what is performed in Buchbergers algorithm.

Theorem 3 . (Buchberger algorithm [BW93] 5.53) *Let F be a finite subset of $\mathcal{F}[\mathbf{X}]$. Then, algorithm 1 (GROEBNER) computes a Gröbner base G in $\mathcal{F}[\mathbf{X}]$ such that $F \subseteq G$ and $\langle G \rangle = \langle F \rangle$.*

Algorithm 1 GROEBNER

Require: F a finite subset of $\mathcal{F}[\mathbf{X}]$
  $G \leftarrow F$
  $B \leftarrow \{(g_1, g_2) \mid g_1, g_2 \in G \text{ with } g_1 \neq g_2\}$
  while $B \neq 0$ **do**
    select (g_1, g_2) from B
    $B \leftarrow B \setminus \{(g_1, g_2)\}$
    $h \leftarrow \text{spol}(g_1, g_2)$
    $h \xrightarrow[G]{*} h_0$
    if $h_0 \neq 0$ **then**
      $B \leftarrow B \cup \{(g, h_0) \mid g \in G\}$
      $G \leftarrow G \cup \{h_0\}$
    end if
  end while
  return G

Now, if we also keep all generators in a set pairwise irreducible, we arrive at a unique Gröbner base, the *reduced Gröbner base*. Note that there is at least an exponential upper worst case bound of space for computing Gröbner bases given by Mayr and Kuhnel in [KM96]. Quite recently, lower bounds for computing Gröbner base have been shown to be at least exponential in [IPS99].

Considering modules of polynomials, in principle, finding a Gröbner base can be done by introducing m new variables $\mathbf{y}_i$, one for each dimension of the module, and performing standard Gröbner base creation with the Buchberger algorithm. As a matter of fact, there is a version of Gröbner base construction, that performs even better. This is due to the fact, that the variables $\mathbf{y}_i$ will only occur in linear form within the tested polynomials. This implies, that it is only necessary to consider those S-polynomials whose monomials are linear in the variables $\mathbf{y}_i$.

2.1.6 Basic ideal algorithms

One of the fundamental operations on ideals is the intersection of an ideal $I \subseteq \mathcal{F}[\mathbf{X}]$ with the subspace $\mathcal{F}[\mathbf{U}]$, that spans polynomials over a subset of variables $\mathbf{U} \subseteq \mathbf{X}$. The algorithm for generating a Gröbner base for an elimination ideal relies on Gröbner base construction, with an appropriate term order. It is given as follows:

Example 5 . Let $F = \{\mathbf{x}_3^2 + 3\mathbf{x}_2, \mathbf{x}_1 + 3\mathbf{x}_2 - 7\mathbf{x}_4, \mathbf{x}_2 + \mathbf{x}_4\}$ and $\mathbf{U} = \{\mathbf{x}_1, \mathbf{x}_3, \mathbf{x}_4\}$, with the total degree lexicographic term order $\leq$: $\mathbf{x}_4 < \mathbf{x}_3 < \mathbf{x}_2 < \mathbf{x}_1$.

Algorithm 2 ELIMINATION

Require: F a finite subset of $\mathcal{F}[\mathbf{X}]$ and $\mathbf{U} \subseteq \mathbf{X}$
  choose a term order $\leq'$ with $\mathbf{U} \ll' \mathbf{X} \setminus \mathbf{U}$
  $G' \leftarrow$ GROEBNER(F) w.r.t. $\leq'$
  $G \leftarrow G' \cap \mathcal{F}[\mathbf{U}]$
  return G

Choose the lexicographic term order $\leq'$ with $\mathbf{x}_4 <' \mathbf{x}_3 <' \mathbf{x}_1 <' \mathbf{x}_2$, and rewrite the polynomials in F according to this order:

$$F = \{3\mathbf{x}_2 + \mathbf{x}_3^2, 3\mathbf{x}_2 + \mathbf{x}_1 - 7\mathbf{x}_4, \mathbf{x}_2 + \mathbf{x}_4\}$$

G' := *GROEBNER*(F) $w.r.t$ $\leq'$ *will give:*

$$G' = \{\mathbf{x}_2 + \mathbf{x}_4, \mathbf{x}_3^2 - 3\mathbf{x}_4, \mathbf{x}_1 - 10\mathbf{x}_4\}$$

$$G := G' \cap \mathbb{Z}[\mathbf{U}] = \{\mathbf{x}_3^2 - 3\mathbf{x}_4, \mathbf{x}_1 - 10\mathbf{x}_4\}$$

Intuitively, ELIMINATION consists in creating a Gröbner base with all variables, which are desired to be eliminated at the head of the polynomials. From this Gröbner base, only those polynomials, which consist of variables from U exclusively are taken into the Gröbner base G for the resulting ideal.

Performing the intersection of two ideals on their representation as Gröbner bases is also a frequent problem. Let G_1 and G_2 be Gröbner bases for ideals $\in \mathcal{F}[\mathbf{X}]$. First, we create a common ideal I which connects the two ideals $\langle G_1 \rangle$ and $\langle G_2 \rangle$ with the help of an artificial variable $\mathbf{y}$: $I = \langle \{\mathbf{y} \cdot G_1\} \cup \{(1 - \mathbf{y}) \cdot G_2\} \rangle$ Intuitively, for $\mathbf{y} = 0$ this corresponds to $\langle G_1 \rangle$ and for $\mathbf{y} = 1$ this corresponds to $\langle G_2 \rangle$. Now, lets consider $\forall \mathbf{y} \langle \{\mathbf{y} \cdot G_1\} \cup \{(1 - \mathbf{y}) \cdot G_2\} \rangle$, which filters the ideal such that only those polynomial, implied from both generator sets remain in the result set. This can be implemented by means of the ELIMINATION algorithm, and leads to the algorithm INTERSECTION:

Algorithm 3 INTERSECTION

Require: F, G finite subsets of $\mathcal{F}[\mathbf{X}]$
  $H \leftarrow$ ELIMINATION($\{\mathbf{y} \cdot F\} \cup \{(1 - \mathbf{y}) \cdot G\}$,$\mathbf{X}$)
  return H

Example 6 . Let $F = \{\mathbf{x}_3^2 + 3\mathbf{x}_2, \mathbf{x}_1 + 3\mathbf{x}_2 - 7\mathbf{x}_4, \mathbf{x}_2 + \mathbf{x}_4\}$ *and* $G = \{2\mathbf{x}_2^2 - \mathbf{x}_4, \mathbf{x}_3^2 - 3\mathbf{x}_4, \mathbf{x}_1 - 10\mathbf{x}_4\}$.
The Gröbner base, spanning the intersection ideal of $\langle F \rangle$ *and* $\langle G \rangle$ *is given by the elimination ideal w.r.t.* $\mathbf{X}$ *of* $\{\mathbf{x}_3^2 + 3\mathbf{x}_2, \mathbf{x}_1 + 3\mathbf{x}_2 - 7\mathbf{x}_4, \mathbf{x}_2 + \mathbf{x}_4\} \cdot \mathbf{y} \cup \{2\mathbf{x}_2^2 - \mathbf{x}_4, \mathbf{x}_3^2 - 3\mathbf{x}_4, \mathbf{x}_1 - 10\mathbf{x}_4\} \cdot (1 - \mathbf{y})$*:*

$$F \cdot \mathbf{y} \cup G \cdot (1 - \mathbf{y}) = \{\mathbf{x}_3^2\mathbf{y} + 3\mathbf{x}_2\mathbf{y}, \mathbf{x}_1\mathbf{y} + 3\mathbf{x}_2\mathbf{y} - 7\mathbf{x}_4\mathbf{y}, \mathbf{x}_2\mathbf{y} + \mathbf{x}_4\mathbf{y}\}$$

$$\cup$$

$$\{-2\mathbf{x}_2^2\mathbf{y} + \mathbf{x}_4\mathbf{y} + 2\mathbf{x}_2^2 - \mathbf{x}_4, -\mathbf{x}_3^2\mathbf{y} + 3\mathbf{x}_4\mathbf{y} + \mathbf{x}_3^2 - 3\mathbf{x}_4, -\mathbf{x}_1\mathbf{y} + 10\mathbf{x}_4\mathbf{y} + \mathbf{x}_1 - 10\mathbf{x}_4\}$$

The result for the intersection is then yielded by the following elimination ideal:

$$\mathit{ELIMINATION}(F \cdot \mathbf{y} \cup G \cdot (1-\mathbf{y}), \mathbf{X}) = \{\mathbf{x}_3^2 - 3\mathbf{x}_4, \mathbf{x}_1 - 10\mathbf{x}_4\}$$

2.2 Program analysis basics

After this rather basic properties of polynomials in general, we now define our notion of programs and their semantics in a common section. Each particular program analysis is based on this common understanding of programs. We start by describing the structure of programs by means of control flow graphs. We then generically define the concepts of runs of a program and connect runs of programs with their collecting semantics, leaving the concrete interpretation of the program instructions to the particular analyses.

2.2.1 Control flow graphs

We assume that a *Program* is given by a finite collection of procedures, with one special procedure $main$, where program execution is said to start. Such a program operates on the set of global variables $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$. We consider these variables to range over values from some set of values $\mathbb{V}$. Throughout this thesis, we will introduce different concrete domains for analysing different properties of the program. In the next subsection, we give a more detailed model of how values and program semantics are related. The basic statements in our programs are assignments of the form $\mathbf{x}_j := t$, where t is an expression of variables and operators, interpreted in the particular analysis' domain. Furthermore, we consider interprocedural control flow, which is realised via procedure calls $f()$. For a start, we consider procedures to access all variables from $\mathbf{X}$ as global variables. In special sections, we demonstrate, how to extend the particular approaches to a separation of global and local variables, call-by-value parameter passing and return values. Further, programs in general may support guarded or non-deterministic branching. Depending on the kind of the particular analysis, programs may be abstracted by considering guards that cannot be handled by the analysis as non-deterministic branches, i.e. as if both branches would be possible. This abstraction turns out to be quite common as i.e. in [MOS04b] it is pointed out that in presence of equality guards, even exact constant propagation becomes undecidable.

Formally, we define the control flow of a program as a set of control flow graphs. Let stm be the set of assignments, procedure calls and guards. Now, each procedure f is given by a distinct finite edge-labelled *control flow graph* $G_f = (N_f, E_f, \mathsf{st}_f, \mathsf{r}_f)$ that consists of

- a set N_f of *program points*;
- a set of edges $E_f \subseteq N_f \times \mathsf{stm} \times N_f$;
- f's *start point* $\mathsf{st}_f \in N_f$ and
- f's *return point* $\mathsf{r}_f \in N_f$

An example of a program in CFG representation is shown in Fig. 2.1.

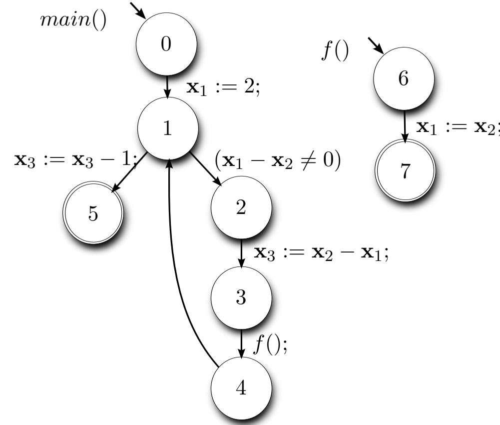


Figure 2.1: An interprocedural program with guards.

We use non-deterministic assignments $\mathbf{x}_j :=?$ for all those instructions with side-effects, that we cannot model. This assignment is equivalent to the non-deterministic choice between all assignments $\mathbf{x}_j := c$ with constants c in the particular variable domain. These non-deterministic assignments can be used to abstract, e.g. input statements which return unknown values or assignments which cannot be interpreted exactly in the domain of the analysis. Assignments $\mathbf{x}_j := \mathbf{x}_j$ have no effect on the program state. They model skip statements as e.g. for the abstraction of guards.

2.2.2 Semantics of programs

Similar to [Gra91, MOS04b, MOS05b], we find it convenient to start our approaches from the *collecting* semantics. With the collecting semantics, we describe for each program point the values which the program variables hold. In the following analysis, we present analyses, based on different kinds of values, e.g. the field $\mathbb{Q}$, the ring $\mathbb{Z}_{2^w}$ or even uninterpreted terms $\mathcal{T}_\Omega$. In this section, we introduce the basics for the concrete semantics, which all these analyses share, keeping the concrete domain $\mathbb{V}$ of the variables and the interpretation $[\![s]\!]_\mathbb{V}$ of the program instructions s generic.

We model a state attained via the execution of our program by a k-dimensional vector $x = [x_1, \ldots, x_k] \in \mathbb{V}^k$ where x_i is the value assigned to the variable $\mathbf{x}_i$. Runs r through the program execute sequences of assignments and guards: $r \equiv r_1; \ldots; r_m$ where each r_i is a statement in form of an assignment or a guard. The execution of a single statement r_i induces a partial transformation $[\![r_i]\!]_\mathbb{V} : \mathbb{V} \mapsto \mathbb{V}$ of a program state. A run, as a sequence of statements, thus also induces a partial transformation. Transformers of particular states can be extended to work as transformers of sets of states $\mathbb{V}^k \mapsto \mathbb{V}^k$ by component wise application:

$$[\![r]\!]_\mathbb{V}(X) = \{[\![r]\!]_\mathbb{V}(x) \mid x \in X\}$$

In order to model procedure calls, we gather their semantics as sets of runs. The runs,

reaching a program point t can be characterised for each program point u by the smallest solution of a constraint system $[\mathbf{R}]$:

$$\begin{array}{lll} [\text{R1}] & \mathbf{R}_t(t) \supseteq \{\varepsilon\} & \\ [\text{R2}] & \mathbf{R}_t(u) \supseteq f_e(\mathbf{R}_t(v))\,, & \text{for each } (u,e,v) \in E \\ [\text{R3}] & \mathbf{R}_t(u) \supseteq \mathbf{R}_{\mathsf{r}_f}(\mathsf{st}_f) \circ \mathbf{R}_t[v] & \text{for each } (u,f(),v) \in E \end{array}$$

Constraint [R1] expresses, that the set of runs reaching program point t when starting from t contains the empty run, denoted by "ε". By [R2], a run starting from u is obtained by considering an outgoing edge (u, e, v) and concatenating the run corresponding to e with a run starting from v, where where $f_e(R) = \{r; t \mid r \in \mathbf{R}(e) \wedge t \in R\}$. If edge $e \equiv (p \neq 0)$ or $e \equiv \mathbf{x}_i := p;$, it gives rise to a single execution: $\mathbf{R}(e) = \{p \neq 0\}$ or $\mathbf{R}(e) = \{\mathbf{x}_i := p\}$. The effect of an edge e annotated by $\mathbf{x}_i$:=? is captured by collecting *all constant* assignments:

$$\mathbf{R}(e) = \{\mathbf{x}_i := c \mid c \in \mathbb{V}\}$$

Constraint [R3] finally expresses, that the runs which pass a procedure call have to be extended by all runs from the procedures start, reaching its return node. The $\circ$ operator here denotes the component wise concatenation of the runs.

Based on this rather syntactical approach to the operational semantics of a program, we define the collecting semantics of a program as the set of program states for each program point $\mathcal{C}_{\mathbb{V}}[t]$, which is valid after all runs from program start which reach that point. In short, this means $\mathcal{C}_{\mathbb{V}}[t] = \{[\![r]\!]_{\mathbb{V}}\mathbb{V}^k \mid r \in \mathbf{R}_t(\mathsf{st}_{main})\}$. Collecting semantics can be characterised as the least solution of a constraint system $[\mathcal{C}_{\mathbb{V}}]$:

$$\begin{array}{llll} [\mathcal{C}1] & \mathcal{C}_{\mathbb{V}}[\mathsf{st}_{main}] & \supseteq \mathbb{V}^k & \text{program starts with } main \\ [\mathcal{C}2] & \mathcal{C}_{\mathbb{V}}[v] & \supseteq [\![s]\!]_{\mathbb{V}}(\mathcal{C}_{\mathbb{V}}[u]) & \text{for each } (u,s,v) \in E \\ [\mathcal{C}3] & \mathcal{C}_{\mathbb{V}}[v] & \supseteq [\![r]\!]_{\mathbb{V}}(\mathcal{C}_{\mathbb{V}}[u]) \mid r \in \mathbf{R}_{\mathsf{r}_f}(\mathsf{st}_f) & \text{for each } (u,f(),v) \in E \\ [\mathcal{C}4] & \mathcal{C}_{\mathbb{V}}[\mathsf{st}_f] & \supseteq \mathcal{C}_{\mathbb{V}}[u] & \text{for each } (u,f(),_) \in E \end{array}$$

For the program start, $[\mathcal{C}1]$ specifies that arbitrary initial variable assignments are assumed. $[\mathcal{C}2]$ interprets the statements annotated at edges in $\mathbb{V}$. Procedure calls consist in executing the runs from a procedures start to its return node, as in $[\mathcal{C}3]$. Interpreting runs in the respective domain $\mathbb{V}$, we obtain $[\![\epsilon]\!] = \mathsf{Id}$ and $[\![r; r_{rest}]\!]_{\mathbb{V}} = [\![r_{rest}]\!]_{\mathbb{V}} \circ_{\mathbb{V}} [\![r]\!]_{\mathbb{V}}$ with $\circ_{\mathbb{V}}$ as the composition of the partial functions. $[\mathcal{C}4]$ finally connects the states of all call sites to a procedure with those of the procedure's start.

2.3 Analysing polynomials in $\mathbb{Q}$

Invariants and intermediate assertions are the key to deductive verification of programs. Correspondingly, techniques for automatically checking and finding invariants and intermediate assertions have been studied (cf., e.g., [BBM97, BLS96, SSM04]). Here, we present analyses that check and find valid polynomial identities in programs. A polynomial identity is a formula $p(\mathbf{x}_1, \ldots, \mathbf{x}_k) = 0$ where $p(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ is a multivariate polynomial in the program variables $\mathbf{x}_1, \ldots, \mathbf{x}_k$.

Looking for valid polynomial identities is a rather general question with many applications. Many classical data flow analysis problems can be seen as problems about polynomial identities. Some examples are: finding *definite equalities among variables* as $\mathbf{x} = \mathbf{y}$; *constant propagation*, i.e., detecting variables or expressions with a constant value at run-time; *discovery of symbolic constants* like $\mathbf{x} = 5\mathbf{y}+2$ or even $\mathbf{x} = \mathbf{y}\mathbf{z}^2+42$; *detection of complex common sub-expressions* where even expressions are sought which are syntactically different but have the same value at run-time such as $\mathbf{x}\mathbf{y} + 42 = \mathbf{y}^2 + 5$; and *discovery of loop induction variables*.

Polynomial identities found by an automatic analysis are also useful for program verification, as they provide non-trivial valid assertions about the program. In particular, loop invariants can be discovered fully automatically. As polynomial identities express quite complex relationships among variables, the discovered assertions may form the backbone of the program proof and thus significantly simplify the verification task.

In the following, we critically review different approaches for determining valid polynomial identities with an emphasis on their precision. In expressions, only addition and multiplication are treated exactly, and, except for guards of the form $p \neq 0$ for polynomials p, conditional choice is generally approximated by non-deterministic choice. These assumptions are crucial for the design of effective exact analyses [MOS02, MOS04b]. Such programs will be called *polynomial* in the sequel.

Much research has been devoted to polynomial programs without procedure calls, i.e., *intraprocedural* analyses. Karr was the first who studied this problem [Kar76]. He considers polynomials of degree at most 1 (*affine* expressions) both in assignments and in assertions and presents an algorithm which, in absence of guards, determines all valid affine identities. This algorithm has been improved by Müller-Olm and Seidl and extended to deal with polynomial identities up to a fixed degree [MOS04b]. Gulwani and Necula also re-considered Karr's analysis problem [GN03] recently. They use randomisation in order to improve the complexity of the analysis at the price of a small probability of finding invalid identities.

The first attempt to generalise Karr's method to *polynomial* assignments is [MOS02] where Müller-Olm and Seidl show that validity of a polynomial identity at a given target program point is decidable for polynomial programs. Later, Rodriguez-Carbonell et al. propose an analysis based on the observation that the set of identities which are valid at a program point can be described by a polynomial *ideal* [RCK04b].

Their analysis is based on a constraint system over polynomial ideals whose greatest solution precisely characterises the set of all valid identities. The problem, however, with this approach is that *descending* chains of polynomial ideals may be infinite implying that

no effective algorithm can be derived from this characterisation. Therefore, they provide special cases [RCK04a] or approximations that allow to infer some valid identities. Opposed to that, the approach by Müller-Olm and Seidl is based on effective weakest precondition computations [MOS02, MOS04a]. They consider assertions to be checked for validity and compute for every program point weakest preconditions which also are represented by ideals. In this case, fixpoint iteration results in *ascending* chains of ideals which are guaranteed to terminate by Hilbert's basis theorem. Therefore, this method provides a decision procedure for validity of polynomial identities. By using a *generic* identity with unknowns instead of coefficients, this method also provides an algorithm for *inferring* all valid polynomial identities up to a given degree [MOS04a].

Both approaches are *intraprocedural* analyses, i.e., cannot handle procedure calls. However, an interprocedural generalisation of Karr's algorithm is given by Müller-Olm and Seidl in [MOS04c]. Using techniques from linear algebra, they succeed in inferring all interprocedurally valid affine identities in programs with affine assignments and no guards. The method easily generalises to inferring also all polynomial identities up to a fixed degree in these programs. A generalisation of the intraprocedural randomised algorithm to programs with procedures is possible as well, as shown by Gulwani and Necula [GN05]. A first attempt to infer polynomial identities in presence of polynomial assignments and procedure calls is provided by Colon [Col04]. His approach is based on ideals of polynomial *transition invariants*. We illustrate, though, the pitfalls of this approach and instead show how the idea of precondition computations can be extended to an interprocedural analysis. In a natural way, the latter analysis also extends the interprocedural analysis from [MOS04c] where only affine assignments such as $\mathbf{x}_2 := 2\mathbf{x}_1 - 3$ are considered.

In this chapter, we give an overview over the common approaches to analysing polynomial equalities in programs with procedure calls. This chapter is organised as follows. We begin with Section 2.3.1 where we instantiate the framework from the general section to the concrete semantics for programs we analyse in this part. Section 2.3.2 provides a precise characterisation of all valid polynomial identities by means of a constraint system. This characterisation is based on forward propagation. Section 2.3.3 provides a second characterisation based on effective weakest precondition computation. This leads to backwards-propagation algorithms. Both Sections 2.3.2 and 2.3.3 consider only programs without procedures. Section 2.3.5 explains an extension to polynomial programs with procedures based on polynomial transition invariants and indicates its limitations. Section 2.3.6 presents a possible extension of the weakest-precondition approach to procedures. Section 2.3.7 then indicates how equality guards can be added to the analyses. Further, we provide an extension of the backward framework to handle local and global variables in Section 2.3.8. Finally, Section 2.3.9 summarises and gives further directions of research.

2.3.1 General setup

We use programs modelled by systems of non-deterministic flow graphs that can recursively call each other as in section 2.2.1. Further, we use the collecting semantics from

section 2.2.2 as reference semantics for our approach, instantiated with the field $\mathbb{Q}$ as domain for program variables. Similar arguments, though, can also be applied in case values are integers from $\mathbb{Z}$. Further, we assume that in the programs we analyse, the assignments to variables are of the form $\mathbf{x}_j := p$ for some polynomial p from $\mathbb{Q}[\mathbf{X}]$, i.e., the ring of all polynomials with coefficients from $\mathbb{Q}$ and variables from $\mathbf{X}$. Note that this restriction does not come by accident. It is well-known [Hec77, RL77] that it is undecidable for non-deterministic flow graphs to determine whether a given variable holds a constant value at a given program point in all executions if the full standard signature of arithmetic operators (addition, subtraction, multiplication, and division) is available. Constancy of a variable is obviously a polynomial identity: $\mathbf{x}$ is a constant at program point n if and only if the polynomial identity $\mathbf{x} - c = 0$ is valid at n for some $c \in \mathbb{Q}$. Clearly, we can write all expressions involving addition, subtraction, and multiplication with polynomials. Thus, if we allow also division, validity of polynomial identities becomes undecidable, as shown in [MOS02],

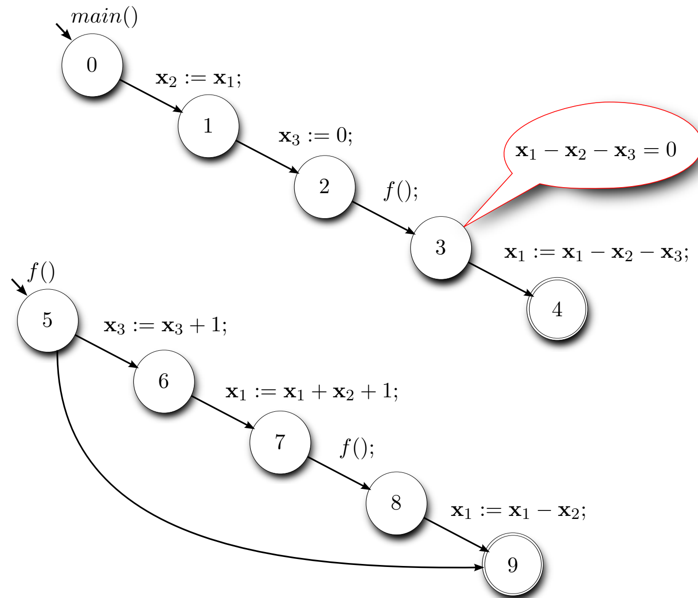


Figure 2.2: An interprocedural program.

Assignments with non-polynomial expressions or input dependent values are therefore assumed to be abstracted with *non-deterministic assignments* $\mathbf{x}_j := c$, $c \in \mathbb{Q}$, as sketched in 2.2.1.

The only form of guards at edges which we can handle precisely within the framework in the following subsections are disequality guards of the form $p \neq 0$ for some polynomial p. For the moment, skip-statements are used to abstract, e.g., equality guards. In Section 2.3.7, we present methods which proximately deal with equality guards. For all other guards, we assume that conditional branching is abstracted with non-deterministic branching for the following sections.

Formally our concrete semantics is thus specified on program states from the set $\mathbb{Q}^k$. We thus instantiate the generic concrete semantics from section 2.2.2 with $\mathbb{Q}$ for $\mathbb{V}$. As the whole section only treats this domain, we may Each run of a single statement or sequence of statements induces a *partial polynomial transformation* of the set of underlying program states $X \subseteq \mathbb{Q}^k$. In the case of a disequality guard ($p \neq 0$) this results in a partial identity function:

$$[\![p \neq 0]\!](X) \;=\; \{x \in \mathbb{Q} \mid p(x) \neq 0\}$$

A polynomial assignment $\mathbf{x}_j := p$ causes the transformation

$$[\![\mathbf{x}_j := p]\!]\,(X) = \{(x_1, \ldots, x_{j-1}, p(x), x_{j+1}, \ldots, x_k) \mid x \in X\}$$

The partial polynomial transformations corresponding to single assignments and disequality guards can be represented by a total transformation and a polynomial: First, the total transformation $\tau = [q_1, \ldots, q_k]$ which applied to a vector x yields $\tau(x) = [q_1(x), \ldots, q_k(x)]$. A partial transformation π now is a tuple $\pi = (q, \tau)$ of a polynomial q and a transformation τ. If $q(x) \neq 0$, then $\pi(x)$ is defined an yields $\tau(x)$. Together, this yields the following representation for the basic instructions:

$$\begin{array}{lcl} [\![\mathbf{x}_j := p]\!] & = & (1, [\mathbf{x}_1, \ldots, \mathbf{x}_{j-1}, p, \mathbf{x}_{j+1}, \ldots, \mathbf{x}_k]) \\ [\![q \neq 0]\!] & = & (q, [\mathbf{x}_1, \ldots, \mathbf{x}_k]) \end{array}$$

The partial transformation $f = [\![r]\!]$ induced by a run r can always be represented by polynomials $q_0, \ldots, q_k \in \mathbb{Q}$ such that $f(X) = \{(q_1(x), \ldots, q_k(x)) \mid x \in X \wedge q_0(x) \neq 0\}$. For the identity transformation induced by the empty path ε the polynomials $1, \mathbf{x}_1, \ldots, \mathbf{x}_k$ would hold. Transformations induced by polynomial assignments or guards can thus be represented in this manner and are closed under composition, similarly to [MOS04a].

In the following subsections, we take up the basic approach of [MOS04c, MOS04b, MOS05a], which consists in constructing a precise abstract interpretations of the collecting semantics.

2.3.2 Forward analysis framework

We establish a forward analysis framework in several steps. First, we introduce our notion of abstract states for the forward analysis. We then show, why these states form a complete lattice. Then, we characterise this abstract semantics by means of a constraint system, giving effective implementations of the transfer functions. At last, we address the problem of infinitely decreasing chains of ideals, and the consequences for the forward analysis.

Let $\pi = (q, \tau)$ be the partial polynomial transformation induced by some program run. Then, a polynomial identity $p = 0$ is said to be *valid* after this run if, for each initial state $x \in \mathbb{Q}^k$, either $q(x) = 0$ – in this case the run is not executable from x – or $q(x) \neq 0$ and $p(\tau(x)) = 0$ – in this case the run is executable from x and the final state computed by the run is $\tau(x)$. A polynomial identity $p \doteq 0$ is said to be valid at a program

point t if it is satisfied by the program states after every run reaching t, i.e. $p(x) = 0$ for $x \in \mathcal{C}[t]$. It is clear, that if $p = 0$ is a valid polynomial relation then also $q \cdot p = 0$ for all $q \in \mathbb{Q}[\mathbf{X}]$ are valid polynomial relations. Also, when $p = 0$ and $q = 0$ are invariant polynomial relations, $p + q = 0$ as well represents an invariant. Generally, all polynomials which evaluate to zero for the same variable values form a *polynomial ideal*, as mentioned in section 2.1.4. Recall that, by Hilbert's basis theorem (c.f. theorem 1), every polynomial ideal $I \subseteq \mathbb{Q}[\mathbf{X}]$ can be finitely represented. In particular, membership is decidable for ideals as well as containment and equality, as described in the previous sections. Therefore, we choose ideals of the polynomials, that are valid at a program point as abstraction of the concrete program states at this point.

Moreover, the set of all ideals $I \subseteq \mathbb{Q}[\mathbf{X}]$ forms a *complete lattice* w.r.t. set inclusion "$\subseteq$" where the least and greatest elements are the zero ideal $\{0\}$ and the complete ring $\mathbb{Q}[\mathbf{X}]$, respectively. The greatest lower bound of a set of ideals is simply given by their intersection while their least upper bound is the ideal *sum*. More precisely, the sum of the ideals I_1 and I_2 is defined by

$$I_1 \oplus I_2 = \{p_1 + p_2 \mid p_1 \in I_1, p_2 \in I_2\}$$

A set of generators for the sum $I_1 \oplus I_2$ is obtained by taking the union of sets of generators for the ideals I_1 and I_2.

For the moment, let us consider *intraprocedural* analysis only, i.e., analysis of programs just consisting of the procedure $main$ and without procedure calls. Such program consist of a single control-flow graph. As an example, consider the program in Fig. 2.3.

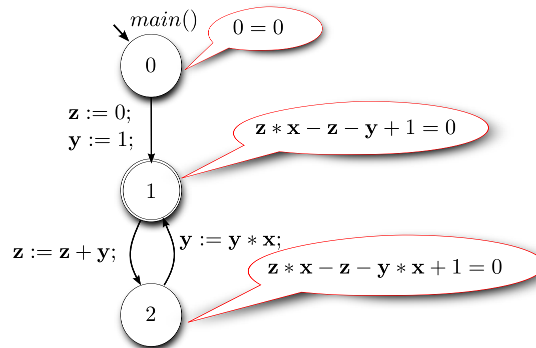


Figure 2.3: A program without procedures.

Given that the set of valid polynomial identities at every program point can be described by polynomial ideals, we can characterise the sets of valid polynomial identities by means of the following constraint system $\mathcal{F}$:

$$
\begin{array}{llll}
[\mathcal{F}1] & \mathcal{F}(\mathsf{st}) & \subseteq \{0\} & \\
[\mathcal{F}2] & \mathcal{F}(v) & \subseteq [\![\mathbf{x}_i := p]\!]^\sharp(\mathcal{F}(u)) & \text{for each } (u, \mathbf{x}_i := p, v) \in E \\
[\mathcal{F}3] & \mathcal{F}(v) & \subseteq [\![\mathbf{x}_i := ?]\!]^\sharp(\mathcal{F}(u)) & \text{for each } (u, \mathbf{x}_i :=?, v) \in E \\
[\mathcal{F}3] & \mathcal{F}(v) & \subseteq [\![p \neq 0]\!]^\sharp(\mathcal{F}(u)) & \text{for each } (u, p \neq 0, v) \in E
\end{array}
$$

where the effects of assignments and disequality guards onto ideals I are given by:

$$
\begin{array}{lcl}
[\![\mathbf{x}_i := p]\!]^\sharp(I) & = & \{q \mid q[p/\mathbf{x}_i] \in I\} \\
[\![\mathbf{x}_i := ?]\!]^\sharp(I) & = & \{\sum_{j=0}^m q_j \mathbf{x}_i^j \mid q_j \in I \cap \mathbb{Q}[\mathbf{X}\backslash\{\mathbf{x}_i\}]\} \\
[\![p \neq 0]\!]^\sharp(I) & = & \{q \mid p \cdot q \in I\}
\end{array}
$$

Intuitively, these definitions can be read as follows. A polynomial identity q is valid after an execution step iff its weakest precondition was valid before the step. For an assignment $\mathbf{x}_i := p$, this weakest precondition equals $q[p/\mathbf{x}_i] = 0$. For a non-deterministic assignment $\mathbf{x}_i := ?$, the weakest precondition of a polynomial $q = \sum_{j=0}^m q_j \mathbf{x}_i^j$ with $q_j \in \mathbb{Q}[\mathbf{X}\backslash\{\mathbf{x}_i\}]$ is given by:

$$\forall\, \mathbf{x}_i.\, q = 0 \quad \equiv \quad q_0 = 0 \wedge \ldots \wedge q_m = 0$$

Finally, for a disequality guard $p \neq 0$, the weakest precondition is given by:

$$\neg(p \neq 0) \vee q = 0 \;\equiv\; p = 0 \vee q = 0 \;\equiv\; p \cdot q = 0$$

Obviously, the operations $[\![\mathbf{x}_i := t]\!]^\sharp$, $[\![\mathbf{x}_i := ?]\!]^\sharp$, and $[\![p \neq 0]\!]^\sharp$ are monotonic. Therefore by the fixpoint theorem of Knaster-Tarski, the constraint system $\mathcal{F}$ has a unique greatest solution over the lattice of ideals of $\mathbb{Q}[\mathbf{X}]$. By definition, the operations commute with arbitrary intersections. Therefore, using standard coincidence theorems for completely distributive intraprocedural data flow frameworks [KU76], we conclude:

Theorem 4 . *Assume* p *is a program without procedures. The greatest solution of the constraint system* $\mathcal{F}$ *for* p *precisely characterises at every program point* v*, the set of all valid polynomial identities.* □

The abstract effect of a disequality guard is readily expressed as an ideal quotient for which effective implementations are well-known. The abstract assignment operations, though, which we have used in the constraint system $\mathcal{F}$ are not very explicit. In order to obtain an effective abstract assignment operation, we intuitively proceed as follows. First, we replace the variable $\mathbf{x}_i$ appearing on the left-hand side of the assignment with a new variable $\mathbf{z}$ both in the ideal I and the right-hand side of the assignment. The variable $\mathbf{z}$ thus represents the value of $\mathbf{x}_i$ *before* the assignment. Then we add the new relationship introduced by the assignment (if there is any) and compute the ideal closure to add all implied polynomial relationships between the variables $\mathbf{X}$ and $\mathbf{z}$. Since the

old value of the overwritten variable is no longer accessible, we keep from the implied identities only those between the variables from $\mathbf{X}$. Formally, we verify in [MOPS06]:

Lemma 1 . *For every ideal $I = \langle p_1, \ldots, p_k \rangle \subseteq \mathbb{Q}[\mathbf{X}]$ and polynomial $p \in \mathbb{Q}[\mathbf{X}]$,*

(i) $\{q \mid q[p/\mathbf{x}_i] \in I\} \;=\; \langle \mathbf{x}_i - s, s_1, \ldots, s_k \rangle \cap \mathbb{Q}[\mathbf{X}]$ *and*

(ii) $\{\sum_{j=0}^{m} q_j \mathbf{x}_i^j \mid q_j \in I \cap \mathbb{Q}[\mathbf{X}\backslash\{\mathbf{x}_i\}]\} \;=\; \langle s_1, \ldots, s_k \rangle \cap \mathbb{Q}[\mathbf{X}]$,

where $s = p[\mathbf{z}/\mathbf{x}_i]$ and $s_j = p_j[\mathbf{z}/\mathbf{x}_i]$ for $i = 1, \ldots, n$.

Note that the only extra operation on ideals we use here is the restriction of an ideal to polynomials with variables from a subset. This operation is effectively implemented in ELIMINATION (c.f. figure 2 in section 2.1.6).

According to Lemma 1, all operations used in the constraint system $\mathcal{F}$ are effective. Nonetheless, this does not in itself provide us with an analysis algorithm. The reason is that the polynomial ring has *infinite decreasing* chains of ideals. And indeed, simple programs can be constructed where fixpoint iteration will not terminate.

Example 7 . Consider our simple example from Fig. 2.3. There, we obtain the ideal for program point 1 as the infinite intersection:

$$\begin{array}{rcl} \mathcal{F}(1) & = & \langle \mathbf{z}, \mathbf{y} - 1 \rangle \;\cap \\ & & \langle \mathbf{z} - 1, \mathbf{y} - \mathbf{x} \rangle \;\cap \\ & & \langle \mathbf{z} - 1 - \mathbf{x}, \mathbf{y} - \mathbf{x}^2 \rangle \;\cap \\ & & \langle \mathbf{z} - 1 - \mathbf{x} - \mathbf{x}^2, \mathbf{y} - \mathbf{x}^3 \rangle \;\cap \\ & & \ldots \end{array}$$

□

Despite infinitely descending chains, the greatest solution of $\mathcal{F}$ has been determined precisely by Rodriguez-Carbonell et al. [RCK04a] — but only for a sub-class of programs. Rodriguez-Carbonell et al. consider simple loops whose bodies consist of a finite non-deterministic choice between sequences of assignments satisfying additional restrictive technical assumptions. No complete methods are known for significantly more general classes of programs. Based on constraint system $\mathcal{F}$, we nonetheless obtain an effective analysis which infers *some* valid polynomial identities by applying *widening* for fixpoint acceleration [CC77]. This idea has been proposed, e.g., by Rodriguez-Carbonell and Kapur [RCK04b] and Colon [Col04]. We will not pursue this idea here. Instead, we propose a different approach, focused on backward analysis as in [MOS02, MOS04a].

2.3.3 Backward analysis framework

The key idea of backward analysis is this: instead of propagating ideals of valid identities in a forward direction, we consider preconditions of runs reaching a particular program point. In detail, we start with a conjectured identity $q \doteq 0$ at some program point t and compute weakest preconditions for this assertion by backwards propagation. The conjecture is proven if and only if the weakest precondition at program entry st_{main} is `true`. More formally, for a given run R, the abstraction let the function β_q

yield the weakest precondition for the validity of an equality $q \doteq 0$ before this run: $\beta_q(R) = \langle\{[\![r]\!]^\top q \doteq 0 \mid r \in R\}\rangle$. The precondition of the validity of $q \doteq 0$ at some point t thus can be reduced to $\beta_q(\mathbf{R}_t(\mathsf{st})) = \langle\{[\![r]\!]^\top q \doteq 0 \mid r \in \mathbf{R}_t(\mathsf{st})\}\rangle$ being `true`. The assertion `true`, i.e., the empty conjunction is uniquely represented by the ideal $\{0\}$. Note that it is decidable whether or not a polynomial ideal equals $\{0\}$.

We will now at first provide weakest precondition transformers which we need to set up a constraint system as abstract interpretation of the sets of runs reaching t. With this we show, that it is decidable, whether a polynomial equality is valid at a program point t. Then, we demonstrate how to infer all equalities up to a fixed degree d by computing the weakest precondition of a generic template polynomial.

Assignments and disequality guards now induce transformations which for every postcondition return the corresponding weakest precondition:

$$
\begin{array}{lcll}
[\![\mathbf{x}_i := p]\!]^\top q & = & \langle q[p/\mathbf{x}_i]\rangle & \\
[\![\mathbf{x}_i := ?]\!]^\top q & = & \langle q_1, \ldots, q_m\rangle & \text{where } q = \sum_{j=0}^{m} q_j \mathbf{x}_i^j \text{ with } q_j \in \mathbb{Q}[\mathbf{X}\backslash\{x_i\}] \\
[\![p \neq 0]\!]^\top q & = & \langle p \cdot q\rangle &
\end{array}
$$

Note that we have represented the disjunction $p = 0 \vee q = 0$ by $p \cdot q = 0$. Also, we have represented conjunctions of equalities by the ideals generated by the respective polynomials. The definitions of our transformers are readily extended to transformers for ideals, i.e., conjunctions of identities. For a given target program point t and conjecture $q \doteq 0$, we therefore can construct a constraint system $[\mathcal{B}]$. The setup for $[\mathcal{B}]$ is a precise abstraction of the constraint system $[\mathbf{R}_t]$ of sets of runs, reaching a program point t with the abstraction function β_q:

$$
\begin{array}{llll}
[\mathcal{B}1] & \mathcal{B}_t(t) & \supseteq \langle q\rangle & \\
[\mathcal{B}2] & \mathcal{B}_t(u) & \supseteq [\![\mathbf{x}_i := p]\!]^\top(\mathcal{B}_t(v)) & \text{for each } (u, \mathbf{x}_i := p, v) \in E \\
[\mathcal{B}3] & \mathcal{B}_t(u) & \supseteq [\![\mathbf{x}_i := ?]\!]^\top(\mathcal{B}_t(v)) & \text{for each } (u, \mathbf{x}_i := ?, v) \in E \\
[\mathcal{B}4] & \mathcal{B}_t(u) & \supseteq [\![p \neq 0]\!]^\top(\mathcal{B}_t(v)) & \text{for each } (u, p \neq 0, v) \in E
\end{array}
$$

Since the basic operations are monotonic, the constraint system $\mathcal{B}$ has a unique least solution in the lattice of ideals of $\mathbb{Q}[\mathbf{X}]$. Consider a single execution path π whose effect is described by the partial polynomial transformation $(q_0, [q_1, \ldots, q_k])$. Then the corresponding weakest precondition is given by:

$$
[\![\pi]\!]^\top p \quad = \quad \langle q_0 \cdot p[q_1/\mathbf{x}_1, \ldots, q_k/\mathbf{x}_k]\rangle
$$

The weakest precondition of p w.r.t. a set of execution paths can be described by the ideal generated by the weakest preconditions for every execution path in the set separately. Since the basic operations in the constraint system $\mathcal{B}$ commute with arbitrary least upper bounds, we once more apply standard coincidence theorems to conclude:

Theorem 5 . *Assume p is a polynomial program without procedures and t is a program point of p. Assume the least solution of the constraint system $\mathcal{B}$ for a conjecture $q \doteq 0$ at t assigns the ideal I to program point* st. *Then, $q \doteq 0$ is valid at t iff $I = \{0\}$.* □

Using a representation of ideals through finite sets of generators, the applications of weakest precondition transformers for edges can be effectively computed. A computation of the least solution of the constraint system $\mathcal{B}$ by standard fixpoint iteration leads to ascending chains of ideals. Therefore, in order to obtain an effective algorithm, we only must assure that *ascending* chains of ideals are ultimately stable. Due to Hilbert's basis theorem, this property indeed holds in polynomial rings over fields (as well as over integral domains like $\mathbb{Z}_{2^w}$). Therefore, the fixpoint characterisation of Theorem 5 gives us an effective procedure for deciding whether or not a conjectured polynomial identity is valid at some program point of a polynomial program.

Corollary 1 . *In a polynomial program without procedures, it can effectively be checked whether or not a polynomial identity is valid at some target point.* □

Example 8 . Consider our example program from Fig. 2.3. If we want to check the conjecture $\mathbf{z} \cdot \mathbf{x} - \mathbf{z} - \mathbf{y} + 1 = 0$ *for program point 1, we obtain:*

$$\begin{array}{rcl} \mathcal{B}(2) & \supseteq & \langle (\mathbf{z} \cdot \mathbf{x} - \mathbf{z} - \mathbf{y} + 1)[\mathbf{y} \cdot \mathbf{x}/\mathbf{y}] \rangle \\ & = & \langle \mathbf{z} \cdot \mathbf{x} - \mathbf{z} - \mathbf{y} \cdot \mathbf{x} + 1 \rangle \end{array}$$

Since,

$$(\mathbf{z} \cdot \mathbf{x} - \mathbf{z} - \mathbf{y} \cdot \mathbf{x} + 1)[\mathbf{z} + \mathbf{y}/\mathbf{z}] \;=\; \mathbf{z} \cdot \mathbf{x} - \mathbf{z} - \mathbf{y} + 1$$

the fixpoint is already reached for program points 1 *and* 2*. Thus,*

$$\begin{array}{rcl} \mathcal{B}(1) & = & \langle \mathbf{z} \cdot \mathbf{x} - \mathbf{z} - \mathbf{y} + 1 \rangle \\ \mathcal{B}(2) & = & \langle \mathbf{z} \cdot \mathbf{x} - \mathbf{z} - \mathbf{y} \cdot \mathbf{x} + 1 \rangle \end{array}$$

Moreover,

$$\begin{array}{rcl} \mathcal{B}(0) & = & \langle (\mathbf{z} \cdot \mathbf{x} - \mathbf{z} - \mathbf{y} + 1)[0/\mathbf{z}, 1/\mathbf{y}] \rangle \\ & = & \langle 0 \rangle \;=\; \{0\} \end{array}$$

Therefore, the conjecture is proved. □

It seems that the algorithm of testing whether a certain given polynomial identity $p_0 = 0$ is valid at some program point contains no clue on how to infer so far unknown valid polynomial identities. This, however, is not quite true. We show now how to determine all polynomial identities of some arbitrary given form that are valid at a given program point of interest. The form of a polynomial is given by a selection of monomials that may occur in the polynomial.

Let $D \subseteq \mathbb{N}_0^k$ be a finite set of exponent tuples for the variables $x_1, \ldots, x_k$. Then a polynomial q is called a D-polynomial if it contains only monomials $b \cdot \mathbf{x}_1^{i_1} \cdot \ldots \cdot \mathbf{x}_k^{i_k}$, $b \in \mathbb{Q}$, with $(i_1, \ldots, i_k) \in D$, i.e., if it can be written as

$$q = \sum_{\sigma = (i_k, \ldots, i_k) \in D} a_\sigma \cdot \mathbf{x}_1^{i_1} \cdot \ldots \cdot \mathbf{x}_k^{i_k}$$

If, for instance, we choose $D = \{(i_1, \ldots, i_k) \mid i_1 + \ldots + i_k \leq d\}$ for a fixed maximal degree $d \in \mathbb{N}$, then the D-polynomials are all the polynomials up to degree d. Here the

degree of a polynomial is the maximal degree of a monomial occurring in q where the degree of a monomial $b \cdot \mathbf{x}_1^{i_1} \cdot \ldots \cdot \mathbf{x}_k^{i_k}$, $b \in \mathbb{Q}$, equals $i_1 + \ldots + i_k$.

We introduce a new set of variables $\mathbf{A}_D$ given by:

$$\mathbf{A}_D = \{\mathbf{a}_\sigma \mid \sigma \in D\} .$$

Then we introduce the *generic* D-polynomial as

$$q_D = \sum_{\sigma=(i_k,\ldots,i_k)\in D} \mathbf{a}_\sigma \cdot \mathbf{x}_1^{i_1} \cdot \ldots \cdot \mathbf{x}_k^{i_k} .$$

The polynomial q_D is an element of the polynomial ring $\mathbb{Q}[\mathbf{X} \cup \mathbf{A}_D]$. Note that every concrete D-polynomial $q \in \mathbb{Q}[\mathbf{X}]$ can be obtained from the generic D-polynomial q_D simply by substituting concrete values $a_\sigma \in \mathbb{Q}$, $\sigma \in D$, for the variables $\mathbf{a}_\sigma$. If $a : \sigma \mapsto a_\sigma$ and $\mathbf{a} : \sigma \mapsto \mathbf{a}_\sigma$, we write $q_D[a/\mathbf{a}]$ for this substitution.

Instead of computing the weakest precondition of each D-polynomial q separately, we may compute the weakest precondition of the single generic polynomial q_D once and for all and substitute the concrete coefficients a_σ of the polynomials q into the precondition of q_D later. Indeed, [MOS04a] yields:

Theorem 6 . *Assume p is a polynomial program without procedures and let $\mathcal{B}_D(v)$, v program point of p, be the least solution of the constraint system $\mathcal{B}$ for p with conjecture q_D at target t. Then a polynomial $q = q_D[a/\mathbf{a}]$ is valid at t iff $q'[a/\mathbf{a}] = 0$ for all $q' \in \mathcal{B}_D(\mathsf{st})$.* □

Clearly, it suffices that $q'[a/\mathbf{a}] = 0$ only for a set of generators of $\mathcal{B}_D(\mathsf{st})$. Still, this does not immediately give us an effective method of determining all suitable coefficient vectors, since the precise set of solutions of arbitrary polynomial equation systems are not computable. We observe, however, in [MOS04a]:

Lemma 2 . *Every ideal $\mathcal{B}_D(u)$, u a program point, of the least solution of the abstract constraint system $\mathcal{B}$ for conjecture q_D at some target node t is generated by a finite set G of polynomials q where each variable $\mathbf{a}_\sigma$ occurs only with degree at most 1. Moreover, such a generator set can be effectively computed.* □

Thus, the set of (coefficient maps) of D-polynomials which are valid at our target program point t can be characterised as the set of solutions of a *linear* equation system. Such equation systems can be algorithmically solved, i.e., finite representations of their sets of solutions can be constructed explicitly, e.g., by Gaussian elimination. We conclude:

Theorem 7 . *For a polynomial program p without procedures and a program point t in p, the set of all D-polynomials which are valid at t can be effectively computed.* □

As a side remark, we should mention that instead of working with the larger polynomial ring $\mathbb{Q}[\mathbf{X} \cup \mathbf{A}_D]$, we could work with *modules* over the polynomial ring $\mathbb{Q}[\mathbf{X}]$ consisting of vectors of polynomials whose entries are indexed with $\sigma \in D$. The operations on modules turn out to be practically much faster than corresponding operations

on the larger polynomial ring itself, see [Pet04] for a practical implementation and preliminary experimental results.

Example 9 . Again consider our program from Fig. 2.3 on page 18. We are interested in invariants at program point 1. *We choose the set* $D = \{(1,0,1),(1,0,0),(0,1,0),(0,0,1),(0,0,0)\}$ *as our set of exponents. The generic polynomial then is* $p_D = a_0\mathbf{xz} + a_1\mathbf{x} + a_2\mathbf{y} + a_3\mathbf{z} + a_4$.
We now obtain for program point 2*:*

$$\mathcal{B}_D(2) \supseteq \langle (a_0\mathbf{xz} + a_1\mathbf{x} + a_2\mathbf{y} + a_3\mathbf{z} + a_4)[\mathbf{y}\cdot\mathbf{x}/\mathbf{y}] \rangle$$
$$= \langle a_0\mathbf{xz} + a_2\mathbf{xy} + a_1\mathbf{x} + a_3\mathbf{z} + a_4 \rangle$$

Since $\langle (a_0\mathbf{xz} + a_2\mathbf{xy} + a_1\mathbf{x} + a_3\mathbf{z} + a_4)[\mathbf{z}+\mathbf{y}/\mathbf{z}] \rangle$

$$= \langle (a_0 + a_2)\mathbf{xy} + a_0\mathbf{xz} + a_1\mathbf{x} + a_3\mathbf{y} + a_3\mathbf{z} + a_4 \rangle$$

we obtain

$$\mathcal{B}_D(1) \supseteq \langle (a_0 + a_2)\mathbf{xy} + (a_3 - a_2)\mathbf{y}\ ,\ a_0\mathbf{xz} + a_1\mathbf{x} + a_2\mathbf{y} + a_3\mathbf{z} + a_4 \rangle$$

which stabilises in the next round of iterations. Finally, we get

$$\mathcal{B}_D(0) = \langle (a_0 + a_2)\mathbf{xy} + (a_3 - a_2)\mathbf{y}\ ,\ a_0\mathbf{xz} + a_1\mathbf{x} + a_2\mathbf{y} + a_3\mathbf{z} + a_4 \rangle [0/\mathbf{z}, 1/\mathbf{y}]$$
$$= \langle (a_0 + a_2)\mathbf{x} + (a_3 - a_2)\ ,\ a_1\mathbf{x} + a_2 + a_4 \rangle$$

The only way for $\mathcal{B}_D(0) \subseteq \{0\}$ *is under the following constraints:*

$$a_0 = -a_2, a_2 = a_3, a_1 = 0, a_2 = -a_4$$

This means for $p_D[0/a_1, -a_0/a_2, -a_0/a_3, a_0/a_4] = a_0(\mathbf{xz} - \mathbf{y} - \mathbf{z} + 1)$. *We have thus found an invariant for program point* 0*, namely* $\mathbf{xz} - \mathbf{y} - \mathbf{z} + 1$*!* □

2.3.4 Optimistic procedure effect tabulation

Trying to extend the systems $[\mathcal{F}]$ and $[\mathcal{B}]$ to handle procedure calls turns out not to be easy: For this, we have to find a concise representation of procedure effects. One of the simplest representations of procedure effects as a function is to tabulate the effects for fixed inputs to the procedure calls. This proceeding can be seen as a variant of *inlining* a procedure's body into the caller's body. This approach suites best for cases, in which the input for which to tabulate the procedure's effect is finite. In case, that the input is not finite, one may stick to *optimistic tabulation* – in this approach, we limit the number of tabulated mappings to a threshold. We tabulate the inputs until we reach this limit, and for all other inputs provide a generic effect, consisting in a safe worst-case treatment, i.e. dropping all information, which may be altered by the call. This leads to an exact procedure call treatment for a bounded number of call contexts at least.
Example 10 . Consider the program from Fig. 2.4.

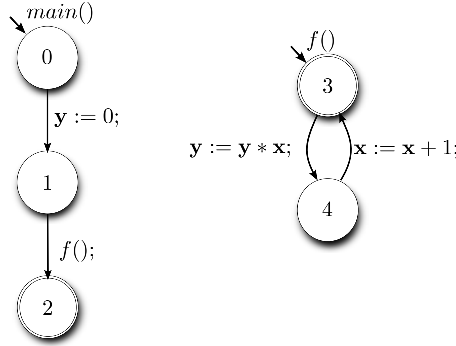


Figure 2.4: A simple program with procedures.

We compute:

$$\mathcal{F}(1) = \langle \mathbf{x} - \mathbf{x}', \mathbf{y} \rangle$$

We now tabulate the effect of `f()` *for the polynomial ideal* $\langle \mathbf{y} \rangle$*:*

$$\begin{array}{rcl} \mathcal{F}(f, \langle \mathbf{y} \rangle) & = & \langle \mathbf{y} \rangle \cap \\ & & \langle \mathbf{x} - \mathbf{x}' - 1, \mathbf{y} \rangle \cap \\ & & \langle \mathbf{x} - \mathbf{x}' - 2, \mathbf{y} \rangle \cap \\ & & \ldots \\ & = & \langle \mathbf{y} \rangle \end{array}$$

□

Of course, this approach does not scale as it is dependent on the number of calling contexts, which e.g. for recursive calls is unbounded. So we proceed to other alternatives for effective representations of procedure calls.

2.3.5 Interprocedural analysis with transition invariants

The main question of precise interprocedural analysis is this: how can the effects of procedure calls be finitely described? An interesting idea (essentially) due to Colon [Col04] is to represent effects by polynomial *transition invariants*. This means that we introduce a separate copy $\mathbf{X}' = \{\mathbf{x}'_1, \ldots, \mathbf{x}'_k\}$ of variables denoting the values of variables before the execution. Then we use polynomials to express possible relationships between pre- and post-states of the execution. Obviously, all such valid relationships again form an ideal, now in the polynomial ring $\mathbb{Q}[\mathbf{X} \cup \mathbf{X}']$.

The transformation ideals for assignments, non-deterministic assignments and disequality guards are readily expressed by:

$$\begin{array}{lcl} [\![\mathbf{x}_i := p]\!]^{\sharp\sharp} & = & \langle \{\mathbf{x}_j - \mathbf{x}'_j \mid j \neq i\} \cup \{\mathbf{x}_i - p[\mathbf{x}'/\mathbf{x}]\} \rangle \\ [\![\mathbf{x}_i := ?]\!]^{\sharp\sharp} & = & \langle \{\mathbf{x}_j - \mathbf{x}'_j \mid j \neq i\} \rangle \\ [\![p \neq 0]\!]^{\sharp\sharp} & = & \langle \{p[\mathbf{x}'/\mathbf{x}] \cdot (\mathbf{x}_j - \mathbf{x}'_j) \mid j = 1, \ldots, k\} \rangle \end{array}$$

In particular, the last definition means that either the guard is wrong before the transition or the states before and after the transition are equal. The basic effects can be composed to obtain the effects of larger program fragments by means of a composition operation "$\circ$". Composition on transition invariants can be defined by:

$$I_1 \circ I_2 \quad = \quad (I_1[\mathbf{y}/\mathbf{x}'] \oplus I_2[\mathbf{y}/\mathbf{x}]) \cap \mathbb{Q}[\mathbf{X} \cup \mathbf{X}']$$

where a fresh variable set $\mathbf{Y} = \{\mathbf{y}_1, \ldots, \mathbf{y}_k\}$ is used to store the intermediate values between the two transitions represented by I_1 and I_2 and the postfix operator $[\mathbf{y}/\mathbf{x}]$ denotes renaming of variables in $\mathbf{X}$ with their corresponding copies in $\mathbf{Y}$. Note that "$\circ$" is defined by means of well-known effective ideal operations. Using this operation, we can put up a constraint system $\mathcal{T}$ for ideals of polynomial transition invariants of procedures:

$[\mathcal{T}1]$	$\mathcal{T}(\mathsf{st}_f)$	$\subseteq \langle \mathbf{x}_i - \mathbf{x}'_i \mid i = 1, \ldots, k \rangle$	at entry points
$[\mathcal{T}2]$	$\mathcal{T}(v)$	$\subseteq [\![\mathbf{x}_i := p]\!]^{\sharp\sharp} \circ \mathcal{T}(u)$	for each $(u, \mathbf{x}_i := p, v) \in E$
$[\mathcal{T}3]$	$\mathcal{T}(v)$	$\subseteq [\![\mathbf{x}_i :=?]\!]^{\sharp\sharp} \circ \mathcal{T}(u)$	for each $(u, \mathbf{x}_i :=?, v) \in E$
$[\mathcal{T}4]$	$\mathcal{T}(v)$	$\subseteq [\![p \neq 0]\!]^{\sharp\sharp} \circ \mathcal{T}(u)$	for each $(u, (p \neq 0), v) \in E$
$[\mathcal{T}5]$	$\mathcal{T}(v)$	$\subseteq \mathcal{T}(f) \circ \mathcal{T}(u)$	for each $(u, f(), v) \in E$
$[\mathcal{T}6]$	$\mathcal{T}(f)$	$\subseteq \mathcal{T}(\mathsf{r}_f)$	v exit point of f

Example 11 . Consider the program from Fig. 2.5. We calculate:

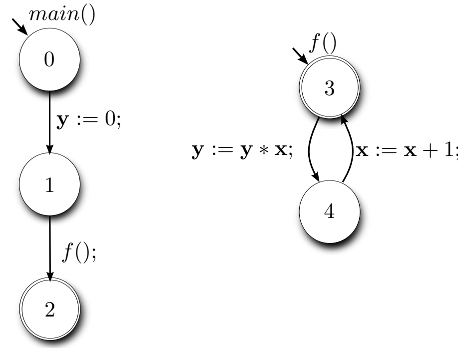


Figure 2.5: A simple program with procedures.

$$\begin{aligned}\mathcal{T}(f) \quad = \quad & \langle \mathbf{x} - \mathbf{x}', \mathbf{y} - \mathbf{y}' \rangle \ \cap \\ & \langle \mathbf{x} - \mathbf{x}' - 1, \mathbf{y} - \mathbf{y}' \cdot \mathbf{x}' \rangle \ \cap \\ & \langle \mathbf{x} - \mathbf{x}' - 2, \mathbf{y} - \mathbf{y}' \cdot \mathbf{x}' \cdot (\mathbf{x}' + 1) \rangle \ \cap \\ & \ldots \\ = \quad & \langle 0 \rangle\end{aligned}$$

Using this invariant for analysing the procedure main, *we only find the trivial transition invariant* 0. *On the other hand, we may instead* inline *the procedure* f *as in Fig. 2.6. A*

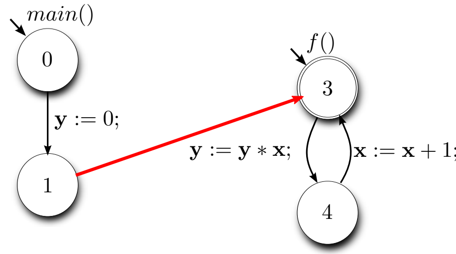


Figure 2.6: The inlined version of the example program.

corresponding calculation of the transition invariant of main *yields:*

$$\begin{aligned}\mathcal{T}(\texttt{main}) &= \langle \mathbf{x}-\mathbf{x}', \mathbf{y}\rangle \;\cap \\ &\quad\; \langle \mathbf{x}-\mathbf{x}'-1, \mathbf{y}\rangle \;\cap \\ &\quad\; \langle \mathbf{x}-\mathbf{x}'-2, \mathbf{y}\rangle \;\cap \\ &\quad\; \ldots \\ &= \langle \mathbf{y}\rangle\end{aligned}$$

Thus, for this analysis, inlining may gain precision. □

Clearly, using transition invariants incurs the same problem as forward propagation for intraprocedural analysis, namely, that fixpoint iteration may result in infinite decreasing chains of ideals. Our minimal example exhibited two more problems, namely that the composition operation is not *continuous*, i.e., does not commute with greatest lower bounds of descending chains in the second argument, and also that a less compositional analysis through inlining may infer more valid transition invariants.

It should be noted that Colon did not propose to use *ideals* for representing transition invariants. Colon instead considered *pseudo*-ideals, i.e., ideals where polynomials are considered only up to a given degree bound. This kind of further abstraction solves the problems of infinite decreasing chains as well as missing continuity — at the expense, though, of further loss in precision. Colon's approach, for example, fails to find a nontrivial invariant in the example program from Fig. 2.5 for $main$.

2.3.6 Interprocedural analysis with WP transformers

Due to the apparent weaknesses of the approach through polynomials as transition invariants, we propose to represent effects of procedures by pre-conditions of generic polynomials. Procedure calls are then dealt with through instantiation of generic coefficients. Thus, effects are still described by ideals — over a larger set of variables (or by modules; see the discussion at the end of Section 2.3.3). Suppose we have chosen some finite set $D \subseteq \mathbb{N}_0^k$ of exponent tuples and assume that the polynomial $p = p_D[a/\mathbf{a}]$ is the D-polynomial that is obtained from the generic D-polynomial through instantiation of the generic coefficients with a. Assume further that the effect of some procedure call is

given by the ideal $I \subseteq \mathbb{Q}[\mathbf{X} \cup \mathbf{A}_D] = \langle q_1, \ldots, q_m \rangle$. Then we determine a precondition of $p = 0$ w.r.t. to the call by:

$$I\ (p) \ = \ \langle q_1[a/\mathbf{a}], \ldots, q_m[a/\mathbf{a}] \rangle$$

This definition is readily extended to ideals I' generated by D-polynomials. There is no guarantee, though, that all ideals that occur at the target program points v of call edges $(u, f(), v)$ will be generated by D-polynomials. In fact, there is a simple example where no uniform set D of exponent tuples can be given:
Example 12 . Consider the program from Fig. 2.7.

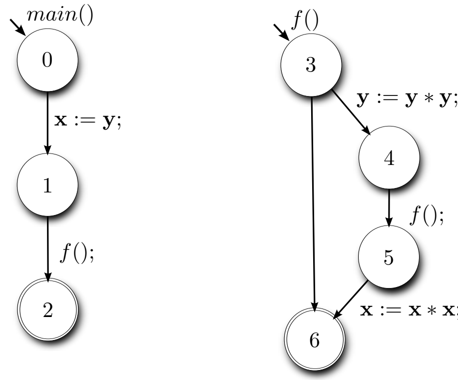


Figure 2.7: A recursive program

Let d be the largest degree with which $\mathbf{x}$ occurs in the set of terms fixed by D. When computing the set $\mathcal{E}(f)$, we start with the generic polynomial p_D. $\mathbf{x}$'s degree in p_D is d. Now, evaluating the dataflow-value for $\mathcal{E}(5)$, we obtain a new polynomial from p_D via substitution of $\mathbf{x}$ by $\mathbf{x}^2$. This new polynomial now has a term, with a degree $2d$ for $\mathbf{x}$ – which is not a D-polynomial. □

Therefore, we additionally propose to use an abstraction operator $\mathbf{W}$ that splits polynomials appearing as post-condition of procedure calls which are not D-polynomials.

We choose a maximal degree d_j for each program variable x_j and let

$$D = \{(i_1, \ldots, i_k) \mid i_j \leq d_j \text{ for } i = 1, \ldots, k\}$$

The abstraction operator $\mathbf{W}$ takes generators of an ideal I and maps them to generators of a (possibly) larger ideal $\mathbf{W}(I)$ which is generated by D-polynomials. In order to construct such an ideal, we need a heuristics which decomposes an arbitrary polynomial q into a linear combination of D-polynomials $q_1, \ldots, q_m$:

$$q = r_1 q_1 + \ldots + r_m q_m \tag{2.1}$$

We could, for example, decompose q according to the first variable:

$$q = q'_0 + \mathbf{x}_1^{d_1+1} \cdot q'_1 + \ldots + \mathbf{x}_1^{s(d_1+1)} \cdot q'_s$$

where each q'_i contains powers of $\mathbf{x}_1$ only up to degree d_1 and repeat this decomposition with the polynomials q'_i for the remaining variables. We can replace every generator of I by D-polynomials in order to obtain an ideal $\mathbf{W}(I)$ with the desired properties:

Lemma 3 . *Let t be an arbitrary term from $\mathbb{Q}[\mathbf{X}]$ and the function $\mathbf{W} : 2^{\mathbb{Q}[\mathbf{X}]} \mapsto 2^{\mathbb{Q}[\mathbf{X}]}$ be a decomposition function. Let $\mathbf{W}(\langle q_1, \ldots, q_m \rangle) = \langle \bigcup_{i=1}^{m} \{q'_{i_0}, \ldots, q'_{i_n}\} \rangle$ with $q_i = q'_{i_0} + tq'_{i_1} + \ldots + t^n q'_{i_n}$, where each q'_{i_j} is free of any power of t. Then*

$$\langle I \rangle \subseteq \langle \mathbf{W}(I) \rangle$$

Proof. We prove this statement via contradiction. Let us assume, that q is a polynomial for which $q \in \langle I \rangle$ but $q \notin \langle \mathbf{W}(I) \rangle$. However, we defined q to be decomposable into $q'_0 + tq'_1 + \ldots + t^n q'_n$, where each of the $q'_i \in \langle \mathbf{W}(I) \rangle$. By the definition of polynomial ideals, q must also be an element of $\langle \mathbf{W}(I) \rangle$. □

Example 13 . Let $p = a + b\mathbf{x}_0 + c\mathbf{x}_2 + d\mathbf{x}_1 + e\mathbf{x}_3$ be the template polynomial and $q = 5a\mathbf{x}_0 + 2c\mathbf{x}_2 + 3d\mathbf{x}_2^2 - a\mathbf{x}_2^2\mathbf{x}_1$.
Applying $\mathbf{W}(\{q\})$, we perform a decomposition of q choosing $t = \mathbf{x}_2^2$ as term to decompose with:

$$q = 5a\mathbf{x}_0 + 2c\mathbf{x}_2 + (3d - a\mathbf{x}_1)\mathbf{x}_2^2$$

This lead to a decomposition with two factors, providing a new set of polynomials, whose generated ideal contains q:

$$\mathbf{W}(\{q\}) = \{5a\mathbf{x}_0 + 2c\mathbf{x}_2,\ 3d - a\mathbf{x}_1\}$$

Each of the coefficients of these polynomials now can be matched with those of the template p.

We use the new application operator as well as the abstraction operator $\mathbf{W}$ to generalise our constraint system $\mathcal{B}$ to a constraint system $\mathcal{E}$ for the effects of procedures:

$$\begin{array}{lll}
[\mathcal{E}1] & \mathcal{E}(\mathsf{r}_f) \supseteq \langle q_D \rangle & \text{at is exit points} \\
[\mathcal{E}2] & \mathcal{E}(u) \supseteq [\![\mathbf{x}_i := p]\!]^\mathsf{T}(\mathcal{E}(v)) & \text{for each } (u, \mathbf{x}_i := p, v) \in E \\
[\mathcal{E}3] & \mathcal{E}(u) \supseteq [\![\mathbf{x}_i :=?]\!]^\mathsf{T}(\mathcal{E}(v)) & \text{for each } (u, \mathbf{x}_i :=?, v) \in E \\
[\mathcal{E}4] & \mathcal{E}(u) \supseteq [\![p \neq 0]\!]^\mathsf{T}(\mathcal{E}(v)) & \text{for each } (u, (p \neq 0), v) \in E \\
[\mathcal{E}5] & \mathcal{E}(u) \supseteq \mathcal{E}(f)(\mathbf{W}(\mathcal{E}(v))) & \text{for each } (u, f(), v) \in E \\
[\mathcal{E}6] & \mathcal{E}(f) \supseteq \mathcal{E}(\mathsf{st}_f) & \text{at entry points}
\end{array}$$

Example 14 . Consider again the example program from Fig. 2.5. Let us choose $d = 1$ where $p_1 = \mathbf{ay} + \mathbf{bx} + \mathbf{c}$. Then we calculate for f:

$$\begin{array}{rcl}
\mathcal{E}(f) & = & \langle \mathbf{ay} + \mathbf{bx} + \mathbf{c} \rangle \oplus \\
& & \langle \mathbf{ayx} + \mathbf{b}(\mathbf{x}+1) + \mathbf{c} \rangle \oplus \\
& & \langle \mathbf{ayx}(\mathbf{x}+1) + \mathbf{b}(\mathbf{x}+2) + \mathbf{c} \rangle \oplus \\
& & \langle \mathbf{ayx}(\mathbf{x}+1)(\mathbf{x}+2) + \mathbf{b}(\mathbf{x}+3) + \mathbf{c} \rangle \oplus \\
& & \ldots \\
& = & \langle \mathbf{ay}, \mathbf{b}, \mathbf{c} \rangle
\end{array}$$

This description tells us that for a linear identity $\mathbf{a}\mathbf{y} + \mathbf{b}\mathbf{x} + \mathbf{c} = 0$ *to be valid after a call to* f*, the coefficients* $\mathbf{b}$ *and* $\mathbf{c}$ *necessarily must be equal to 0. Moreover, either coefficient* $\mathbf{a}$ *equals 0 (implying that the whole identity is trivial) or* $\mathbf{y} = 0$*. Indeed, this is the optimal description of the behaviour of* f *with polynomials.* □

The effects of procedures as approximated by constraint system $\mathcal{E}$ can be used to check a polynomial conjecture $q \doteq 0$ at a given target node t along the lines of constraint system $\mathcal{B}$. We only have to extend it by extra constraints dealing with function calls. Thus, we put up the following constraint system:

$$
\begin{array}{llll}
[\mathrm{R1}^\sharp] & \mathbf{R}_t^\sharp(t) & \supseteq \langle q \rangle & \text{the conjecture at } t \\
[\mathrm{R2}^\sharp] & \mathbf{R}_t^\sharp(u) & \supseteq [\![\mathbf{x}_i := p]\!]^\top(\mathbf{R}_t^\sharp(v)) & \text{for each } (u, \mathbf{x}_i := p, v) \in E \\
[\mathrm{R3}^\sharp] & \mathbf{R}_t^\sharp(u) & \supseteq [\![\mathbf{x}_i :=?]\!]^\top(\mathbf{R}_t^\sharp(v)) & \text{for each } (u, \mathbf{x}_i :=?, v) \in E \\
[\mathrm{R4}^\sharp] & \mathbf{R}_t^\sharp(u) & \supseteq [\![p \neq 0]\!]^\top(\mathbf{R}_t^\sharp(v)) & \text{for each } (u, (p \neq 0), v) \in E \\
[\mathrm{R5}^\sharp] & \mathbf{R}_t^\sharp(u) & \supseteq \mathcal{E}(f)(\mathbf{W}(\mathbf{R}_t^\sharp(v))) & \text{for each } (u, f(), v) \in E \\
[\mathrm{R6}^\sharp] & \mathbf{R}_t^\sharp(f) & \supseteq \mathbf{R}_t^\sharp(\mathsf{st}_f) & \text{for each entry point} \\
[\mathrm{R7}^\sharp] & \mathbf{R}_t^\sharp(u) & \supseteq \mathbf{R}_t^\sharp(f) & \text{for each } (u, f(), _) \in E
\end{array}
$$

This constraint system again has a least solution which can be computed by standard fixpoint iteration. Summarising, we obtain the following theorem:

Theorem 8 . *Assume* p *is a polynomial program with procedures. Assume further that we assert a conjecture* $q \doteq 0$ *at program point* t*.*

Safety: *(i) For every procedure* f*, the ideal* $\mathcal{E}(f)$ *represents a precondition of the identity* $p_D \doteq 0$ *after the call.*

(ii) If the ideal $\mathbf{R}_t^\sharp(\mathsf{st}_{main})$ *equals* $\{0\}$*, then the conjecture* $q \doteq 0$ *is valid at* t*.*

Completeness: *If during fixpoint computation, all ideals at target program points* v *of call edges* $(u, f(), v)$ *are represented by D-polynomials as generators, the conjecture is valid only if the ideal* $\mathbf{R}_t^\sharp(\mathsf{st}_{main})$ *equals* $\{0\}$*.*

.

The safety-part of Theorem 8 tells us that our analysis will never assure a wrong conjecture but may fail to certify a conjecture although it is valid. According to the completeness-part, however, the analysis algorithm provides slightly more information: if no approximation steps are necessary at procedure calls, the analysis is precise. For simplicity, we have formulated Theorem 8 in such a way that it only speaks about checking conjectures. In order to infer valid polynomial identities up to a specified degree bound, we again can proceed analogous to the intraprocedural case by considering a generic postcondition in constraint system $\mathbf{R}_t^\sharp$.

2.3.7 Equality guards

In this section, we discuss methods for dealing with equality guards $(p = 0)$. Recall, that in presence of equality guards, the question whether a variable is constantly 0 at

a program point or not is undecidable even in absence of procedures and with affine assignments only. Thus, we cannot hope for complete methods here. Still, in practical contexts, equality guards are a major source of information about values of variables. Consider, e.g., the control flow graph from Fig. 2.8. Then, according to the equality

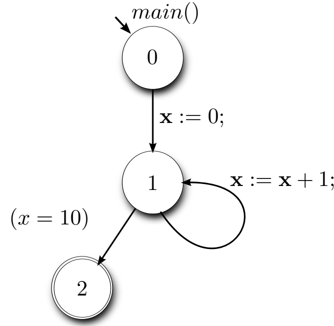


Figure 2.8: A simple for-loop.

guard, we definitely know that $\mathbf{x} = 10$ whenever program point 2 is reached. In order to deal with equality guards, we thus extend forward analysis by the constraints:

$$[\mathcal{F}4] \quad \mathcal{F}(v) \subseteq [\![p = 0]\!]^\sharp(\mathcal{F}(u)) \qquad \text{for each } (u, (p = 0), v) \in E$$

where the effect of an equality guard is given by:

$$[\![p = 0]\!]^\sharp\, I \;=\; I \oplus \langle p \rangle$$

This formalises our intuition that after the guard, we additionally know that $p = 0$ holds. Such an approximate treatment of equality guards is common in forward program analysis and already proposed by Karr [Kar76]. A similar extension is also possible for inferring transition invariants. The new effect is monotonic. However, it is no longer distributive, i.e., it does not commute with intersections. Due to monotonicity, the extended constraint systems $\mathcal{F}$ as well as $\mathcal{T}$ still have greatest solutions which provide safe approximations of the sets of all valid invariants and transition invariants in presence of equality guards, respectively.

Example 15 . Consider the program from Fig. 2.8. For program point 1 we have:

$$\begin{array}{rcl}\mathcal{F}(1) & = & \langle \mathbf{x} \rangle \cap \langle \mathbf{x} - 1 \rangle \cap \langle \mathbf{x} - 2 \rangle \cap \ldots \\ & = & \{0\}\end{array}$$

Accordingly, we find for program point 2,

$$\begin{array}{rcl}\mathcal{F}(2) & = & \{0\} \oplus \langle \mathbf{x} - 10 \rangle \\ & = & \langle \mathbf{x} - 10 \rangle\end{array}$$

Thus, given the lower bound $\{0\}$ for the infinite decreasing chain of program point 1, we arrive at the desired result for program point 2. □

It would be nice if also backward analysis could be extended with some approximate method for equality guards. Our idea for such an extension is based on *Lagrange multipliers*. Recall that the *weakest* precondition for validity of $q = 0$ after a guard $p = 0$ is given by:

$$(p = 0) \Rightarrow (q = 0)$$

which, for every λ, is implied by:

$$q + \lambda \cdot p = 0$$

The value λ is called a *Lagrange*-multiplier and can be arbitrarily chosen. We remark that a related technique has been proposed in [Cou05] for inferring parametric program invariants. Thus, we define:

$$[\![p = 0]\!]^{\mathsf{T}}(q) = \langle q + p \cdot \lambda \rangle \tag{2.2}$$

where a different formal multiplier λ is chosen for every occurrence of an equality guard. Similar to the treatment of generic postconditions, the parameters λ will occur linearly in a suitably chosen set of generators for the precondition ideal at program start where they can be chosen appropriately.

Example 16 . Again consider the program from Fig. 2.8 and assume that we are interested in identities up to degree 1 at the exit point of the program. Thus we start with the generic polynomial $\mathbf{a}\mathbf{x} + \mathbf{b} = 0$ *at node 2. This gives us for program point 1:*

$$\begin{aligned} \mathcal{B}_1(1) &= \langle (\mathbf{a} + \lambda) \cdot \mathbf{x} + \mathbf{b} - 10\lambda \rangle \ \oplus \\ &\quad\ \langle (\mathbf{a} + \lambda) \cdot \mathbf{x} + \mathbf{a} + \lambda + \mathbf{b} - 10\lambda \rangle \\ &= \langle \mathbf{a} + \lambda, \mathbf{b} - 10\lambda \rangle \end{aligned}$$

Choosing $\lambda = -\mathbf{a}$*, we obtain* $\mathbf{b} = -10\mathbf{a}$*. Therefore all multiples of the polynomial* $\mathbf{x} - 10$ *are valid identities for program point 2.* □

Instead of using a single variable λ as a Lagrange multiplier we could also use an entire polynomial. This means that we use in (2.2) a generic polynomial q_D (for some set D of exponent tuples) instead of the variable λ for each equality guard $p = 0$:

$$[\![p = 0]\!]^{\mathsf{T}}(q) = \langle q + p \cdot q_D \lambda \rangle$$

where we use new variables $A_D = \{\mathbf{a}_\sigma \mid \sigma \in D\}$ in q_D for each equality guard. Now, all the variables in A_D can be adjusted in the computed weakest precondition. This may lead to more precise results – at the price of a more expensive analysis.

2.3.8 Local variables

In this section, we extend the semantics of our programs from global variables to global and local variables. Parameter passing by value is possible via the global variables in this setting. We introduce a new set of variables $\mathbf{Y} = \{\mathbf{y}_1, \ldots, \mathbf{y}_m\}$ to hold the values of the locals. The vector representing a program state thus grows in size by m, with

the meaning that for a program state x, the expression x_{k+i} is the value of the local variable $\mathbf{y}_i$. We readily extend the transformers s for the concrete semantics thus to $[\![s]\!] : \mathbb{Q}^{k+m} \mapsto \mathbb{Q}^{k+m}$.

The only parts which we change now are the procedure call sites in concrete semantics. Therefore, we introduce a helper function $enter$, which carries the values of the globals at the call site over to the callee:

$$enter(X) = \{(x_1, \ldots, x_k, c_1, \ldots, c_m) \mid c_i \in \mathbb{Q}, x \in X\}$$

Likewise, we now extend the present procedure call specification to a version, which respects that the locals of the caller have to be preserved during a procedure call:

$$H(f)(X) = \bigcup_{x \in X} \{(x'_1, \ldots, x'_k, x_{k+1}, \ldots, x_m) \mid x' \in (f(enter(\{x\})))\}$$

These two helper functions now replace their counterparts in the concrete semantics, yielding the following updated versions of the constraints:

[R3′]	$\mathbf{R}_t[u] \supseteq H(\mathbf{R}_{\mathsf{r}_f}(\mathsf{st}_f)) \circ \mathbf{R}_t[v]$	for each $(u, f(), v) \in E$
[$\mathcal{C}$3′]	$\mathcal{C}[v] \supseteq H(\bigcup [\![r]\!])(\mathcal{C}[u]) \mid r \in \mathbf{R}_{\mathsf{r}_f}(\mathsf{st}_f)$	for each $(u, f(), v) \in E$
[$\mathcal{C}$4′]	$\mathcal{C}[\mathsf{st}_f] \supseteq enter(\mathcal{C}[u])$	for each $(u, f(), _) \in E$

After having specified, how procedure call with local variables is handled in the concrete semantics, we now present, how to analyse this within the backward analysis framework. Conceptually, as we assume for local variables to be fresh at the beginning of a procedure's body, we fix this precondition in the abstract $enter^{-1}$ function. Conceptually, this can be implemented by a sequence of non-deterministic assignments to the locals, but can also be done concisely the following way:

$$enter^{-1}(I) = \langle \{q_j \mid q_j \in \mathbb{Q}[\mathbf{X} \cup \mathbf{A}_D] \,,\, \sum_{j=1}^{\infty} q_j \mathbf{y}_1^{j_1} \mathbf{y}_m^{j_m} \ = q, q \in I\} \rangle$$

Notice, that this is the exact inverse of the $enter$ function on the concrete semantics:

Lemma 4 . *For $I \subseteq \mathbb{Q}[\mathbf{X} \cup \mathbf{A}_D \cup \mathbf{Y}]$ and $X \in \mathbb{Q}^{k+m}$:*

$$p(x) = 0 \mid x \in enter(X), p \in I \quad \Leftrightarrow \quad p(x) = 0 \mid x \in X, p \in enter^{-1}(I)$$

Proof.

$$p(x) = 0 \mid x \in enter(X), p \in I \quad \Leftrightarrow$$

$$p(x_1, \ldots, x_k, c_1, \ldots, c_m) = 0 \mid x \in X, p \in I, c_i \in \mathbb{Q} \quad \Leftrightarrow$$

$$p(x) = \sum_{d \in \mathbb{N}_0^m} (q_d(x)\mathbf{y}^d) \wedge \forall_d . q_d(x) = 0 \mid x \in X, q_d \in \mathbb{Q}[\mathbf{X} \cup \mathbf{A}_D], p \in I \quad \Leftrightarrow$$

$$p(x) = 0 \mid x \in X, p \in \langle q_d \rangle, \sum_{d \in \mathbb{N}_0^m} q_d \mathbf{y}^d \in I \quad \Leftrightarrow$$

$$p(x) = 0 \mid x \in X, p \in \mathit{enter}^{-1}(I)$$

□

Finally, we introduce the helper function H^{-1}, which applies the weakest precondition effect of a procedure call taking the separated namespaces of locals in caller and callee into account:

$$H^{-1}(f)(I) = (\mathit{enter}^{-1}(f(I[\tilde{\mathbf{Y}}/\mathbf{Y}])))[\mathbf{Y}/\tilde{\mathbf{Y}}]$$

where $[\tilde{\mathbf{Y}}/varY]$ is short for $[\tilde{\mathbf{y}}_1/\mathbf{y}_1, \ldots, \tilde{\mathbf{y}}_m/\mathbf{y}_m]$. This helper function is also exact, given that the weakest precondition transformer for f is:

Theorem 9 . *Let* $p(x) = 0 \mid x \in f(X), p \in I \Leftrightarrow p(x) = 0 \mid x \in X, p \in f^{-1}(I)$. *Then*

$$p(x) = 0 \mid x \in H(f)(X), p \in I \quad \Leftrightarrow \quad p(x) = 0 \mid x \in X, p \in H^{-1}(f^{-1})(I)$$

Proof.

$$p(x) = 0 \mid x \in X, p \in H^{-1}(f)(I) \quad \Leftrightarrow$$

$$p(x) = 0 \mid x \in X, p \in (\mathit{enter}^{-1}(f^{-1}(I[\tilde{\mathbf{Y}}/\mathbf{Y}]))\,[\mathbf{Y}/\tilde{\mathbf{Y}}]) \quad \Leftrightarrow$$

$$p(x) = 0 \mid x \in X[\tilde{\mathbf{Y}}/\mathbf{Y}], p \in (\mathit{enter}^{-1}(f^{-1}(I[\tilde{\mathbf{Y}}/\mathbf{Y}]))) \quad \Leftrightarrow$$

By lemma 4:

$$p(x) = 0 \mid x \in \mathit{enter}(X[\tilde{\mathbf{Y}}/\mathbf{Y}]), p \in (f^{-1}(I[\tilde{\mathbf{Y}}/\mathbf{Y}])) \quad \Leftrightarrow$$

By applying the precondition:

$$p(x) = 0 \mid x \in f(\mathit{enter}(X[\tilde{\mathbf{Y}}/\mathbf{Y}])), p \in (I[\tilde{\mathbf{Y}}/\mathbf{Y}]) \quad \Leftrightarrow$$

As neither enter nor f are defined on $\tilde{\mathbf{Y}}$, we obtain:

$$p(x'') = 0 \mid x'' \in \bigcup_{x \in X} \{(x'_1, \ldots, x'_k, x_{k+1}, \ldots, x_m) \mid x' \in (f(\mathit{enter}(\{x\})))\}, p \in I \quad \Leftrightarrow$$

$$p(x) = 0 \mid x \in H(f)(X), p \in I$$

□

This theorem yields that by treating local variables with weakest preconditions, we do not lose any further precision, and thus come to the modified versions for our constraints, characterising the weakest preconditions of a conjecture $q \doteq 0$ at a program point t:

$[\mathcal{E}3']$	$\mathcal{E}[u] \supseteq H^{-1}(\mathcal{E}_D(\mathsf{st}_f)) \circ \mathbf{W}(\mathcal{E}[v])$	for each $(u, f(), v) \in E$
$[\mathrm{R}5^{\sharp\prime}]$	$\mathbf{R}^\sharp_{p_t}[u] \supseteq H(\mathcal{E}_D(\mathsf{st}_f))(\mathbf{W}(\mathbf{R}^\sharp_{p_t}[v]))$	for each $(u, f(), v) \in E$
$[\mathrm{R}6^{\sharp\prime}]$	$\mathbf{R}^\sharp_{p_t}[f] \supseteq \mathit{enter}^{-1}(\mathbf{R}^\sharp_{p_t}[\mathsf{st}_f])$	for each f

2.3.9 Conclusion

We summarised forward and backward analyses for inferring valid polynomial equalities. For programs without procedure calls, we characterised all valid polynomial equalities for each program point with the help of a constraint system. However, as we lack of a possibility to compute the least solution of this system via fixpoint iteration, an analysis based on this forward semantics remains incomplete. However, the backward analysis which we introduced allows to compute the weakest precondition for the validity of a polynomial equality exactly. By analysing a generic polynomial template, it even yields all valid equalities up to a degree d.

Introducing procedure calls in the framework complicates matters significantly. Apart from optimistic tabulation of effects, we demonstrated the approach of polynomial transition invariants in order to represent the effects of procedure calls in forward analysis. However, we show that this approach suffers from the same problem as the intraprocedural analysis, namely that it misses polynomial equalities in the worst case. As an alternative, we evaluate the idea of computing effects of procedures as weakest preconditions for template polynomials, similar to the approach of inferring equalities in the intraprocedural case. However, we show that this idea leads only to limited success, as the template is naturally bounded in its maximal degree, which can be exceeded in certain cases.

To round up the picture, we then introduced two further extensions to the model, one providing some support for equality guards – even though this extension is not complete, it nonetheless enables us to consider more information from real-world programs in our analyses. The second extension finally shows, how to embed more realistic handling of different scopes of variables seamlessly into our framework.

2.4 Analysing polynomials in $\mathbb{Z}_{2^w}$

There are several reasons, why analysing polynomials over $\mathbb{Q}$ is not the end of the line. In particular, there are more relations, which are valid when interpreting the values of variables over $\mathbb{Z}_{2^w}$ then over $\mathbb{Q}$. These relations are most important e.g. when dealing with obfuscated programs, as exploiting these relations is popular among so called *opaque predicates*, which make sure that nasty pieces of code, meant to confuse program analysers, are not reachable in any execution. Being able to analyse these predicates with an interpretation over $\mathbb{Z}_{2^w}$ is thus crucial. Besides that, analysing over $\mathbb{Z}_{2^w}$ also means to reason about another problem category as reasoning with program states is a suddenly a finite problem when interpreting values over $\mathbb{Z}_{2^w}$.

The previous analyses interpret the values of variables regarding the field $\mathbb{Q}$. Modern computer architectures, on the other hand, provide arithmetic operations modulo suitable powers of 2. It is well-known that there are equalities valid modulo 2^w, which do not hold in general. The polynomial $2^{31}\mathbf{x}(\mathbf{x}+1)$, for example, constantly evaluates to 0 modulo 2^{32} but may show non-zero values over $\mathbb{Q}$. Accordingly, an analysis based on $\mathbb{Q}$ will systematically miss a whole class of potential program invariants.

```
1 int b = ?;
2 int c = 2147483648, y = 0, x = 0;
3 while(y-b!= 0){
4   x = c*x*x + (c+1)*x + 1;
5   y = x*x + y;
6 }
```

Figure 2.9: Computing the square power sum on 32bit machines

Example 17 . As an example, consider the program from figure 2.9. This program repeatedly increases the value of program variable x *in line* 4 *by 1 – if arithmetic is modulo* 2^{32}*. Therefore, the program* powersum() *computes a square sum. Thus, at program line* 6 *the polynomial invariant* $2\cdot\mathbf{x}^3+3\cdot\mathbf{x}^2+\mathbf{x}-6\cdot\mathbf{y}=0$ *holds modulo* 2^{32} *— but not over the field* $\mathbb{Q}$*.*

An exact analysis of the example program should take into account the structure of polynomials over the domain $\mathbb{Z}_{2^{32}}$: The right hand side in the assignment in line 4 of the example can be rewritten as $2^{31}\mathbf{x}(\mathbf{x}+1)+\mathbf{x}+1$ where the first summand $2^{31}\mathbf{x}(\mathbf{x}+1)$ is equivalent to the zero polynomial over $\mathbb{Z}_{2^{32}}$. Such polynomials are called *vanishing*. Singmaster [Sin74] investigates the special structure of univariate vanishing polynomials over $\mathbb{Z}_m$ and provides necessary and sufficient conditions for a polynomial to vanish over $\mathbb{Z}_m$. Hungerbühler and Specker extend this result to multivariate polynomials and introduce a *canonical form* for polynomials in quotient rings [HS06]. Shekhar et.al. present an algorithm to compute this canonical representation over the quotient ring $\mathbb{Z}_{2^w}$ [SKEG05]. A minimal Gröbner base characterising all vanishing polynomials in arbitrary quotient rings is given by Wienand in [Wie07]. In contrast to the infinite field $\mathbb{Q}$, the ring $\mathbb{Z}_{2^w}$ is finite. Therefore, there are just finitely many distinct k-ary polynomial functions. In fact, it will turn out that we can restrict ourselves to polynomials in k variables up to a total degree $1.5(w+k)$. Due to this upper bound on the total degrees of the polynomials of interest, the problem of checking or inferring of polynomials over

$\mathbb{Z}_{2^w}$ becomes an analysis problem over finite domains only and therefore trivially is computable. Hence, the key issue is to provide tight upper complexity bounds as well as algorithms which also show decent behaviour on practical examples.

In this section, we first consider the problem of checking whether a given polynomial relation is valid at a given program point. While being decidable over $\mathbb{Q}$, we treat more precise complexity bounds for the analogous problem over $\mathbb{Z}_{2^w}$. Furthermore, we present a practical algorithm for this problem which is based on effective precise weakest precondition computation. In case that the number of variables is bounded by a (small) constant, this algorithm even runs in polynomial time.

Secondly, we consider the problem of inferring all polynomial relations which are valid at a given program point. This problem, though not known to be computable in $\mathbb{Q}$, turns out to be computable in exponential time over $\mathbb{Z}_{2^w}$. Again, we present an algorithm for inferring all polynomial invariants of a given shape, whose runtime turns out to be polynomial given that the number of variables is bounded by a constant. Both algorithms have been implemented, and we report on preliminary experiments.

2.4.1 Related work

The pioneer in the area of finding polynomial relations was Karr [Kar76] who inferred the validity of polynomial relations of degree at most 1 (i.e., affine relations) over programs using affine assignments and tests only. An algorithm for checking validity of polynomial relations over programs using polynomial assignments is provided by Müller-Olm and Seidl [MOS02] and was extended later to deal with disequality guards as well [MOS04a]. Their approach is based on effective weakest precondition computations where conjunctions of polynomial relations are described by *polynomial ideals*. Termination of a fixpoint computation in $\mathbb{Q}$ thus is guaranteed by Hilbert's base theorem. In [MOS04a], the authors also observe that their method for checking the validity of polynomial relations can be used to construct an algorithm for inferring all polynomial invariants up to a fixed degree. In [RCK04b, RCK07] Rodriguez-Carbonell et al. pick up the idea of describing invariants by polynomial ideals and propose a *forward* propagating analysis, based on a constraint system over these ideals. As infinite *descending* chains of polynomial ideals cannot be avoided in $\mathbb{Q}$ when merging execution paths [MOPS06, Example 1], they provide special cases or widening techniques to infer polynomial identities. Sankaranarayanan et al. also investigate polynomial invariants[SSM04]. They propose to use *polynomial templates* to capture the effect of assignments in their analysis. These templates describe parametric polynomial properties. When determining the generic parameters via Gröbner bases, certain inductive invariants can be inferred. In contrast to the former approaches, Colon [Col07] provides an *interprocedural forward analysis* for polynomial programs. This analysis is based on ideals of polynomial *transition invariants*. In order to deal with infinite descending chains, Colon abstracts ideals with *pseudo-ideals*, which essentially are vector spaces of polynomials up to a given degree. The application of weakest precondition computations to interprocedural analysis of polynomial relations over $\mathbb{Q}$ is discussed in [MOPS06]. An exact (even interprocedural) analysis of affine relations for programs using affine assignments over

the domain $\mathbb{Z}_m$ is provided in [MOS05a, MOS07]. In practise, the special properties of modular arithmetic are used consciously in the area of control flow obfuscation as demonstrated e.g. in [LD03]. In this context so called *opaque predicates*, i.e. predicates which dynamically evaluate always to the same constant value, are placed as guards for fragments of code which serves to throw static analysers off track instead of being executed. In the context of $\mathbb{Z}_{2^w}$, an instruction like `if((x*x+x)<<31)` for example always evaluates to zero, interpreted in $\mathbb{Z}_{2^w}$, creating room in its body for instructions that confuse an analyser. A custom made solution for breaking special opaque predicates is given in [PMBG06], which fixes the family of opaque predicates to patterns like $2|(\mathbf{x}^2 + \mathbf{x})$.

This chapter is organised as follows. In Section 2.4.2 we specify the concrete semantics for the program class that is inspected by our analysis by means of control flow graphs. Section 2.4.3 gives a detailed description of the characteristics of polynomials in $\mathbb{Z}_{2^w}$. In Section 2.4.4, we first provide the complexity class for the general case of verifying polynomial invariants in $\mathbb{Z}_{2^w}$. We then specify our abstraction for the concrete semantics with the help of polynomial ideals. In Section 2.4.5 we present our specific concrete representation of polynomial ideals in $\mathbb{Z}_{2^w}$. We show how they contribute in the case of constantly many program variables to a better runtime complexity than the theoretical worst case in Section 2.4.4. We then illustrate in Section 2.3.3, how to extend this procedure to infer valid invariants up to a fixed degree and thus for inferring all valid relations. Section 2.4.8 finally summarises our results.

2.4.2 Concrete semantics

In this section we introduce the relation of general concrete semantics with modular aspects, which we are going to analyse in this chapter. Basically, we emanate from the control flow graphs, as described in section 2.2.1. As for this section, the main aspect is the difference of analysing with modular arithmetics instead of $\mathbb{Q}$ or $\mathbb{Z}$, we only consider programs without procedure call here – any way, an interprocedural extension would be possible, with the means from section 2.3.6.

In order to analyse properties within $\mathbb{Z}_{2^w}$, we instantiate the framework from section 2.2.2 with $\mathbb{Z}_{2^w}$ for $\mathbb{V}$. We thus consider the program variables $\mathbf{X}$ to take values in the ring $\mathbb{Z}_{2^w}$, with program states are now modelled by a k-dimensional vector $x = (x_1, \ldots, x_k) \in \mathbb{Z}_{2^w}^k$. Expressions p on the right-hand-side of the assignments $\mathbf{x}_i := p$ are assumed to be polynomial, i.e. either addition or multiplication, and carried out in the ring $\mathbb{Z}_{2^w}$. All other right-hand-side expressions are assumed to be abstracted via non-deterministic assignments. Thus, the instances of our program statements in the concrete semantics are very similar to the ones, when analysing polynomials over $\mathbb{Q}$, as in section 2.3.1. In the case of a disequality guard $(p \neq 0)$ this results in a partial identity function:

$$[\![(p \neq 0)]\!]_{\mathbb{Z}_{2^w}}(X) \;=\; \{x \in \mathbb{Z}_{2^w} \mid p(x) \neq 0\}$$

A polynomial assignment $\mathbf{x}_j := p$ on x causes the evaluation of $p(x)$ over $\mathbb{Z}_{2^w}$ for the

value of $\mathbf{x}$ as transformation:

$$[\![\mathbf{x}_i := p]\!]_{\mathbb{Z}_{2^w}}(X) = \{(x_1, \ldots, x_{i-1}, [\![p(x)]\!]_{\mathbb{Z}_{2^w}}, x_{i+1}, \ldots, x_k) \mid x \in X\}$$

As for a representation π of these partial transformers, we can again take pairs of a polynomial and a total transformer, i.e. $\pi = (q, \tau)$, with an interpretation analogous to the one in for $\mathbb{Q}$ as in section 2.3.1:

$$\begin{array}{lcl} [\![\mathbf{x}_i := p]\!]_{\mathbb{Z}_{2^w}} & = & (1, [\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, p, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_k]) \\ [\![q \neq 0]\!]_{\mathbb{Z}_{2^w}} & = & (q, [\mathbf{x}_1, \ldots, \mathbf{x}_k]) \end{array}$$

Instantiating these transformers leads to the following constraints as our reference semantics, its least solution characterising for each program point u the runs, which reach a program point t:

$$\begin{array}{ll} [\text{R1}] & \mathbf{R}_t(t) \supseteq \{\varepsilon\} \\ [\text{R2}] & \mathbf{R}_t(u) \supseteq [\![s]\!]_{\mathbb{Z}_{2^w}} \circ (\mathbf{R}_t(v)), \text{ for each } (u, s, v) \in E \end{array}$$

In the course of this section, we may omit the index $\mathbb{Z}_{2^w}$, as all operations are carried out in $\mathbb{Z}_{2^w}$ per default. The analysis in this section provides a precise abstract interpretation of this constraint system [R].

2.4.3 The ring of polynomials in $\mathbb{Z}_{2^w}$

In order to develop an analysis inferring all polynomial relations modulo $m = 2^w$, we fix a bit width $w \geq 2$ of our numbers in the following. For $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$, let $\mathbb{Z}_{2^w}[\mathbf{X}]$ denote the ring of all polynomials with coefficients in $\mathbb{Z}_{2^w}$. Recall that $\mathbb{Z}_{2^w}$ is not a field. More precisely, only all odd elements are invertible while every even element is a *zero divisor*. Thus, e.g., $2 \cdot 2^{w-1} \equiv 0$ in $\mathbb{Z}_{2^w}$. Useful facts about this ring can be found in [MOS05a] or basic text books on commutative ring theory, as [Mat89].

Similarly to the case of fields [MOS05a], the set of polynomials $p \in \mathbb{Z}_{2^w}[\mathbf{X}]$ which evaluate to 0 for a given subset $X \subseteq \mathbb{Z}_{2^w}^k$, is closed under addition and multiplication with arbitrary polynomials, and is thus an *ideal*. Thus, our program analysis maintains for every program point an ideal of polynomials.

Recall that the ring $\mathbb{Z}_{2^w}$ is a *principal ideal* ring meaning that every ideal $I \subseteq \mathbb{Z}_{2^w}$ can be represented as the set $I = \{z \cdot a \mid z \in \mathbb{Z}_{2^w}\}$ of all multiples of a single ring element a. Thus by Hilbert's basis theorem, every ideal $I \subseteq \mathbb{Z}_{2^w}[\mathbf{X}]$ can be represented as the set of all linear combinations of a finite set $G = \{g_1, \ldots, g_n\} \subseteq \mathbb{Z}_{2^w}[\mathbf{X}]$, i.e., $I = \{p_1 g_1 + \ldots + p_n g_n \mid p_i \in \mathbb{Z}_{2^w}[\mathbf{X}]\}$. In this case, we also refer to G as the set of generators of I and denote this by $I = \langle G \rangle$.

Further notice, that the specific devisability rules for $\mathbb{Z}_{2^w}$ also influence the reducibility conditions for polynomials over $\mathbb{Z}_{2^w}$: Let $p, p' \in G$ be polynomials which share the same monomial t in their headterm, i.e. $HT(p) = a \cdot 2^e \cdot s \cdot t$ and $HT(p') = a' \cdot 2^{e'} \cdot t$ with $e \geq e'$, some monomial s and odd $a, a' \in \mathbb{Z}_{2^w}$. In this case, we say that p is *reducible* by p', regardless whether a or a' is greater. Of course, this is also extendable

to *reducibility by a set* R of polynomials. If p is reducible by the polynomial p', p can be reduced to the polynomial $q = a' \cdot p - a \cdot 2^{e-e'} \cdot s \cdot p'$. If $q = 0$, p is a multiple of p' and thus redundant in every set G of generators containing p', i.e., $\langle G \backslash \{p\} \rangle = \langle G \rangle$. If $q \neq 0$, we can replace the polynomial p in G with the (simpler) polynomial q, i.e., the set G generates the same ideal as the set $G' = (G \backslash \{p\}) \cup \{q\}$.

Starting from a set G of generators, we can successively apply reduction to eventually arrive at a *reduced* set $\bar{G}$ generating the same ideal as G. Here, we call the set $\bar{G}$ reduced iff no polynomial $p \in \bar{G}$ is reducible w.r.t. $\bar{G} \backslash \{p\}$.

Lemma 5 . *Assume that $p \in \mathbb{Z}_{2^w}[\mathbf{X}]$ and $G \subseteq \mathbb{Z}_{2^w}[\mathbf{X}]$ is a finite reduced set of polynomials. Then a reduced set $\bar{G} \subseteq \mathbb{Z}_{2^w}[\mathbf{X}]$ can be constructed with $\langle \{p\} \cup G \rangle = \langle \bar{G} \rangle$. The algorithm runs in time $\mathcal{O}(k \cdot r^2)$ if r is the number of different exponents of monomials occurring during reduction, and k the number of program variables.*

Proof. A single reduction of a polynomial by a set of reduced polynomials can be carried out by as many subtractions (each of cost k) as a polynomial has monomials. This number is bounded by the number of different exponents r occuring during the reduction. Adding p to G is carried out by reducing potentially $|G| \leq r$ many polynomials. □

Here, the total degree of a monomial $\mathbf{x}_1^{r_1} \ldots \mathbf{x}_k^{r_k}$ is $d = r_1 + \ldots + r_k$, and the total degree of a polynomial is the total degree of its head monomial. Thus, the number r of possibly occurring different exponents of monomials is bounded by:

Proposition 1 . *The number of different exponents of monomials is given by*

$$r \leq \binom{d+k}{k} \leq min(\ (d+1)^k\ ,\ (k+1)^d\ )$$

Note that the upper complexity bound is a crude worst-case estimation only. The practical run-time might be much smaller if the occurring polynomials are short, i.e., contain only few monomials.

Now assume that the ideal I is generated from a set G of generators and p is a polynomial. If p can be reduced (perhaps in several steps) to the 0 polynomial by means of the polynomials in G, then $p \in I$. The reverse, however, is only true for particularly saturated sets of generators such as *Gröbner bases* [BW93].

In the case of polynomials over a field, the constant zero polynomial is the only polynomial which evaluates to 0 for all vectors $x \in {\mathbb{Z}_{2^w}}^k$. This is no longer the case for the polynomial ring $\mathbb{Z}_{2^w}[\mathbf{X}]$. Let $I_v \subseteq \mathbb{Z}_{2^w}[\mathbf{X}]$ denote the ideal of all polynomials p with $p(x) = 0$ for all $x \in {\mathbb{Z}_{2^w}}^k$. The elements of I_v are also called *vanishing* polynomials. Only recently, a precise characterisation of the ideal I_v has been provided by Hungerbühler and Specker [HS06], which is recalled briefly, here. The first observation is that whenever 2^e divides $r! = r(r-1)\ldots 1$, then 2^e also divides $(x+r-1) \cdot \ldots \cdot (x+1) \cdot x$ for all x. Let $\nu_2(y)$ denote the maximal exponent e such that 2^e divides y. Thus, e.g., $\nu_2(1!) = 0$, $\nu_2(2!) = \nu_2(3!) = 1$ and $\nu_2(2^s!) = 2^s - 1$ for all $s \geq 1$. In particular, $\nu_2(r!) \geq w$ for $r \geq w + \log(w) + 1$. Since $1.5w + 1 \geq w + \log(w) + 1$, this implies that $\nu_2(r!) \geq \frac{2}{3}r - 1$.

Now consider the polynomial $p_r(\mathbf{x}_i) = \mathbf{x}_i \cdot (\mathbf{x}_i + 1) \cdot \ldots \cdot (\mathbf{x}_i + r - 1)$. Then $2^{\nu_2(r!)}$ divides the value $p_r(z)$ for every z. Thus we obtain the following family $G(k, w)$ of

vanishing polynomials:

$$2^a \cdot p_{r_1}(\mathbf{x}_1) \cdot \ldots \cdot p_{r_k}(\mathbf{x}_k)$$

where $a \geq 0$, and $a + \nu_2(r_1!) + \ldots + \nu_2(r_k!) \geq w$.

Example 18 . Take $\mathbb{Z}_{2^2}$ as domain. Then $p = \mathbf{x}^4 + 2\mathbf{x}^3 + 3\mathbf{x}^2 + 2\mathbf{x} = \mathbf{x}(\mathbf{x}+1)(\mathbf{x}+2)(\mathbf{x}+3) = p_4(\mathbf{x})$. Since $\nu_2(4!) = 3 \geq 2$, $p(\mathbf{x})$ is a vanishing polynomial. □

Note that Wienand [Wie07] proves that the set $G(k, w)$ is not only contained in I_v but that I_v is in fact generated by $G(k, w)$.

Two polynomials $p, p' \in \mathbb{Z}_{2^w}[\mathbf{X}]$ are *semantically* equivalent if they define the same function ${\mathbb{Z}_{2^w}}^k \to \mathbb{Z}_{2^w}$, i.e., if $p - p' \in I_v$. We observe that for every polynomial in $\mathbb{Z}_{2^w}[\mathbf{X}]$, we can effectively find a semantically equivalent polynomial of small total degree. We have:

Lemma 6 . *Every polynomial $p \in \mathbb{Z}_{2^w}[\mathbf{X}]$ in k variables is semantically equivalent to a polynomial $p' \in \mathbb{Z}_{2^w}[\mathbf{X}]$ of degree less than $1.5(w + k)$.*

Proof. Let p' denote a polynomial of minimal total degree r and minimal number of monomials of total degree r which is semantically equivalent to p. Assume for a contradiction that $r \geq 1.5(w+k)$ and t is a monomial in p' of maximal total degree. Then t can be written as $t = a \cdot \mathbf{x}_1^{r_1} \ldots \mathbf{x}_k^{r_k}$ where w.l.o.g. all $r_i \geq 1$. Then $\nu_2(r_j!) \geq \frac{2}{3}r_j - 1$ for all j. Consequently, the sum of these values is at least

$$\sum_j (\frac{2}{3}r_j - 1) = \frac{2}{3}\sum_j (r_j - 1.5) = \frac{2}{3}(1.5(w + k) - 1.5k) = w$$

Therefore, the polynomial $q = p_{r_1}(\mathbf{x}_1) \cdot \ldots \cdot p_{r_k}(\mathbf{x}_k)$ is vanishing. Note that the polynomial q has exactly one monomial of maximal total degree. We conclude that $p'' = p' - a \cdot q$ is a polynomial which is still semantically equivalent to p. Moreover the total degree of p'' is not larger than the total degree of p' and if the respective total degrees are equal, then p'' has less monomials of maximal total degree – thus contradicting our assumption. □

Example 19 . Consider the polynomials $p = \mathbf{x}^4 + 3\mathbf{x}$ and $p' = 2\mathbf{x}^3 + \mathbf{x}^2 + \mathbf{x}$ over $\mathbb{Z}_{2^2}$. Subtracting p' from p results in $q = p - p' = \mathbf{x}^4 + 2\mathbf{x}^3 + 3\mathbf{x}^2 + 2\mathbf{x} \in I_v$. Thus, p and p' are equivalent. □

In [SKEG05], Shekhar et al. prove that a polynomial p is vanishing iff p can be reduced to 0 by means of the polynomials in $G(k, w)$. In the worst case, this takes $\mathcal{O}((d + 1)^k)$ reduction steps if d is the total degree of p. As each of these reductions involves $\mathcal{O}((d + 1)^k)$ many different monomials in the worst case, checking a polynomial for vanishing by means of reduction costs $\mathcal{O}((d + 1)^{2k})$. Here, we sketch an alternative method. It consists in evaluating the polynomial for a finite set of selected arguments. The latter technique is based on the following observation.

Lemma 7 . *Let $\mathcal{R}$ be an arbitrary ring. A polynomial $p \in \mathcal{R}[\{\mathbf{x}\}]$ of degree d is semantically equivalent to the zero polynomial, i.e., $\forall x \in \mathcal{R}. p(x) = 0$ iff $p(h) = 0$ for $h = 0, 1, \ldots, d$.*

Proof. "$\Rightarrow$" is trivial.
"$\Leftarrow$" by induction on degree d:
case $d = 0$*:* If the degree is zero, the polynomial is just described by a constant function $p(\mathbf{x}) \equiv c$. This polynomial is only zero for any h, if the constant value c is zero. Therefore $p \equiv 0$ and the assertion follows.
case $d > 0$*:* Let $p(h) = 0$ for $h = 0, \ldots, d$. We consider the polynomial q of degree $d-1$ with $q(\mathbf{x}) = p(\mathbf{x}+1) - p(\mathbf{x})$ that has $q(h') = 0$ at least for $h' = 0, \ldots, d-1$. By induction hypothesis, $q(x) = 0$ for all $x \in \mathcal{R}$. Thus, $p(\mathbf{x})$ and $p(\mathbf{x}+1)$ are semantically equivalent. Since $p(0) = 0$, then also $p(1) = 0$ and thus, by induction, $p(x) = 0$ for all $x \in \mathcal{R}$, and the assertion follows. □

Lemma 7 shows that an arbitrary polynomial p is vanishing iff it vanishes for suitably many argument vectors. Substituting in a polynomial p of degree d all k different variables by $d+1$ values each indicates that $p \notin I_v$, if it does not evaluate to zero each time. Otherwise $p \in I_v$. As evaluating p can be done in $|p|$, we conclude:
Corollary 2 . *Assume* $p \in \mathbb{Z}_{2^w}[\mathbf{X}]$ *is a polynomial where each variable has a maximal degree in* p *bounded by* d*. Then* $p \in I_v$ *can be tested in time* $\mathcal{O}((d+1)^k \cdot |p|)$*.* □

2.4.4 Verifying polynomial relations in $\mathbb{Z}_{2^w}$

Similarly to [MOS04a] we denote a *polynomial relation* over the vector space ${\mathbb{Z}_{2^w}}^k$ as an equation $p = 0$ for some $p \in \mathbb{Z}_{2^w}[\mathbf{X}]$, which is representable by p alone. The vector $y \in {\mathbb{Z}_{2^w}}^k$ *satisfies* the polynomial relation p iff $p(y) = 0$. The polynomial relation p is valid at a program point v iff p is satisfied by $[\![r]\!]x$ for every run r of the program from program start st to v and every vector $x \in {\mathbb{Z}_{2^w}}^k$. In [MOR01], Rüthing and Müller-Olm prove that deciding whether a polynomial relation over $\mathbb{Q}$ is valid at a program point v of an *intraprocedural polynomial program* is at least *PSPACE*-hard. Their lower-bound construction is based on a reduction of the *language universality problem* of non-deterministic finite automata and uses only the values 0 and 1. Therefore, literally the same construction also shows that validity of a polynomial relation over $\mathbb{Z}_{2^w}$ is also *PSPACE*-hard. Regarding an upper bound, we construct a Turing machine which non-deterministically computes a counterexample for the validity of a polynomial relation p. This counterexample can be found by simulating the original program on vectors over $\mathbb{Z}_{2^w}$ representing the program state. The representation of a program state in absence of procedure calls can be done in polynomial space in $\mathbb{Z}_{2^w}$. The Turing machine accepts if it reaches program point v with a state $x \in {\mathbb{Z}_{2^w}}^k$ which does not satisfy p. Thus, we have a *PSPACE*-algorithm for dis-proving the validity of polynomial relations. Since the complexity class *PSPACE* is closed under complementation, we obtain:

Theorem 10 . *Checking validity of polynomial invariants over* $\mathbb{Z}_{2^w}$ *is PSPACE-complete for intraprocedural polynomial programs.* □

Incorporating procedure calls into our program class, another layer of complexity is added when inferring polynomial invariants. In order to verify equalities of this kind,

we must take each procedure's functional effect on polynomial equalities into account. A straight-forward approach is resorting to modular linear arithmetics, introducing an auxiliary variable for each non-zero-reducible monomial of $\mathbb{Z}_{2^w}$. As lemma 6 states that this number of monomials is bounded by $(1.5(w+k)+1)^k$, we arrive at a bound on the number of auxiliary variables, exponential in the number of real program variables. Analysing linear equalities with exponentially many auxilliary variables by means of, e.g. [MOS07] can be achieved in exponential time.

Theorem 11 . *Checking validity of polynomial invariants over $\mathbb{Z}_{2^w}$ is in EXPTIME for interprocedural polynomial programs.* □

This is bad news for a general algorithm for the verification of polynomial invariants over $\mathbb{Z}_{2^w}$. The theoretical algorithm providing the upper bound in theorem 10 is not suitable for practical application. Therefore, we subsequently present an algorithm which has reasonable runtime behaviour at least for meaningful examples. In particular, it has polynomial complexity — given that the number of program variables is bounded by a constant.

This algorithm is based on the effective computation of weakest preconditions. Following [MOS04a], we characterise the weakest precondition of the validity of a relation $p \doteq 0$ at program point t by means of a constraint system on ideals of polynomials.

In order to construct this constraint system, we rely on the weakest precondition transformers $[\![s]\!]^\top$ for tests, single assignments, or non-deterministic assignments:

$$\begin{array}{lll} [\![p \neq 0]\!]^\top q & = & \{p \cdot q\} \\ [\![\mathbf{x}_j := p]\!]^\top q & = & \{q[p/\mathbf{x}_j]\} \\ [\![\mathbf{x}_j := ?]\!]^\top q & = & \{q[h/\mathbf{x}_j] \mid h = 0, \ldots, d\} \end{array}$$

where d is the maximal degree of $\mathbf{x}_j$ in q. The transformers for assignments and disequality tests are the same which, e.g., have been used in [MOS04a]. Only for non-deterministic assignments $\mathbf{x}_j :=?$, extra considerations are necessary. The treatment of non-deterministic assignments in [MOS04a] for $\mathbb{Q}$ consists in collecting all coefficient polynomials p_i not containing $\mathbf{x}_j$ in the sum $q = \sum_{i \geq 0} p_i \cdot \mathbf{x}_j^i$. This idea does no longer work over $\mathbb{Z}_{2^w}$. Consider, e.g., $p = 2^{31}\mathbf{x}_1^2\mathbf{x}_2 + 2^{31}\mathbf{x}_1\mathbf{x}_2$. Equating every $\mathbf{x}_1$-coefficient with zero would lead to the polynomial $2^{31}\mathbf{x}_2 = 0$ — which is not the weakest precondition, as $p = 0$ is trivially valid. The correctness of the new definition of $[\![\mathbf{x}_j :=?]\!]^\top$ for $\mathbb{Z}_{2^w}$ on the other hand, follows from lemma 7.

The weakest precondition transformers $[\![s]\!]^\top$ for polynomials can be extended to transformers of ideals. Assume that the ideal I is given through the set G of generators. Then:

$$[\![s]\!]^\top I = \langle \bigcup \{ [\![s]\!]^\top g \mid g \in G \} \rangle$$

Note that $([\![s]\!]^\top q)$ is vanishing whenever q is already vanishing. Therefore,

$$[\![s]\!]^\top (I_v) \subseteq I_v$$

for all s. Using the extended transformers, we put up the constraint system $\mathbf{R}_t^\sharp$ to represent the precondition for the validity of a polynomial p at program point t:

$$\begin{array}{ll}[\mathrm{R1}]^\sharp & \mathbf{R}_t^\sharp(t) \supseteq \langle\{p\}\rangle \\ [\mathrm{R2}]^\sharp & \mathbf{R}_t^\sharp(u) \supseteq [\![s]\!]^\top(\mathbf{R}_{p_t}^\sharp(v))\,, \text{ if } (u,s,v) \in E\end{array}$$

For all program points, we may safely assume that all vanishing polynomials are valid. Therefore, we may consider the given constraint system over ideals I subsuming I_v, i.e., with $I_v \subseteq I$. This implies that we only consider ideals I where $p \in I$ whenever $p' \in I$ for every polynomial p' which is semantically equivalent to p. Note that the set of ideals subsuming I_v (ordered by the subset relation '$\subseteq$'') forms a complete lattice. Since all transformers $[\![s]\!]^\top$ are monotonic, this system has a unique least solution. Since all transformers $[\![s]\!]^\top$ transform I_v into (subsets of) I_v and distribute over sums of ideals, the least solution of the constraint system precisely characterises the weakest preconditions for the validity of p_t at program point t in a similar way as in [MOS04a]. We have:

Lemma 8 . *Assume that $\mathbf{R}_t^\sharp(u)$, with u a program point, denotes the least solution of the constraint system $[\mathrm{R}^\sharp]$. Then the polynomial relation $p \in \mathbb{Z}_{2^w}[\mathbf{X}]$ is valid at the target node t iff $\mathbf{R}_t^\sharp(\mathsf{st}) \subseteq I_v$.* □

2.4.5 Computing with Ideals over $\mathbb{Z}_{2^w}[\mathbf{X}]$

In order to check the validity of the polynomial relation p_t at program point t, we must find succinct representations for the ideals occurring during fixpoint iteration which allow us first, to decide when the fixpoint computation can be terminated and secondly, to decide whether the ideal for the program start consists of vanishing polynomials only.

The basic idea consists in representing ideals through finite sets G of generators. In order to keep the set G small, we explicitly collect only polynomials *not* in I_v. Thus, G represents the ideal $\langle G\rangle_v = \langle G\rangle \oplus I_v = \{g + g_0 \mid g \in \langle G\rangle, g_0 \in I_v\}$.

Keeping the representation of vanishing polynomials implicit is crucial, since the number of necessary vanishing polynomials in $G(k,w)$ is exponential in k. By lemma 6, only polynomials up to degree $1.5(w+k)$ need to be chosen. By successively applying reduction, we may assume that G is reduced and consists of polynomials which cannot be (further) reduced by polynomials in $G(k,w)$ only. Let us call such sets of generators *normal-reduced*. Then by the characterisation of [SKEG05], $\langle G\rangle_v \subseteq I_v$ iff $G = \emptyset$.

Example 20 . Consider the polynomials $p = \mathbf{x}^5 + \mathbf{x}^4 + 2\mathbf{x}^2 + 3\mathbf{x}$ and $\bar{p} = 6\mathbf{x}^5 + 7\mathbf{x}^3 + 2\mathbf{x}$ over $\mathbb{Z}_{2^3}$. In order to build a normal-reduced set of generators G, $\langle G\rangle_v = \langle\{p,\bar{p}\}\rangle_v$, we begin with a first round, reducing p and $\bar{p}$ with the vanishing polynomial $p_v = \mathbf{x}^4 + 2\mathbf{x}^3 + 3\mathbf{x}^2 + 2\mathbf{x}$: $p' = p + (7\mathbf{x}+1)p_v = 7\mathbf{x}^3 + 3\mathbf{x}^2 + 5\mathbf{x}$ and $\bar{p}' = \bar{p} + (2\mathbf{x}+4)p_v = 5\mathbf{x}^3 + 2\mathbf{x}$. Next, p' can be reduced by $\bar{p}'$, leading to $p'' = p' + 5\bar{p}' = 3x^2 + 7x$. $\bar{p}'$ and p'' are nonreducible with respect to each other and I_v. Then $\langle\{,p'',\bar{p}'\}\rangle_v = \langle\{p,\bar{p}\}\rangle_v$ where the set $\{p'',\bar{p}'\}$ is normal-reduced. □

Concerning the computation of the fixpoint for $\mathbf{R}_{p_t}^\sharp$, consider an edge (u,s,v) in the control-flow graph of the program labelled with s. Each time when a new polynomial

p is added to the ideal $\mathbf{R}^\sharp_{p_t}(v)$ associated with program point v which is not known to be contained in $\mathbf{R}^\sharp_{p_t}(v)$, all polynomials in $[\![s]\!]^\top p$ must be added to the ideal $\mathbf{R}^\sharp_{p_t}(u)$ at program u. The key issue for detecting termination of the fixpoint algorithm therefore is to check whether a polynomial p is contained in the ideal $\mathbf{R}^\sharp_{p_t}(u)$. Assume that the ideal $\mathbf{R}^\sharp_{p_t}(u)$ is represented by the normal-reduced set G of generators. Clearly, the polynomial p is contained in $\langle G\rangle_v = \mathbf{R}^\sharp_{p_t}(u)$ whenever p can be reduced by $G \cup G(k,w)$ to the 0 polynomial. The reverse, however, need not necessarily hold.

Exact ideal membership based on *Gröbner bases* requires to extend the set G with *S-polynomials* [BW93]. However, for generating all S-polynomials, virtually all pairs of generators must be taken into account. This applies also to the vanishing polynomials. The number of vanishing polynomials in $G(k,w)$ of degree $\mathcal{O}(w+k)$, however, is still exponential in k and also may comprise polynomials with many monomials. This implies that any algorithm based on exhaustive generation of S-polynomials cannot provide decent mean- or best case complexity at least in some useful cases. Therefore, we have abandoned the generation of S-polynomials altogether, and hence also exact testing of ideal membership.

Instead of ideals themselves, we therefore work with the complete lattice $\mathbb{D}$ of normal-reduced subsets of polynomials in $\mathbb{Z}_{2^w}[\mathbf{X}]$. The ordering on the lattice $\mathbb{D}$ is defined by $G_1 \sqsubseteq G_2$ iff every element $g \in G_1$ can be reduced to 0 w.r.t. $G_2 \cup G(k,w)$. The least element w.r.t. this ordering is $\emptyset$. Thus by definition, $G_1 \sqsubseteq G_2$ implies $\langle G_1\rangle_v \subseteq \langle G_2\rangle_v$, and $\langle G\rangle_v = I_v$ iff $G = \emptyset$. In order to guarantee the termination of the modified fixpoint computation, we rely on the following observation:

Lemma 9 . *Consider a strictly increasing chain:*

$$\emptyset \sqsubset G_1 \sqsubset \ldots \sqsubset G_h$$

of normal-reduced generator systems over $\mathbb{Z}_{2^w}[\mathbf{X}]$. Then the maximal length h of this chain is bounded by $w \cdot r$ where r is the number of head monomials occurring in any G_i.

Proof. For each G_i consider the set H_i, which denotes the set of terms $t = 2^s \cdot \mathbf{x}_1^{r_1} \ldots \mathbf{x}_k^{r_k}$ for which $a \cdot t$ (a invertible) is the head term of a polynomial in G_i. Then for every i, H_i contains a term t which has not yet occurred in any $H_j, j < i$. The value h thus is bounded by the cardinality of $H_1 \cup \ldots \cup H_h$, which in turn is bounded by $w \cdot r$. □

In case that we are given an upper bound d for the total degree of polynomials in lemma 9, then by prop. 1, the height h is bounded by $h \leq w \cdot r \leq w \cdot (d+1)^k$ and thus is exponential in k only.

The representation of ideals through normal-reduced sets of generators allows us to compute normal-reduced sets of generators for the least solution of the constraint system $\mathbf{R}^\sharp_{p_t}$. We obtain:

Theorem 12 . *Checking the validity of a polynomial invariant in a polynomial program with N nodes and k variables over $\mathbb{Z}_{2^w}$ can be performed in time $\mathcal{O}(N \cdot k \cdot w^2 \cdot r^3)$ where r is the number of monomials occurring during fixpoint iteration.*

Proof.
Verification of a polynomial invariant p_t at a program point t is done via fixpoint iteration on sets of generators. Considering a set $G[u]$ of generators representing the ideal $\mathbf{R}^\sharp_{p_t}(u)$ of preconditions at program point u, we know from lemma 9, that an increasing chain of normal-reduced generator systems is bounded by $w \cdot r$. Each time, that the addition of a polynomial p leads to an increase of $G[u]$, the evaluation of $[\![s]\!]^\top(p)$ is triggered for each edge (u, s, v). Each precondition transformer creates only one precondition polynomial, except for the nondeterministic assignment. Essentially, each $[\![\mathbf{x}_j :=?]\!]^\top$ causes $d+1$ (d the maximal degree of $\mathbf{x}_j$) polynomials to be added to the set of generators at the source u of the corresponding control flow edge. Since the degree d of any variable $\mathbf{x}_j$ in an occurring generator polynomial is bounded by $1.5(w+1)$, we conclude that the total number of increases for the set $G[u]$ of generators for program point u along the control flow edge $(u, _, v)$ amounts to $\mathcal{O}(w^2 \cdot r)$. As we can estimate the complexity of a complete reduction by $\mathcal{O}(k \cdot r^2)$ with the help of lemma 5, we find that the amount of work induced by a single control flow edge therefore is bounded by $\mathcal{O}(k \cdot w^2 \cdot r^3)$. This provides us with the upper complexity bound stated in this theorem.
□

Assume that the maximal degree of a polynomial occurring in an assignment of the input program has degree 2. Then the maximal total degree d of any monomial occurring during fixpoint iteration is bounded by $1.5(w+k) + 3w + 2 = 4.5w + 1.5k + 2$. By prop. 1, the number r of monomials in the complexity estimation of theorem 12 is thus bounded by $(4.5w+k+3)^k$. From that, we deduce that our algorithm runs in polynomial time – at least in case of constantly many variables. Of course, for three variables and $w = 2^5$, the number r of possibly occurring monomials is already beyond $2^{7 \cdot 4} = 2^{28}$ — which is far beyond what one might expect to be practical.

2.4.6 Inferring all polynomials

Still, no algorithm is known which, for a given polynomial program, infers all valid polynomial relations over $\mathbb{Q}$. In [MOS04a] it is shown, however, that at least all polynomial relations up to a *maximal total degree* can be computed. For the finite ring $\mathbb{Z}_{2^w}$, on the other hand, we know from lemma 6 that every polynomial has an equivalent polynomial of total degree at most $1.5(w+k)$. Therefore over $\mathbb{Z}_{2^w}$, any algorithm which computes all polynomial invariants up to a given total degree is sufficient to compute all valid polynomial invariants.

For a comparison, we remark that, since $\mathbb{Z}_{2^w}$ is finite, the collecting semantics of a polynomial program of length N is finite and computable by ordinary fixpoint iteration in time $N \cdot 2^{\mathcal{O}(wk)}$. Given the set $X \subseteq {\mathbb{Z}_{2^w}}^k$ of states possibly reaching a program point v, we can determine all polynomials p of total degree at most $1.5(w+k)$ with $p(x) = 0$ for all $x \in X$ by solving an appropriate linear system of $|X| \leq 2^{wk}$ equations for the coefficients of p. For every program point u, this can be done in time $2^{\mathcal{O}(wk)}$. Here, our goal is to improve on this trivial (and intractable) upper bound. Our contribution is to remove the w in the exponent and to provide an algorithm whose runtime, though

exponential in k in the worst case, may still be much faster on meaningful examples.

For constructing this algorithm, we are geared to the approach from [MOS04a]. This means that we fix a template for the form of polynomials that we want to infer. Such a template is given by a set M of terms $m = \mathbf{x}_1^{r_1} \ldots \mathbf{x}_k^{r_k}$ with $r_1 + \ldots + r_k \leq 1.5(w + k)$. Note that for small maximal total degree d the cardinality of M is bounded by $(k + 1)^d$ while without restriction on d, the cardinality is bounded by an exponential in k (see prop. 1). Given the set M, we introduce a set $\mathbf{A}_M = \{\mathbf{a}_m \mid m \in M\}$ of auxiliary fresh variables $\mathbf{a}_m$ for the coefficients of the terms m in a possible invariant. The template polynomial p_M for M then is given by $p_M = \sum_{m \in M} \mathbf{a}_m \cdot m$.

Example 21 . Consider the program variables $\mathbf{x}_1$ *and* $\mathbf{x}_2$*. Then the template polynomial for the set of all terms of total degree at most 2 is given by:* $\mathbf{a}_1\mathbf{x}_1^2 + \mathbf{a}_2\mathbf{x}_2^2 + \mathbf{a}_3\mathbf{x}_1\mathbf{x}_2 + \mathbf{a}_4\mathbf{x}_1 + \mathbf{a}_5\mathbf{x}_2 + \mathbf{a}_0$. □

With the help of the verification algorithm from section 2.4.4, we can compute the weakest precondition for a given template polynomial p_m. Since during fixpoint computation, no substitutions of the generic parameters $\mathbf{a}_m$ are involved, each polynomial p in any occurring set of generators is always of the form $p = \sum_{m \in M} \mathbf{a}_m \cdot q_m$ for polynomials $q_m \in \mathbb{Z}_{2^w}[\mathbf{X}]$. In particular, this holds for the set of generators computed by the fixpoint algorithm for the ideal at the start point st of the program. We have:

Lemma 10 . *Assume that G is a set of generators of the ideal $\mathbf{R}^\sharp_{p_t}(\mathsf{st})$ for the template polynomial p_M at program point t. Then for any $a_m \in \mathbb{Z}_{2^w}, m \in M$, the polynomial $\sum_{m \in M} a_m m$ is valid at program point t iff for all $g \in G$, the polynomial $g[a_m/\mathbf{a}_m]_{m \in M}$ is a vanishing polynomial.* □

It remains to determine the values $a_m, m \in M$ for which all polynomials g in a finite set G are vanishing. First assume that the polynomials in G may contain variables from $\mathbf{X}$. Assume w.l.o.g. that it is $\mathbf{x}_k$ which occurs in some polynomial in G where the maximal degree of $\mathbf{x}_k$ in polynomials of G is bounded by d. Then we construct a set G' by:

$$G' = \{g[j/\mathbf{x}_k] \mid g \in G, j = 0, \ldots, d\}$$

The set G' consists of polynomials g' which contain variables from $\mathbf{X}\backslash\{\mathbf{x}_k\}$ only. Moreover by lemma 7, $g[a_m/\mathbf{a}_m]_{m \in M}$ is vanishing for all $g \in G$ iff $g'[a_m/\mathbf{a}_m]_{m \in M}$ is vanishing for all $g' \in G'$. Repeating this procedure, we successively may remove all variables from $\mathbf{X}$ to eventually arrive at a set $\bar{G}$ of polynomials without variables from $\mathbf{X}$. This means each $g \in \bar{G}$ is of the form $g = \sum_{m \in M} \mathbf{a}_m \cdot c_m$ for $c_m \in \mathbb{Z}_{2^w}$. Therefore, we can apply the methods from [MOS07] for *linear* systems of equations over $\mathbb{Z}_{2^w}$ (now with variables from $\mathbf{A}_M$) to determine a set of generators for the $\mathbb{Z}_{2^w}$-module of solutions. Thus, we obtain the following result:

Theorem 13 . *Assume p is a polynomial program of length N with k variables over the ring $\mathbb{Z}_{2^w}$. Further assume that M is a subset of terms of total degree bounded by $1.5(w + k)$. Then all valid polynomial invariants $\sum_{m \in M} c_m m$ with $c_m \in \mathbb{Z}_{2^w}$ can be computed in time $\mathcal{O}(N \cdot k \cdot w^2 (r_0 r)^3)$ where r_0 is the cardinality of M and r is the maximal number of terms occurring during fixpoint iteration.*

Proof. Generator sets of polynomials over M and X are always composed of polynomials p of the form $p = \sum_{m \in M} \mathbf{a}_m \cdot \mathbf{x}_1^{d_1} \ldots \mathbf{x}_k^{d_k}$. Thus, the number of occurring different terms is bounded by $r_0 \cdot r$. Therefore, the maximal length of a strictly increasing chain of normal-reduced sets of generators is bounded by $w \cdot r_0 \cdot r$. As the number of terms in a polynomial is bounded by $r_0 \cdot r$, the costs for updating a normal-reduced set of generators with a single polynomial is now $\mathcal{O}(k \cdot (r_0 \cdot r)^2)$. Again, we have to account $1.5(w+1)$ for the number of polynomials which can be produced by weakest precondition transformers in a single step. Altogether, we therefore have costs $\mathcal{O}(w \cdot wr_0 r \cdot k(r_0 r)^2)$ which are incured at each of the control flow edges of the program to be analysed. □ □

Finding all valid polynomial invariants means to compute the precondition for a template with all terms up to degree $d = 1.5(w + k)$. We thus obtain $(1.5(w + k) + 1)^k$ as an upper bound for the number r_0 of terms to be considered in the postcondition. For input programs where the maximal degree of polynomials in assignments or disequalities is bounded by 2, the number r of occurring terms can be bounded by $(4.5w + k + 3)^k$. Summarising, we find that all polynomial invariants which are valid at a given program point can be inferred by an algorithm whose runtime is only exponential in k. This means that this algorithm is polynomial whenever the number of variables is bounded by a constant.

Example 22 . Consider the program `geo-1` *next to this paragraph which computes the geometrical sum. At program end, we obtain the invariant* $\mathbf{x} - \mathbf{y} + 1 = 0$ *as expected. Beyond that, we obtain the additional invariant* $2^{31}\mathbf{y} + 2^{31}\mathbf{x} = 0$ *which is valid over* $\mathbb{Z}_{2^{32}}$ *only. This invariant expresses that* $\mathbf{x}$ *and* $\mathbf{y}$ *are either both odd or both even at a specific program state.* □

```
1 int count = ?;
2 int x = 1, y = z;
3 while (count != 0){
4     count = count - 1;
5     x = x*z + 1;
6     y = z*y;
7 }
8 x = x*(z-1);
```

2.4.7 Implementation of modular analysis

Verification

We implemented our approach and evaluated it on selected benchmark programs, similar to the ones from [Pet04]. We considered the series of programs `power-`i which compute sums of $(i-1)$-th powers, i.e., the value $\mathbf{x} = \sum_{\mathbf{y}} \mathbf{y}^{i-1}$. In the case $i = 6$, for example, the invariant $12\mathbf{x} - 2\mathbf{y}^6 - 6\mathbf{y}^5 - 5\mathbf{y}^4 + \mathbf{y}^2$ could be verified for the end point of the program. Additionally, we considered programs `geo-`i for computing variants of the geometrical sum. An overview with verified invariants is shown in table 2.1. All these invariants could be verified instantly on a contemporary desktop computer with 2.4 GHz and 2GB of main memory.

Inferring all polynomials

We used a prototypical implementation of the presented approach for conducting a test series whose results on our 2,4 GHz 2 GB machine are shown in table 2.2. The

Name	Computation		verified invariant
power-1	$x_1 = \sum_{k=0}^{K} 1$	$x_2 = \sum_{k=0}^{K} 1$	$x_1 = x_2$
power-2	$x_1 = \sum_{k=0}^{K} k$	$x_2 = \sum_{k=0}^{K} 1$	$2x_1 = x_2^2 + x_2$
power-3	$x_1 = \sum_{k=0}^{K} k^2$	$x_2 = \sum_{k=0}^{K} 1$	$6x_1 = 2x_2^3 + 3x_2^2 + x_2$
power-4	$x_1 = \sum_{k=0}^{K} k^3$	$x_2 = \sum_{k=0}^{K} 1$	$4x_1 = x_2^4 + 2x_2^3 + x_2^2$
power-5	$x_1 = \sum_{k=0}^{K} k^4$	$x_2 = \sum_{k=0}^{K} 1$	$30x_1 = 6x_2^5 - 15x_2^4 - 10x_2^3 + x_2$
power-6	$x_1 = \sum_{k=0}^{K} k^5$	$x_2 = \sum_{k=0}^{K} 1$	$12x_1 = 2x_2^6 - 6x_2^5 - 5x_2^4 + x_2^2$
geo-1	$x_1 = (x_3 - 1) \sum_{k=0}^{K} x_3^k$	$x_2 = x_3^{K-1}$	$x_1 = x_2 + 1$
geo-2	$x_1 = \sum_{k=0}^{K} x_3^k$	$x_2 = x_3^{K-1}$	$x_1 \cdot (x_3 - 1) = x_2 x_3 - 1$
geo-3	$x_1 = \sum_{k=0}^{K} x_4 \cdot x_3^k$	$x_2 = x_3^{K-1}$	$x_1 \cdot (x_3 - 1) = x_4 x_3 x_2 - x_4$

Table 2.1: Test programs and verified invariants in $w = 32$

Name	inferred polynomial	time	space
power-1	$x_0 - x_1$	0.065 sec	51 MB
power-2	$x_1^2 - 2x_0 + x_1$	0.195 sec	63 MB
power-3	$2x_1^3 - 3x_1^2 - x_1 - 6x_0$, $2^{30}x_1^2 - 2^{31}x_0 + 2^{30}x_1$, $3 \cdot 2^{29}x_0x_1 + 15 \cdot 2^{27}x_1^2 - 5 \cdot 2^{28}x_0 + 15 \cdot 2^{27}x_1$, $3 \cdot 2^{28}x_0^2 - 25 \cdot 1^{26}x_0x_1 - 77 \cdot 2^{24}x_1^2 - 25 \cdot 2^{25}x_0 - 77 \cdot 2^{24}x_1$, $21 \cdot 2^{24}x_1^3 + 191 \cdot 2^{23}x_1^2 + 65 \cdot 2^{24}x_0 + 149 \cdot 2^{23}x_1$, $-19 \cdot 2^{26}x_0^3 + 2^{24}x_0^2x_1 - 235 \cdot 2^{22}x_0x_1^2 - 191 \cdot 2^{23}x_0x_1 + 57 \cdot 2^{25}x_1^2 - 27 \cdot 2^{26}x_0 + 37 \cdot 2^{25}x_1$	1.115 sec	89 MB
power-4	n.a.	>24 h	> 1 GB
geo-1	$x_0 - x_1 - 1$, $2^{31}x_1 + 2^{31}x_2$	0.064 sec	48 MB
geo-2	$2^{31}x_1x_2 + 2^{31}x_2$, $2^{31}x_1 + 2^{31}x_2$, $2^{28}x_1^2 + 230x_1x_2 - 7 \cdot 2^{28}x_2^2 - 3 \cdot 2^{29}x_2 + 2^{31}$, $x_0x_2 - x_1x_2 - x_0 + 1$	0.636 sec	65 MB
geo-3	*23 polynomials ...*	2.064 sec	96 MB

Table 2.2: Test programs and inferred invariants in $w = 32$

algorithm quickly terminates when inferring all invariants up to degree i for sums of powers of degree $i - 1$ for $i = 1, 2$ and 3 and also for the two variants of geometrical sums. Interestingly, it failed to terminate within reasonable time bounds for $i = 4$. In those cases when terminating, it inferred the invariants known from the analysis of polynomial relations over $\mathbb{Q}$ — and quite a few extra non-trivial invariants which could not be inferred before. It remains a challenge for future work to improve on our methods so that also more complicated programs such as e.g. `power-4` can be analysed precisely.

2.4.8 Summary and prospects

We have shown that verifying polynomial program invariants over $\mathbb{Z}_{2^w}$ is *PSPACE*-complete. By that, we have provided a clarification of the complexity of another analysis problem in the taxonomy of [MOR01]. Beyond the theoretical algorithm for the upper bound, we have provided a realistic method by means of normal-reduced generator sets. In case of constantly many variables, this algorithm runs in polynomial time. Indeed, our prototypical implementation was amazingly fast on all tested examples.

We extended the method for checking invariants to a method for inferring polynomial invariants of bounded degree — which in case of the ring $\mathbb{Z}_{2^w}$ also allows to infer all polynomial invariants. Beyond the vanishing polynomials, the algorithm finds further invariants over $\mathbb{Z}_{2^w}$, which would not be valid over the field $\mathbb{Q}$ and thus cannot be detected by any analyser over $\mathbb{Q}$. While still being polynomial for constantly many variables, our method turned out to be decently efficient only for small numbers of variables and low degree invariants. It remains for future work to improve on the method for inferring invariants in order to deal with larger numbers of variables and moderate degrees at least for certain meaningful examples.

Chapter 3

Interprocedural Herbrand equaltities

Analysing equalities between particular expressions has already been of interest in the early days of compiler development, since they enable optimising program transformations as e.g. eliminating redundant code. However, equality of the values of two variables is undecidable even in programs with non-deterministic branching, as shown by Reif and Lewis in [RL77]. This means for us, that we can only expect to compute subsets of valid equalities.

Herbrand interpreted terms are a representation of expressions with non-interpreted functions. In program analysis, they are quite commonly used for an abstract representation of values of arbitrary data structures. Two Herbrand interpreted terms are equal, iff their value was constructed by syntactically equal combinations of functions. Equalities between Herbrand terms are independent of the actual interpretation of the expressions, and thus allow machine-independent analyses and optimisations, where plain copy propagation or common subexpression approaches fail. Nonetheless they turn out to be quite useful in optimisation and transformation of program code, such as eliminating partially redundant code [RWZ88], safely performing code motion [KRS98] or applying strength reduction [SKR91].

An early idea to infer equalities of expressions, constructed via non-interpreted operators within basic blocks has been *value numbering*, developed by Cocke and Schwartz [CS69], who used hash values to represent computed expressions. Generalised to control flow graphs, Kildall [Kil73] presented his approach of *global value numbering* enabling the treatment of joins of basic blocks by intersections of partitions. His approach inspired a lot of other work to speed up his approach [AWZ88, CC95, Gar02, GN04, RL77, RKS99].

Variants of this approach were presented by Steffen [Ste87] and Steffen, Knoop and Rüthing in [SKR90], who showed the relation of global value numbering to Herbrand interpretation and introduced structured partition DAGs as compact representation of Herbrand equalities. Gulwani and Necula show in [GN04, GN07] an approach of representing Herbrand terms with DAGs which achieves polynomial runtime by imposing a size bound on the inferred equalities. Rüthing et al. improve this approach in [MORS05] by verifying positive boolean combinations (incl. disjunctions) of Herbrand equalities for programs with disequality guards in polynomial time. Problems arise when

procedure calls are added to the program class to analyse. Then intraprocedural methods for inferring invariants fail as the domain of Herbrand equalities is infinite. In [MOSS05] Müller-Olm, Seidl and Steffen deal with Herbrand interpreted programs which allow for local variables and call-by-value procedure calls with a dedicated global variable for the return value of a procedure. They present an analysis allowing to verify equalities between constants and variables in these programs. Additionally, they indicate how to interprocedurally infer all *Herbrand constants*, when branching is non-deterministic.

One approach in [GT07b] by Gulwani and Tiwari to verifying and inferring Herbrand equalities for program points consists in propagating the weakest preconditions of Herbrand equalities and performing unification to combine different program states. They also propose a redundancy test, determining whether further unification is necessary, in order to perform a fixpoint iteration on these program states. Inferring invariants this way imposes the use of so-called context variables. These context variables are unkown terms with holes, which apart from the holes do not contain program variables, but only ground symbols. Unification of these context variables is a form of second-order unification, which in general is undecidable [Gol81]. However, context-unification is a subclass of second-order unification, for which we do not know about decidability. In practice, there are several decidable subclasses for context-unification. Schmidt-Schauss and Schulz i.e. show in [SSS02] that the case of context unification with two context variables is decidable. In presence of only one single context variable, Schmidt-Schauss, Tiwari, et al. show in [GGSST10], that the unification problem is in NP, while the redundancy problem is in $co-NP$. In [GT09], Tiwari et al. introduce the class of *s*loopy programs which are programs without conditional branching, without mutual recursion and with no nested loops or procedure calls in loop bodies. For these sloopy programs, they show that checking the validity of assertions is decidable, by reducing the analysis to Presburger Arithmetics.

In this part of the thesis, we pursue the weakest precondition approach for analysing Herbrand equalities. At first we introduce the concepts for Herbrand interpretation and Herbrand equalities in general in section 3.1.1. With this in mind, we set up a general framework for checking the validity of Herbrand equalities in programs with procedure calls as an abstract interpretation of runs reaching a program point in section 3.1.2. Checking the validity of a Herbrand equality turns out to require an appropriate representation of a procedure's weakest precondition transformer. We thus check for decidable fragments, for which we can provide exact representations for the precondition transformers. We start with encoding Herbrand equalities as polynomial ideals in section 3.2.1. Furthermore, reducing Herbrand weakest precondition transformers to polynomial ideal transformers allows us to abstractly interpret Herbrand programs as polynomial programs in section 3.2.2 and bring the notion of procedure effects from the polynomial analysis into the Herbrand analysis in section 3.2.3. This approach is then studied, how it is suited for inferring Herbrand equalities in section 3.2.4. There, we come up with two properties for programs, which are suited well for inferring all Herbrand equalities at a program point with this approach.

As a second idea how to provide a representation for a procedure's weakest precondition transformer, we then present the case of *Herbrand constants* in section 3.3. We

exploit the fact, that here it suffices to record the effect of a transformer on limited equalities only. Finally, we present in section 3.4, how the general framework for verifying Herbrand equalities can be extended by introducing the concept of local variables. At last, we discuss the impact of the presented approaches in 3.5.

3.1 An introduction to Herbrand programs

3.1.1 Herbrand equalities in general

Herbrand interpreted programs are a very general way to look at programs. The main concept in Herbrand interpretation is to give no special interpretation to the occuring symbols other than their syntactical identity. The only evidence for their meaning is the syntactical structure, in which symbols occur in Herbrand terms. This interpretetation is called *Herbrand interpretation*.

Let Ω denote a signature consisting of a set Ω_0 of constant symbols and disjoint sets $\Omega_r, r > 0,$ of operator symbols of a particular rank r, each. For simplicity, we assume that the set Ω_0 is non-empty and that there is at least one operator of arity > 0. In examples, we write binary operators in *infix* notation, clarifying ambiguities via braces. Let $\mathcal{T}_\Omega$ be the set of all terms built up from Ω by means of finite recursive operator nesting: $f(t_1, \ldots, t_i)$ is a ground Herbrand term, if $f \in \Omega_i$ and each $t_1 \ldots t_i$ are again ground Herbrand terms. $\mathcal{T}_\Omega$ is called the set of *ground terms*. In this constellation, $\mathcal{T}_\Omega$ is infinite.

Example 23 . Let Ω be made up from $\Omega_0 = \{0, 1, 2\}$ and $\Omega_2 = \{\oplus, \ominus, \otimes\}$. Then, e.g. $(2 \ominus 1) \oplus (1 \otimes 2) \in \mathcal{T}_\Omega$

Next, we define a notion of Herbrand terms that may contain variables. Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ be this set of variables. General Herbrand terms are again constructed via operator nesting: Variables $\mathbf{x}_i$ are general Herbrand terms, as well as $f(t_1, \ldots, t_i)$ is a Herbrand term, if $f \in \Omega_i$ and each $t_1 \ldots t_i$ is again a Herbrand term. We assume that the variables can take ground Herbrand terms as values. The set of all general Herbrand terms with symbols from Ω and variables from $\mathbf{X}$ is denoted by $\mathcal{T}_\Omega(\mathbf{X})$. We want to reason about definitive equalities between Herbrand terms, thus we need an equality relation $\doteq$ on Herbrand terms. As with Herbrand interpretation we do not assume any knowledge about the implementation of an operator, equality can only be defined by syntactic equality $\equiv$.

Definition 1 . *Let $s, t \in \mathcal{T}_\Omega(\mathbf{X})$ be Herbrand terms. We define s to be* syntactically equal *to t, i.e. $s \equiv t$ recursively on the structure of the terms s, t. $s \equiv t$ iff*

- *$s, t \in \mathbf{X} \cup \Omega$ and s is the same symbol or variable as t.*
- *s and t are of the form $f(s_1, \ldots, s_i)$ and $g(t_1, \ldots, t_i)$, $f \equiv g$ and $\forall_{r=1}^{i}(s_r \equiv t_r)$.*

Based on this, our goal is to infer the set of all provably valid equalities between Herbrand terms. We may now consider conjunctions of Herbrand equalities. For a substitution $\sigma : \mathbf{X} \mapsto \mathcal{T}_\Omega$, we say, that it *satisfies* a conjunction of Herbrand equalities $E = \bigwedge_{r=1}^{i}(t_r \doteq t'_r)$, or E is valid for σ iff $\bigwedge_{r=1}^{i}(\sigma(t_r) \equiv \sigma(t'_r))$. We denote this with $\sigma \models E$. If (conjunctions of) Herbrand equalities E are valid for all $\sigma \in \Sigma \subseteq (\mathbf{X} \mapsto \mathcal{T}_\Omega(\mathbf{X}))$, we say that E is valid for Σ. Finally, E is a *tautology*, if it is valid for all

$\sigma \in \Sigma_\infty = \{\sigma \mid \sigma(\mathbf{x}_i) = h, h \in \mathcal{T}_\Omega\}$, while it is *satisfiable* iff there exists a substitution $\sigma \in \Sigma_\infty$, such that $\sigma \models E$. If a conjunction of Herbrand equalities is not satisfiable, it is equal to `false`.

Example 24 . Let $\Omega_0 = \{0, 1, 2\}, \Omega_2 = \{\oplus, \ominus, \otimes\}$ *and* $\mathbf{X} = \{\mathbf{x}_0, \mathbf{x}_1\}$.
Then $2 \oplus \mathbf{x}_1 \doteq 2 \oplus \mathbf{x}_1$ *is a tautologic Herbrand equality, while* $\mathbf{x}_1 \oplus 2 \doteq 2 \oplus \mathbf{x}_1$ *is satisfiable and* $\mathbf{x}_1 \oplus 2 \doteq 1 \oplus \mathbf{x}_1$ *is not satisfiable.*

Concerning conjunctions of Herbrand equalities, there are a few basic concepts worth to recall from [MOS05a]. Comparing conjunctions of Herbrand equalities is possible via the logical implication $\Rightarrow$, leading to a preorder. In this preorder `false` represents the smallest element, equivalent to all unsatisfiable conjuntions. The lower bound of two conjunctions of Herbrand equalities w.r.t. $\Rightarrow$ is the conjunction $\wedge$. For a satisfiable conjunction of Herbrand equalities E, a unifier σ is a substitution, such that E is valid for σ. A substitution $\sigma^\star$ is a most general unifier, if for every other unifier σ, there is another unifier σ', such that $\sigma = \sigma' \circ \sigma^\star$. E's most general general unifier is still not unique, thus we define the concept of a *reduced* conjunction of Herbrand equalities:

Definition 2 . *Let $\sigma^\star$ be a most general unifier of a conjunction of Herbrand equalities E. A conjunction of Herbrand equalities*

$$\phi = \bigwedge_i (\mathbf{x}_i \doteq \sigma^\star(\mathbf{x}_i))$$

is called the reduced *form of E, if $\phi \Rightarrow E$ and for each equality $\mathbf{x}_i \doteq \sigma^\star(\mathbf{x}_i)$ both $\mathbf{x}_i \notin \mathbf{Vars}(\sigma^\star(\mathbf{X}))$ and $j > i$ for each $\mathbf{x}_j \in \mathbf{Vars}(\sigma^\star(\mathbf{x}_i))$.*
The constraint on the indices of the variables on the left hand side of the equalities in the definition above immediately implies that

Lemma 11 . *Every reduced conjunction of equalities E consists of maximally k many equalities.* □

Reduced conjunctions of equalities can serve as normal forms for conjunctions of equalities, as they are unique:

Theorem 14 . *For every satisfiable conjunction of Herbrand equalities E, there is a unique reduced conjunction ϕ, with $\phi \Rightarrow E$.*
Proof. Let us first approach the uniqueness of reduced conjunctions of Herbrand equalities. Assume that there is a reduced conjunction $\phi' \neq \phi$ with $(\phi \Rightarrow E) \Leftrightarrow (\phi' \Rightarrow E)$. Let furthermore i be the minimal index, for which $\phi(\mathbf{x}_i) \neq \phi'(\mathbf{x}_i)$. Then let t and t' be the subterms of $\phi(\mathbf{x}_i)$ and $\phi'(\mathbf{x}_i)$ which do not match. t and t' must be either different operators, which contradicts that $(\phi \Rightarrow E) \Leftrightarrow (\phi' \Rightarrow E)$ or different free variables, as for non-free variables $\mathbf{x}_i$, the definition says $\mathbf{x}_i \notin \mathbf{Vars}(\sigma^\star(\mathbf{X}))$. However, different free variables lead to syntactically different Herbrand equalities, which must not be as both reduced conjunctions have to imply the same E. Thus we arrive at a contradiction and therefore $\phi \equiv \phi'$.

For every satisfiable conjunction of Herbrand equalities E we obtain an equisatisfiable system of equalities with only one variable on the left hand side by computing a most general unifier $\sigma^\star$ for E. If done naively, the unification algorithm suffers under

an exponential blowup of the costs for unification, as commented by Baader in [BS01]. Baader shows how to avoid this blowup by representing the Herbrand terms as *DAGs*.

To bring this most general unifier into a form, adhering to the properties in the definition, we start by creating a new substitution $\sigma_0 = \sigma^\star$. We successively modify the substitution σ_i to σ_{i+1} until we reach σ_k:

$$\sigma_i = \sigma_{i-1}[\sigma_{i-1}(\mathbf{x}_i)/\mathbf{x}_i]$$

As a σ satisfie the same equalities as an $\sigma[\sigma(\mathbf{x}_i)/\mathbf{x}_i]$ it follows that $\sigma_k \models E \Leftrightarrow \sigma^\star \models E$. Our reduced conjunction ϕ is now given by $\phi = \bigwedge_i(\mathbf{x}_i \doteq \sigma_k(\mathbf{x}_i))$. By construction, it fullfills that $j > i$ for each $\mathbf{x}_j \in \mathbf{Vars}(\sigma_k(\mathbf{x}_i))$ as well as that each variable, which occurs on a right-hand-side is not on any left-hand-side. □

Example 25 . Let $E = \{\mathbf{y} \doteq b(\mathbf{x}, \mathbf{z}), b(a(\mathbf{x}), a(\mathbf{x})) \doteq b(a(d), \mathbf{z})\}$ *A most general unifier for this set of equalities is* $\sigma^\star$*:*

$$\sigma^\star = \{\mathbf{x} \mapsto d, \mathbf{y} \mapsto b(\mathbf{x}, \mathbf{z}), \mathbf{z} \mapsto a(\mathbf{x})\}$$
$$\sigma_1 = \sigma_0[d/\mathbf{x}] = \{\mathbf{x} \mapsto d, \mathbf{y} \mapsto b(d, \mathbf{z}), \mathbf{z} \mapsto a(d)\}$$
$$\sigma_2 = \sigma_1[b(d, \mathbf{z})/\mathbf{y}] = \{\mathbf{x} \mapsto d, \mathbf{y} \mapsto b(d, \mathbf{z}), \mathbf{z} \mapsto a(d)\}$$
$$\sigma_3 = \sigma_2[a(d)/\mathbf{z}] = \{\mathbf{x} \mapsto d, \mathbf{y} \mapsto b(d, a(d)), \mathbf{z} \mapsto a(d)\}$$

We quickly check, that σ_3 *still satisfies* E*, and come to our* reduced conjunction

$$\phi = \{\mathbf{x} \doteq d, \mathbf{y} \doteq b(d, a(d)), \mathbf{z} \doteq a(d)\}$$

□

The empty conjunction `true` is implied by all reduced conjunctions, and is thus the largest element, when ordering via implication. Thus $\Rightarrow$ is a partial order for reduced conjunctions of Herbrand equalities. As a reduced conjunction can only imply another one, if it is either equal to it, or comprehends more equalities than the other, it turns out that descending chains w.r.t. $\Rightarrow$ have to be ultimately stable:

Proposition 2 . *[MOS05a] For every sequence* $\phi_0 \Leftarrow \ldots \Leftarrow \phi_m$ *of pairwise inequivalent reduced conjunctions* ϕ_i *using* k *variables,* $m \leq k + 1$*.*

Example 26 . Let Ω *be made up from* $\Omega_0 = \{0, 1, 2\}$ *and* $\Omega_2 = \{\oplus, \ominus, \otimes\}$*, while* $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$*.*

Consider $\mathbf{x}_3 \ominus \mathbf{x}_2 \doteq (2 \ominus \mathbf{x}_1) \oplus (\mathbf{x}_3 \otimes \mathbf{x}_1)$*. A most general unifier* $\sigma^\star$ *for this equality is:*

$$\sigma^\star(\mathbf{x}_2) = \mathbf{x}_3 \otimes \mathbf{x}_1, \sigma^\star(\mathbf{x}_3) = 2 \ominus \mathbf{x}_1$$

Then the reduced conjunction is

$$\mathbf{x}_3 \doteq 2 \ominus \mathbf{x}_1 \wedge \mathbf{x}_2 \doteq \mathbf{x}_3 \otimes \mathbf{x}_1$$

As such chains are finite, we get that all pairs of reduced conjunctions have the greatest lower bound $\wedge$. Thus, we obtain a complete lattice, made up of reduced conjunctions and the unsatisfiable conjunction `false`, as well as the implication $\Rightarrow$, which defines a partial order in this lattice. We denote this lattice by $\mathbb{E}$.

3.1.2 Herbrand programs

Herbrand programs are a very general way to model real world programs. They are more concrete than real programs in the sense that all properties, that are valid on Herbrand programs are also valid properties in real world programs. All aspects that can be observed with Herbrand programs relate to the sequence, in which data structures are constructed in the real program. Real world programs add interpretation to the operators, which are used in a program and thus less program states are considered as different than in a Herbrand interpreted program. Analysing Herbrand programs is sound in the sense that equalities found to be valid on the Herbrand program are also valid on the concrete program.

We assume that *Herbrand programs* are given as control flow graphs, as presented in section 2.2.1. The basic statements in Herbrand programs are assignments of the form $\mathbf{x}_j := t$, where the right hand side is a Herbrand term from $\mathcal{T}_\Omega(\mathbf{X})$, non-deterministic assignments $\mathbf{x}_j :=?$ and procedure calls $f()$. First, we consider all program variables from $\mathbf{X}$ to be globally accessible. Further, we assume that branching is non-deterministic in general.

An example of a Herbrand program is shown in Fig. 3.1.

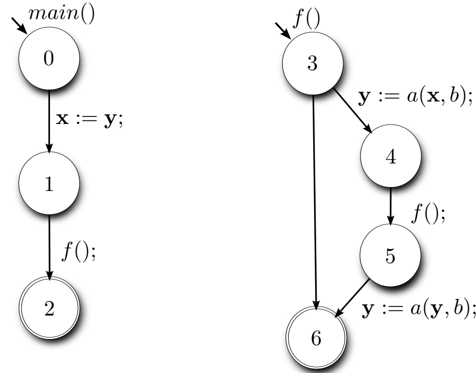


Figure 3.1: An interprocedural Herbrand program.

A program state of a Herbrand program is a ground substitution $\sigma : \mathbf{X} \mapsto \mathcal{T}_\Omega$, mapping all program variables $\mathbf{x}_i \in \mathbf{X}$ to ground terms from $\mathcal{T}_\Omega$. We specify the semantics of program statements as transformers of sets of ground substitutions:

$$\begin{aligned} [\![\mathbf{x}_i := t]\!]_\mathcal{H}\,\Sigma &= \{\sigma \oplus \{\mathbf{x}_i \mapsto \sigma(t)\} \mid \sigma \in \Sigma\} \\ [\![\mathbf{x}_i :=?]\!]_\mathcal{H}\,\Sigma &= \{[\![\mathbf{x}_i := t]\!]_\mathcal{H}\sigma \mid \sigma \in \Sigma, t \in \mathcal{T}_\Omega\} \end{aligned}$$

A procedure f transforms a set of program states before the call into a set of program states after the call. We define the transformer of a procedure call by means of the set of all those transformers for the executions which reach the procedure's end. We accumulate these transformers with the constraint system $\mathbf{R}$, starting from each procedure's end. The summarising transformer of a procedure f is then given as the interpretation of the

runs from the smallest solution of the constraint system $[\mathbf{R}]$ at the procedure's start state $\mathbf{R}_{\mathsf{r}_f}[\mathsf{st}_f]$ as state transformers $2^{\Sigma \mapsto \Sigma}$. Constraint system $\mathbf{R}$ is specified according to the following transfer functions, with the ordering $\supseteq$ on the functions being inherited from the ordering of the sets of mappings themselves:

$[\mathbf{R}1]$	$\mathbf{R}_t[t] \supseteq \mathbf{id}$	
$[\mathbf{R}2]$	$\mathbf{R}_t[u] \supseteq [\![s]\!]_{\mathcal{H}} \circ \mathbf{R}_t[v]$	for each $(u, s, v) \in E$
$[\mathbf{R}3]$	$\mathbf{R}_t[u] \supseteq \mathbf{R}_{\mathsf{r}_f}[\mathsf{st}_f] \circ \mathbf{R}_t[v]$	for each $(u, f(), v) \in E$

The semantics of a Herbrand program can then be defined via a set of constraints again: The least solution of constraint system $[\mathcal{C}_{\mathcal{H}}]$ characterises the collecting semantics of a Herbrand program. $\mathcal{C}_{\mathcal{H}}[t] = \{[\![r]\!]_{\mathcal{H}}\Sigma_\infty \mid r \in \mathbf{R}_t(\mathsf{st}_{main})\}$ i.e. $\mathcal{C}_{\mathcal{H}}[t]$ is the set of program states which are reachable when a program run beginning at the program start reaches t:

$[\mathcal{C}_{\mathcal{H}}1]$	$\mathcal{C}_{\mathcal{H}}[\mathsf{st}_{main}] \supseteq \Sigma_\infty$	at program start
$[\mathcal{C}_{\mathcal{H}}2]$	$\mathcal{C}_{\mathcal{H}}[v] \supseteq [\![s]\!]_{\mathcal{H}}\,\mathcal{C}_{\mathcal{H}}[u]$	for each $(u, s, v) \in E$
$[\mathcal{C}_{\mathcal{H}}3]$	$\mathcal{C}_{\mathcal{H}}[v] \supseteq [\![r]\!]_{\mathcal{H}}(\mathcal{C}_{\mathcal{H}}[u]) \mid r \in \mathbf{R}_{\mathsf{r}_f}[\mathsf{st}_f]$	for each $(u, f(), v) \in E$
$[\mathcal{C}_{\mathcal{H}}4]$	$\mathcal{C}_{\mathcal{H}}[\mathsf{st}_f] \supseteq \mathcal{C}_{\mathcal{H}}[u]$	for each $(u, f(), _) \in E$

Let us first straighten the relation between Herbrand semantics and real program semantics. Program semantics are usually established on domains like integer values or modular arithmetics. Let α be an interpretation of the operators and constants of $\mathcal{T}_\Omega$ over some ring $\mathcal{R}$. Then this interpretation is first extended to Herbrand terms and then componentwise to substitutions and sets of substitutions. The transformers $[\![]\!]_H$ can be interpreted likewise. We call such an α an *interpretation function* for Herbrand programs. As a matter of fact, we get for α that $\Sigma \models (s \doteq t) \Rightarrow \alpha(\Sigma) \models (\alpha(s) \doteq \alpha(t))$ for a $\Sigma \subseteq \mathbf{X} \mapsto \mathcal{T}_\Omega$. This yields:

Proposition 3 . *Let $\alpha : \mathcal{T}_\Omega \mapsto \mathcal{R}$ be an interpretation function. If $\mathcal{C}_{\mathcal{H}}[u] \models (s \doteq_{\mathcal{H}} t)$ then $\mathcal{C}[u] \models (\alpha(s) \doteq_{\mathcal{R}} \alpha(t))$.*

Example 27 . Let $\Sigma = \{\sigma\}$ with $\sigma = \{\mathbf{x}_1 \mapsto a(1, a(1,1)), \mathbf{x}_2 \mapsto a(a(1,1),1), \mathbf{x}_3 \mapsto a(a(1,1),1)\}$. Let α be an interpretation $\mathcal{T}_\Omega \mapsto \mathbb{N}$, such that the operator a corresponds to addition and the symbol 1 to the number 1. Then $\sigma \models (\mathbf{x}_2 \doteq \mathbf{x}_3)$. This implies, that also $\alpha(\sigma) \models (\mathbf{x}_2 \doteq \mathbf{x}_3)$ is valid. Furthermore, $\alpha(\sigma) \models (\mathbf{x}_1 \doteq \mathbf{x}_3)$, which could not be proven without interpretation.

Consequently, every invariant that we can prove to be valid with respect to the Herbrand semantics of a program is also valid with respect to the real concrete semantics of a program.

One way to verify the validity of a particular Herbrand equality at a certain program point is to examine the weakest precondition for this validity at program start. In order to do that, we set up Hoare-triples for each program statement, relating pre- with postconditions in form of Herbrand equalities. Let stm be the set of all possible

assignments. Then, we define the weakest precondition transformers $[\![\mathsf{stm}]\!]_{\mathcal{H}}^{\top} : 2^{\mathbb{E}} \mapsto 2^{\mathbb{E}}$ as follows:
Let $h \in \mathcal{T}_\Omega(\mathbf{X})$ and $E = \bigwedge_j (s_j \doteq t_j)$. Then

$$\begin{array}{rcl} [\![\mathbf{x}_i := h]\!]_{\mathcal{H}}^{\top} E & = & \bigwedge_j (s_j[h/\mathbf{x}_i] \doteq t_j[h/\mathbf{x}_i]) \\ [\![\mathbf{x}_i :=?]\!]_{\mathcal{H}}^{\top} E & = & \bigwedge_{h \in \mathcal{T}_\Omega} ([\![\mathbf{x}_i := h]\!]^{\top} E) = \begin{cases} E & \textbf{if } \mathbf{x}_i \notin \mathbf{Vars}(s_j, t_j) \\ \texttt{false} & \textbf{otherwise} \end{cases} \end{array}$$

These weakest precondition transformers on Herbrand equalities $[\![\mathsf{stm}]\!]_{\mathcal{H}}^{\top}$ actually are exactly the inversed interpretation of the postcondition transformers $[\![\mathsf{stm}]\!]_{\mathcal{H}}$ on the Herbrand semantics:

Lemma 12 . *Let $\Sigma \subseteq \mathbf{X} \mapsto \mathcal{T}_\Omega(\mathbf{X})$ and $E = \bigwedge_j (s_j \doteq t_j)$:*
(i) $\Sigma \models [\![\mathbf{x}_i := h]\!]_{\mathcal{H}}^{\top} E \quad \Leftrightarrow \quad [\![\mathbf{x}_i := h]\!]_{\mathcal{H}} \Sigma \models E.$
(ii) $\Sigma \models [\![\mathbf{x}_i :=?]\!]_{\mathcal{H}}^{\top} E \quad \Leftrightarrow \quad [\![\mathbf{x}_i :=?]\!]_{\mathcal{H}} \Sigma \models E.$

Proof. For (i), by definition we obtain:

$$\sigma \models (s \doteq t) \mid \sigma \in \Sigma, (s \doteq t) \in ([\![\mathbf{x}_i := h]\!]_{\mathcal{H}}^{\top} E) \ \Leftrightarrow$$

$$\sigma \models (s[h/\mathbf{x}_i] \doteq t[h/\mathbf{x}_i]) \mid \sigma \in \Sigma, (s \doteq t) \in E \ \Leftrightarrow$$

$$\sigma(s[h/\mathbf{x}_i]) \doteq \sigma(t[h/\mathbf{x}_i]) \mid \sigma \in \Sigma, (s \doteq t) \in E \ \Leftrightarrow$$

as $\sigma \oplus \{\mathbf{x}_i \mapsto \sigma(h)\} = \sigma \circ [h/\mathbf{x}_i]$:

$$\sigma(s) \doteq \sigma(t) \mid \sigma \in ([\![\mathbf{x}_i := h]\!]_{\mathcal{H}} \Sigma), (s \doteq t) \in E \Leftrightarrow$$

$$[\![\mathbf{x}_i := h]\!]_{\mathcal{H}} \Sigma \models E$$

which shows (i). □

For (ii), by definition we obtain:

$$\sigma \models (s \doteq t) \mid \sigma \in \Sigma, (s \doteq t) \in [\![\mathbf{x}_i :=?]\!]_{\mathcal{H}}^{\top} E \Leftrightarrow$$

$$\sigma \models \bigwedge_{h \in \mathcal{T}_\Omega} (s[h/\mathbf{x}_i] \doteq t[h/\mathbf{x}_i]) \mid \sigma \in \Sigma, (s \doteq t) \in E \ \Leftrightarrow$$

$$\bigwedge_{h \in \mathcal{T}_\Omega} (\sigma \models (s[h/\mathbf{x}_i] \doteq t[h/\mathbf{x}_i])) \mid \sigma \in \Sigma, (s \doteq t) \in E \Leftrightarrow$$

$$\bigwedge_{h \in \mathcal{T}_\Omega} (\sigma(s[h/\mathbf{x}_i]) \doteq \sigma(t[h/\mathbf{x}_i])) \mid \sigma \in \Sigma, (s \doteq t) \in E \Leftrightarrow$$

$$\sigma'(s) \doteq \sigma'(t) \mid \sigma' \in \bigcup_{h \in \mathcal{T}_\Omega} \sigma \oplus \{\mathbf{x}_i \mapsto h\} \mid \sigma \in \Sigma, (s \doteq t) \in E \Leftrightarrow$$

$$\sigma(s) \doteq \sigma(t) \mid \sigma \in ([\![\mathbf{x}_i :=?]\!]_{\mathcal{H}} \Sigma), (s \doteq t) \in E \Leftrightarrow$$

$$(\llbracket \mathbf{x}_i := ? \rrbracket_{\mathcal{H}} \Sigma) \models E$$

which shows (ii). □

We understand procedure calls as the least upper bound of the application of all transformers which correspond to runs from a procedure's start to its return point. In our framework, each of these runs corresponds to a transformer, which comprises the composition of the weakest precondition transformers of those statements, that are encountered during this run. We accumulate thus sets of weakest precondition transformers for procedures which reach the procedure's return point. We characterise each procedure f's set of weakest precondition transformers as the the least solution of the following constraint system $S^\top$ at program start st_f:

$$\begin{array}{lll} [\mathrm{S1}^\top] & S^\top[\mathsf{r}_f] \supseteq \{\mathbf{id}\} & \text{for each procedure } f \\ [\mathrm{S2}^\top] & S^\top[u] \supseteq \llbracket s \rrbracket_{\mathcal{H}}^\top \circ S^\top[v] & \text{for each } (u, s, v) \in E \\ [\mathrm{S3}^\top] & S^\top[u] \supseteq S^\top[\mathsf{st}_f] \circ S^\top[v] & \text{for each } (u, f(), v) \in E \end{array}$$

where $f \circ g$ means the concatenation of the WP tranformers f and g and $\mathbf{id}$ is the identity transformer. The least upper bound of two sets of transformers is defined as their union.

With these three components, we put up a constraint system $[\mathrm{C}^\top]$ on conjunctions of reduced Herbrand equalities, characterising for a program point u the weakest precondition of the validity of a Herbrand equality $h_1 \doteq h_2$ at a program point t with respect to all runs from u that reach t. Consider the abstraction α:

$$\alpha(R) = \bigwedge_{r \in R} (\llbracket r \rrbracket_{\mathcal{H}}^\top (h_1 \doteq h_2))$$

Note that α is obviously monotonic and distributive with arbitrary least upper bounds. Then, we can understand $\mathbf{C}^\top[u]$ as abstraction of the runs from u reaching t: $\mathbf{C}^\top[u](\alpha(\mathbf{R}_t(u)))$. As partial order $\supseteq$ of our lattice, we choose the implication $\Rightarrow$. Thus our least upper bound operator becomes the conjunction $\wedge$. All transfer functions commute with $\wedge$. The least solution (w.r.t. $\supseteq$) of this constraint system $\mathbf{C}^\top$ assigns to each program point u the set of preconditions for the validity of a special Herbrand equality at a particular program point t:

$$\begin{array}{lll} [\mathrm{C1}^\top] & \mathbf{C}^\top[t] \supseteq (h_1 \doteq h_2), & \text{the conjecture to verify at } t \\ [\mathrm{C2}^\top] & \mathbf{C}^\top[u] \supseteq \llbracket s \rrbracket_{\mathcal{H}}^\top \, \mathbf{C}^\top[v]\,, & \text{for each } (u, s, v) \in E \\ [\mathrm{C3}^\top] & \mathbf{C}^\top[u] \supseteq S^\top[\mathsf{st}_f]\mathbf{C}^\top[v]\,. & \text{for each } (u, f(), v) \in E \\ [\mathrm{C4}^\top] & \mathbf{C}^\top[u] \supseteq \mathbf{C}^\top[\mathsf{st}_f] & \text{for each } (u, f(), _) \in E \end{array}$$

By abuse of notation we denote the components of the least solution of the constraint system $\mathbf{C}^\top$ (which exists by Knaster-Tarski's fixpoint theorem and the fact that all transfer functions are monotone, or even distributive with least upper bounds) by $\mathbf{C}^\top[v]$, $v \in N$.

Even more, we can make sure: The conjunction of equalities $\mathbf{C}^\top[\mathsf{st}_{main}]$ precisely describes the weakest precondition of the validity of $(h_1 \doteq h_2)$ after arbitrary program runs up to program point t. We come to the following theorem:

Theorem 15 . *Let the Herbrand equality $h_1 \doteq h_2$ be the conjecture to verify at a program point t, i.e. $\mathbf{C}^\top[t] \supseteq (h_1 \doteq h_2)$. Then*

$$\mathcal{C}_\mathcal{H}[t] \models (h_1 \doteq h_2) \quad \textit{iff} \quad \Sigma_\infty \models \mathbf{C}^\top[\mathsf{st}_{main}]$$

Proof. As $\mathcal{C}_\mathcal{H}[\mathsf{st}_{main}] = \Sigma_\infty$, $\Sigma_\infty \mathbf{C}^\top[\mathsf{st}_{main}]$ iff $\mathcal{C}_\mathcal{H}[\mathsf{st}_{main}] \models \mathbf{C}^\top[\mathsf{st}_{main}]$.
By lemma 12, we get that $[\![f]\!]\Sigma \models E \Leftrightarrow \Sigma \models [\![f]\!]^\top E$, especially:

$$\mathcal{C}_\mathcal{H}[\mathsf{st}_{main}] \models \alpha(\mathbf{R}_t(\mathsf{st}_{main})) \Leftrightarrow \mathcal{C}_\mathcal{H}[t] \models \alpha(\mathbf{R}_t(t))$$

We now prove via the fixpoint transfer lemma (c.f. [Cou97, AP86]), that the least fixpoint of $[\mathrm{C}^\top]$ is exact, i.e. $\mathbf{C}^\top[\mathsf{st}_{main}] = \alpha(\mathbf{R}_t(\mathsf{st}_{main}))$. The precondition to turn this lemma applicable requires that α is monotonic and distributive for arbitrary least upper bound operators. Luckily, this is the case, as shown above. The further requirement for the fixpoint transfer lemma is, that the transfer functions are exact, which is shown by lemma 12.
As a result of this lemma, we get:

$$\mathcal{C}_\mathcal{H}[\mathsf{st}_{main}] \models \mathbf{C}^\top[\mathsf{st}_{main}] \Leftrightarrow \mathcal{C}_\mathcal{H}[t] \models \mathbf{C}^\top[t]$$

which is just another representation of the theorem, as $\mathcal{C}_\mathcal{H}[\mathsf{st}_{main}] = \Sigma_\infty$ and $\mathbf{C}^\top[t] = h_1 \doteq h_2$. □

Thus $\mathbf{C}^\top[\mathsf{st}_{main}]$ characterises the weakest precondition for the validity of a Herbrand equality at program start. Interestingly, this problem can be related to computing preconditions for the validity of polynomial equalities.

3.2 Analysing Herbrand equalities encoded as polynomials

3.2.1 Herbrand equalities as polynomial ideals

The first notable thing is, how Herbrand terms and Herbrand equalities can be coded as polynomials and polynomial equalities. Our first step consists in finding a relation between Herbrand terms and polynomials. Based upon this, we show the relation between reduced conjunctions of Herbrand equalities and polynomial ideals.

If symbols $\in \Omega$ all have a unique rank each, we regard ground terms as strings over the alphabet Ω, when written in prefix notation. In the first step, we introduce an interpretation $\|.\|$ to encode the structure of such strings as polynomials. Each particular symbol $s \in \Omega$ is interpreted as an integer $0 \leq \|s\| < \omega$, where ω is the cardinality of Ω. Thus, strings of symbols $s_1 s_2 \ldots s_j$ can be interpreted as an integer number:

$$\|s_1\| + \omega\|s_2\| + \omega^2\|s_3\| + \ldots + \omega^j\|s_j\|$$

We call these sums *ω-positional systems*. Note that they are composed of *digits* $\|.\| \leq \omega$ and their *weights*, powers of ω.

For general Herbrand terms with variables $\mathcal{T}_\Omega(\mathbf{X})$, a single natural number does not suffice any more to represent it. Especially, as the *weight* of a variable in the integral representation depends now on the term to be substituted for this variable, we need to introduce a new set of variables $\bar{\mathbf{X}} = \{\bar{\mathbf{x}}_1, \ldots \bar{\mathbf{x}}_k\}$, where $\bar{\mathbf{x}}_i$ represents the variable weight of $\mathbf{x}_i$. With these auxillary variables, it is possible to represent the weight of a whole Herbrand term $\in \mathcal{T}_\Omega(\mathbf{X})$ as a monomial in the variables $\subseteq \bar{\mathbf{X}}$ with an integral coefficient. For this purpose, we establish

Definition 3 . *The interpretation $|.| : \mathcal{T}_\Omega(\mathbf{X}) \mapsto \mathbb{Z}[\bar{\mathbf{X}}]$ maps a Herbrand term to a polynomial term representing its weight by an inductive mapping from its subterm structure:*

(i) $|\mathbf{x}_i| \mapsto \bar{\mathbf{x}}_i$
(ii) $|s| \mapsto \omega$
(iii) $|f(t_1, \ldots, t_i)| \mapsto \omega \cdot |t_1| \cdot \ldots \cdot |t_i|$ *with $f \in \Omega_i$ and t_j being Herbrand subterms*

Note, that the total degree $\deg(|s|)$ of the $|.|$-interpretation of a Herbrand term s yields the number of occurrences of program variables within s.

Example 28 . Let $\Omega_0 = \{0, 1\}$, $\Omega_2 = \{\oplus, \ominus\}$, $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2\}$ and $\|.\| = \{0 \mapsto 0, 1 \mapsto 3, \oplus \mapsto 1, \ominus \mapsto 2\}$; $\omega = 4$
Then $| \oplus \oplus \mathbf{x}_1 \ominus \mathbf{x}_2\ \mathbf{x}_1\ 1| = \omega \cdot \omega \cdot |\mathbf{x}_1| \cdot \omega \cdot |\mathbf{x}_2| \cdot |\mathbf{x}_1| \cdot \omega = 256 \cdot \bar{\mathbf{x}}_1^2 \cdot \bar{\mathbf{x}}_2$

The structure of a Herbrand term $\in \mathcal{T}_\Omega(\mathbf{X})$ with variables can be interpreted as a polynomial $\in \mathbb{Z}[\mathbf{X} \cup \bar{\mathbf{X}}]$ in both auxiliary and program variables. We extend the interpretation $\|.\|$, to map Herbrand terms with variables to polynomials inductively with the help of $|.|$:

Definition 4 . *The interpretation $\|.\| : \mathcal{T}_\Omega(\mathbf{X}) \mapsto \mathbb{Z}[\bar{\mathbf{X}} \cup \mathbf{X}]$ is defined inductively on the subterm structure of a Herbrand term:*

(i) $\|s\| \mapsto i$ *for $s \in \Omega$ and i the unique representation of s*
(ii) $\|\mathbf{x}_i\| \mapsto \mathbf{x}_i$
(iii) $\|t_1, t_2, \ldots, t_i\| \mapsto \|t_1\| + |t_1| \cdot \|t_2, \ldots, t_i\|$ *for sequences of operator arguments t_j*
(iv) $\|f(t_1, \ldots, t_i)\| \mapsto \|f\| + \omega \cdot \|t_1, \ldots, t_i\|$ *with $f \in \Omega_i$ and t_j Herbrand subterms*

Lemma 13 . *Let a, b be Herbrand terms from $\mathcal{T}_\Omega(\mathbf{X})$. Then*

$$|a| \doteq |b| \wedge \|a\| \doteq \|b\| \implies a \doteq b$$

Proof. We prove this property via contradiction. Let us assume that there is a pair of Herbrand terms a and b, for which $a \not\doteq b \wedge \|a\| \doteq \|b\| \wedge |a| \doteq |b|$ is true. Let t_a and t_b be the subterms of a and b, for which a structural comparison from the outermost term to the innermost terms yields non-equality. Then there arise several cases, in which t_a and t_b may be not equivalent:

- t_a is a variable, t_b a symbol. In this case, their $\|.\|$ image is different, as the images of symbols and variables are disjoint.
- t_a and t_b both are different variables. As $\|.\|$ acts as identity function for variables, $\|t_a\|$ and $\|t_b\|$ must also be different.

- t_a and t_b are terms, starting with different constructor symbols. Thus their $\|.\|$ images are also different.
- For non-equal compound terms t_a and t_b with the same constructor symbol, their numbers of parameters would also result in $|t_a| \neq |t_b|$.

In all cases, we have seen that $a \neq b \wedge \|a\| \doteq \|b\| \wedge |a| \doteq |b|$ does not evaluate to true, which completes our proof. □

These interpretations can be extended to range over substitutions of Herbrand terms, also:

Definition 5 . *For a substitution $\sigma : \mathbf{X} \mapsto \mathcal{T}_\Omega(\mathbf{X})$, we define*
- $\|\sigma\| = \{\mathbf{x}_i \mapsto \|\sigma(\mathbf{x}_i)\|\}$
- $|\sigma| = \{\bar{\mathbf{x}}_i \mapsto |\sigma(\mathbf{x}_i)|\}$

Before finally relating Herbrand equalities and polynomial equalities, let us first prove the following technical lemma, concerning the distributivity of subsitutions and the interpretations $|.|$ and $\|.\|$:

Lemma 14 . *Let t be a Herbrand term and $[t_r/\mathbf{x}_r]$ be a substution of variable $\mathbf{x}_r$ with the Herbrand term t_r. Then*
(i) $|t[t_r/\mathbf{x}_r]| = |t|[|t_r|/\bar{\mathbf{x}}_r]$
(ii) $\|t[t_r/\mathbf{x}_r]\| = \|t\|[\|t_r\|/\mathbf{x}_r, |t_r|/\bar{\mathbf{x}}_r]$

Proof. We proceed by induction over the structure of the term t. Let us start with (i) and (ii) together; for the induction start, we focus on the non-recursively defined components of the interpretation functions, such as a unary operator symbol s or a single variable $\mathbf{x}_i$, which can be either equal to $\mathbf{x}_r$ or not:

$$|\mathbf{x}_i[t_r/\mathbf{x}_r]| = |\mathbf{x}_i| = \bar{\mathbf{x}}_i = |\mathbf{x}_i|[|t_r|/\bar{\mathbf{x}}_r]$$
$$|\mathbf{x}_r[t_r/\mathbf{x}_r]| = |t_r| = \bar{\mathbf{x}}_r[|t_r|/\bar{\mathbf{x}}_r] = |\mathbf{x}_r|[|t_r|/\bar{\mathbf{x}}_r]$$
$$|s[t_r/\mathbf{x}_r]| = |s| = |s|[|t_r|/\bar{\mathbf{x}}_r]$$

analoguosly for (ii):

$$\|\mathbf{x}_i[t_r/\mathbf{x}_r]\| = \|\mathbf{x}_i\| = \mathbf{x}_i = \|\mathbf{x}_i\|[\|t_r\|/\mathbf{x}_r, |t_r|/\bar{\mathbf{x}}_r]$$
$$\|\mathbf{x}_r[t_r/\mathbf{x}_r]\| = \|t_r\| = \mathbf{x}_r[\|t_r\|/\mathbf{x}_r, |t_r|/\bar{\mathbf{x}}_r] = \|\mathbf{x}_r\|[\|t_r\|/\mathbf{x}_r, |t_r|/\bar{\mathbf{x}}_r]$$
$$\|s[t_r/\mathbf{x}_r]\| = \|s\| = \|s\|[\|t_r\|/\bar{\mathbf{x}}_r, |t_r|/\bar{\mathbf{x}}_r]$$

For the induction step of (i), we assume for the subterms t_j that $|t_j[t_r/\mathbf{x}_r]| = |t_j|[|t_r|/\bar{\mathbf{x}}_r]$:

$$|f(t_1, \ldots, t_i)[t_r/\mathbf{x}_r]| = |f(t_1[t_r/\mathbf{x}_r], \ldots, t_i[t_r/\mathbf{x}_r])| =$$
$$\omega \cdot |t_1[t_r/\mathbf{x}_r]| \cdot \ldots \cdot |t_i[t_r/\mathbf{x}_r]| =$$

by distributivity of substitution over multiplication and the induction hypothesis:

$$(\omega \cdot |t_1| \cdot \ldots \cdot |t_i|)[|t_r|/\mathbf{x}_r] =$$
$$|f(t_1, \ldots, t_i)|[|t_r|/\bar{\mathbf{x}}_r]$$

Now for the induction step of (ii), we assume for subterms t_j that $\|t_j[t_r/\mathbf{x}_r]\| = \|t_j\|[\|t_r\|/\mathbf{x}_r, |t_r|/\bar{\mathbf{x}}_r]$:
This time, we start with induction over the sequences of actual parameters:

$$\|t_1[t_r/\mathbf{x}_r], \ldots, t_i[t_r/\mathbf{x}_r]\| = \|t_1[t_r/\mathbf{x}_r]\| + |t_1[t_r/\mathbf{x}_r]| \cdot \|t_2[t_r/\mathbf{x}_r], \ldots, t_i[t_r/\mathbf{x}_r]\| =$$

by induction hypothesis, and this lemma's part (i) we get

$$\|t_1\|[\|t_r\|/\mathbf{x}_r, |t_r|/\bar{\mathbf{x}}_r] + |t_1|[\|t_r\|/\mathbf{x}_r, |t_r|/\bar{\mathbf{x}}_r] \cdot \|t_2[t_r/\mathbf{x}_r], \ldots, t_i[t_r/\mathbf{x}_r]\| =$$

by induction over the length of the sequence and the distributivity of substitutions over multiplication and addition, we get

$$\|t_1, \ldots, t_i\|[\|t_r\|/\mathbf{x}_r, |t_r|/\bar{\mathbf{x}}_r]$$

With this result, we can now also show (ii) for a complex term, consisting of an i-ary operator f and i subterms t_j:

$$\|f(t_1, \ldots, t_i)[t_r/\mathbf{x}_r]\| = \|f\| + \omega \cdot \|t_1[t_r/\mathbf{x}_r], \ldots, t_i[t_r/\mathbf{x}_r]\| =$$
$$\|f(t_1, \ldots, t_i)\|[\|t_r\|/\mathbf{x}_r, |t_r|/\bar{\mathbf{x}}_r]$$

□

With this property, we can finally relate the validity of Herbrand equalities and the validity of polynomial equalities:

Lemma 15 . *Let s, t be Herbrand Terms $\in \mathcal{T}_\Omega(\mathbf{X})$, and σ a substitution $\in \mathbf{X} \mapsto \mathcal{T}_\Omega(\mathbf{X})$. Then*

$$\sigma \models (s \doteq t) \quad \Leftrightarrow \quad (\|\sigma\|, |\sigma|) \models (\|s\| \doteq \|t\|) \wedge (|s| \doteq |t|)$$

Proof.

$$\sigma \models (s \doteq t) \Leftrightarrow$$
$$\sigma(s) \doteq \sigma(t) \Leftrightarrow$$

by lemma 13 we get:

$$(\|\sigma(s)\| \doteq \|\sigma(t)\|) \wedge (|\sigma(s)| \doteq |\sigma(t)|) \Leftrightarrow$$

Lemma 14 yields that substitutions of particular variables distribute under $\|.\|$ and $|.|$. This also holds for multiple substitutions, and thus:

$$((\|\sigma\|, |\sigma|)(\|s\|) \doteq (\|\sigma\|, |\sigma|)(\|t\|)) \wedge (|\sigma|(|s|) \doteq |\sigma|(|t|)) \Leftrightarrow$$

$$(\|\sigma\|, |\sigma|) \models \|(s \doteq t)\| \wedge |(s \doteq t)|$$

□

Thus for every $\|.\|$ and $|.|$ image of the solution of a Herbrand equality also solves the $\|.\|$ and $|.|$ images of the equalities, as e.g. in the following example:
Example 29 . Let $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$, $\Omega = \{0, 1, \oplus, \ominus\}$, $\omega = 4$ *and* $\mathbf{x}_3 \doteq (\mathbf{x}_1 \oplus (\mathbf{x}_2 \ominus \mathbf{x}_1)) \oplus 1$. *We start with the encoding* $\|0\| = 0, \|1\| = 3, \|\oplus\| = 1, \|\ominus\| = 2\}$*:*

$$\|\mathbf{x}_3 \doteq (\mathbf{x}_1 \oplus (\mathbf{x}_2 \ominus \mathbf{x}_1)) \oplus 1\|$$

$$\begin{gathered}
\mathbf{x}_3 \doteq \|\oplus\| + 4 \cdot (\|(\mathbf{x}_1 \oplus (\mathbf{x}_2 \ominus \mathbf{x}_1)\| + |\mathbf{x}_1 \oplus (\mathbf{x}_2 \ominus \mathbf{x}_1)| \cdot 3) \\
\mathbf{x}_3 \doteq \|\oplus\| + 4 \cdot (\|\oplus\| + 4 \cdot (\mathbf{x}_1 + \bar{\mathbf{x}}_1 \cdot \|\mathbf{x}_2 \ominus \mathbf{x}_1\|) + \bar{\mathbf{x}}_1^2 \bar{\mathbf{x}}_2 \cdot 48) \\
\mathbf{x}_3 \doteq 1 + 4 \cdot (\|\oplus\| + 4 \cdot (\mathbf{x}_1 + \bar{\mathbf{x}}_1 \cdot \|\mathbf{x}_2 \ominus \mathbf{x}_1\|) + \bar{\mathbf{x}}_1^2 \bar{\mathbf{x}}_2 \cdot 48) \\
\mathbf{x}_3 \doteq 1 + 4 \cdot (1 + 4 \cdot (\mathbf{x}_1 + \bar{\mathbf{x}}_1 \cdot (\|\ominus\| + 4(\mathbf{x}_2 + \bar{\mathbf{x}}_2 \mathbf{x}_1)) + \bar{\mathbf{x}}_1^2 \bar{\mathbf{x}}_2 \cdot 48) \\
= \mathbf{x}_3 \doteq 5 + 16\mathbf{x}_1 + 32\bar{\mathbf{x}}_1 + 64\bar{\mathbf{x}}_1\mathbf{x}_2 + 64\bar{\mathbf{x}}_1\bar{\mathbf{x}}_2\mathbf{x}_1 + 192\bar{\mathbf{x}}_1^2\bar{\mathbf{x}}_2
\end{gathered}$$

The polynomial equality, related to the length of the Herbrand equality is given by

$$|\mathbf{x}_3| \doteq |(\mathbf{x}_1 \oplus (\mathbf{x}_2 \ominus \mathbf{x}_1)) \oplus 1| \qquad = \qquad \bar{\mathbf{x}}_3 \doteq 256\bar{\mathbf{x}}_1^2\bar{\mathbf{x}}_2$$

□

The validity of a single Herbrand equality thus implies the validity of two polynomial equalities. Considering the analysis on Herbrand equalities from the last section, we move on to conjunctions of Herbrand equalities. Here, we find that conjunction commutes with substitution:

Lemma 16 . *Let* $s_i, t_i \in \mathcal{T}_\Omega(\mathbf{X})$ *and* $\sigma : \mathbf{X} \mapsto \mathcal{T}_\Omega(\mathbf{X})$*, then:*

$$\sigma \models \bigwedge_i (s_i \doteq t_i) \Leftrightarrow (\|\sigma\|, |\sigma|) \models \bigwedge_i (|s_i| \doteq |t_i|) \wedge (\|s_i\| \doteq \|t_i\|)$$

Proof.

$$\begin{gathered}
\sigma \models \bigwedge_i (s_i \doteq t_i) \Leftrightarrow \\
\bigwedge_i \sigma \models (s_i \doteq t_i) \Leftrightarrow \\
\bigwedge_i (\|\sigma\|, |\sigma|) \models (|s_i| \doteq |t_i|) \wedge (\|s_i\| \doteq \|t_i\|) \Leftrightarrow \\
(\|\sigma\|, |\sigma|) \models \bigwedge_i (|s_i| \doteq |t_i|) \wedge (\|s_i\| \doteq \|t_i\|)
\end{gathered}$$

□

Thus, statements about conjunctions of Herbrand equalities relate to statements on conjunctions of polynomials. The variable assignment under which a conjunction of polynomial equalities is valid can be described as the ideal of polynomials, which evaluate to zero under this assignment. This leads to the following:

Lemma 17 . *Let $s_i, t_i \in \mathcal{T}_\Omega(\mathbf{X})$, $\sigma : \mathbf{X} \mapsto \mathcal{T}_\Omega(\mathbf{X})$ and $I \in \mathbb{Z}[\mathbf{X} \cup \bar{\mathbf{X}}]$ be a polynomial ideal with $I = \langle \bigcup_i \{(|s_i| - |t_i|), (\|s_i\| - \|t_i\|)\} \rangle$, then:*

$$\sigma \models \bigwedge_i (s_i \doteq t_i) \;\Leftrightarrow\; (\|\sigma\|, |\sigma|) \models (p \doteq 0) \mid p \in I$$

Proof. We get by lemma 16 that

$$\sigma \models \bigwedge_i (s_i \doteq t_i) \Leftrightarrow \|\sigma\|, |\sigma| \models \bigwedge_i (|s_i| \doteq |t_i|) \wedge (\|s_i\| \doteq \|t_i\|)$$

$$\Leftrightarrow \|\sigma\|, |\sigma| \models \bigwedge_i (|s_i| - |t_i| \doteq 0) \wedge (\|s_i\| - \|t_i\| \doteq 0)$$

For all polynomials which are member of the ideal, it is known that they share the same roots as the generators of the ideal, which concludes our argument. □

We now set up an abstraction function α to map each conjunction of Herbrand equalities to the ideal of polynomials, whose solution set is a direct interpretation of the substitution that satisfies it:

Definition 6 . *We call $\alpha : \mathbb{E} \mapsto 2^{\mathbb{Z}[\mathbf{X}]}$ the abstraction function from Herbrand equalities to polynomial ideals:*

$$\alpha(\bigwedge_i s_i \doteq t_i) = \langle \{\|s_i\| - \|t_i\|, |s_i| - |t_i|\} \rangle$$

Note that, by construction of this correlation, the conjunction operator on Herbrand equalities corresponds to the union of the generator polynomials for ideals. Under α, $\wedge$ thus distributes with $\cup$.

Polynomial ideals are ordered via the subset relation $\subseteq$, with a pairwise least upper bound as the union of their generating polynomials $\cup$. The greatest lower bound $\wedge$ on conjunctions of Herbrand equalities commutes with the least upper bound $\cup$ under the abstraction relation α.

Lemma 18 . *A conjunction of Herbrand equalities $E \in \mathbb{E}$ is a tautology iff $\alpha(E) = \langle \{0\} \rangle$.*

Proof. Let $E = \bigwedge_i (s_i \doteq t_i)$ and I be the polynomial ideal with $\alpha(E) = I$. A tautologically valid conjunction of Herbrand equalities E is valid for all ground substitutions σ of variables with constant Herbrand terms $t \in \mathcal{T}_\Omega$. If $\models E$, i.e. $\models \bigwedge_i \sigma(s_i) \doteq \sigma(t_i)$ for arbitrary ground substitutions σ, then for $p \in I$ $p[(\|\sigma\|(\mathbf{x}_1))/\mathbf{x}_1, (|\sigma|(\bar{\mathbf{x}}_1))/\bar{\mathbf{x}}_1, \ldots, (\|\sigma\|(\mathbf{x}_k))/\mathbf{x}_k, (|\sigma|(\bar{\mathbf{x}}_k))/\bar{\mathbf{x}}_k] \doteq 0$ are valid. As for the substition σ's values there may occur arbitrarily nested operators around constants, we will obtain infinitely many constants $\mathbf{t}_i \in \mathcal{T}_\Omega$, which all are mapped to different integers. Therefore, p has to have infinitely many roots. The only polynomial, for which this is true over the integers is the polynomial 0, as mentioned in lemma 7. Thus I must be the zero ideal $\langle \{0\} \rangle$.

Vice versa, it is given that the zero ideal is valid for all assignments of program variables with concrete values. As $\|.\|, |.|$ is an injection from $\mathcal{T}_\Omega$ to $\mathbb{Z}$, the conjunction of Herbrand equalities E, with $\alpha(E) = \langle\{0\}\rangle$ also has to be valid for all ground substitutions. This holds in Herbrand interpretation only for syntactically equal terms, and thus E is equivalent to *true*. □

Thus, although we do not know how to compute the inverse of our representation function α, we can determine at least, whether the ideal at hand represents a tautologically valid Herbrand equalitiy or not. Considering, that for verifying Herbrand equalities via weakest precondition computation, we only need to check, whether the precondition at program start is statically *true* by lemma 18, polynomial ideals offer all necessary basic operations to serve as representation of conjunctions of Herbrand equalities. It remains to translate the transformations of Herbrand terms into transformations on the polynomial ideals.

3.2.2 Herbrand programs as polynomial programs

We lift the abstraction function α, to map transformers on Herbrand equalities to transformers on polynomial ideals. To make sure, that the representation as polynomials corresponds to the concrete $[\![\mathsf{stm}]\!]_{\mathcal{H}}^{\top}$ on Herbrand equalities, we show that $\|.\|$ and $|.|$ establish a correspondance between Herbrand equalities and polynomial ideals, w.r.t. the weakest precondition transformers $[\![\mathsf{stm}]\!]_{\mathcal{H}}^{\top}$ and their counterparts $[\![\mathsf{stm}]\!]^{-1}$ on polynomial ideals, as described in detail in section 2.3.3:

Lemma 19 . *Let β be $|.|$ or $\|.\|$, and $[\![\mathsf{stm}]\!]^{-1}$ be the weakest precondition transformer of* stm *w.r.t. polynomial ideals, then*
(i) $\beta([\![\mathbf{x}_i := t]\!]_{\mathcal{H}}^{\top}(t_1 \doteq t_2)) = [\![\mathbf{x}_i := \|t\|, \bar{\mathbf{x}}_i := |t|]\!]^{-1}\beta(t_1 \doteq t_2)$
(ii) $\beta([\![\mathbf{x}_i :=?]\!]_{\mathcal{H}}^{\top}(\mathbf{x}_j \doteq t)) = [\![\mathbf{x}_i :=?]\!]^{-1}\beta(\mathbf{x}_j \doteq t)$
Proof. We prove this lemma for different cases. Note that in **Case (ii)**, we restrict ourselves to postconditions of the form $\mathbf{x}_j = t$ which makes the prove easier and still suffices for our postcondition transformations on reduced conjunctions of Herbrand equalities.
Case (i).1: $\|[\![\mathbf{x}_i := t]\!]_{\mathcal{H}}^{\top}(t_1 \doteq t_2)\| = [\![\mathbf{x}_i := \|t\|, \bar{\mathbf{x}}_i := |t|]\!]^{-1}\|t_1 \doteq t_2\|$

$$[\![\mathbf{x}_i := \|t\|; \bar{\mathbf{x}}_i := |t|]\!]^{-1}\|(t_1 \doteq t_2)\| =$$

$$(\|t_1\| \doteq \|t_2\|)[\|t\|/\mathbf{x}_i, |t|/\bar{\mathbf{x}}_i] =$$

By lemma 14 (ii):

$$= \|(t_1 \doteq t_2)[t/\mathbf{x}_i]\| = \|[\![\mathbf{x}_i := t]\!]_{\mathcal{H}}^{T}(t_1 \doteq t_2)\| \qquad \square$$

Case (i).2: $|[\![\mathbf{x}_i := t]\!]_{\mathcal{H}}^{\top}(t_1 \doteq t_2)| = [\![\mathbf{x}_i := \|t\|, \bar{\mathbf{x}}_i := |t|]\!]^{-1}|t_1 \doteq t_2|$

$$|([\![\mathbf{x}_i := t]\!]_{\mathcal{H}}^{\top}(t_1 \doteq t_2))| = |(t_1 \doteq t_2)[t/\mathbf{x}_i]| =$$

By lemma 14 (i):

$$= (|t_1| \doteq |t_2|)[|t|/\bar{\mathbf{x}}_i] = |(t_1 \doteq t_2)|\ [|t|/\bar{\mathbf{x}}_i] =$$

Since $|(t_1 \doteq t_2)|$ only ranges over $\bar{\mathbf{X}}$, $\mathbf{x}_i$ does not occur:

$$|(t_1 \doteq t_2)|[\|t\|/\mathbf{x}_i, |t|/\bar{\mathbf{x}}_i] = [\![\mathbf{x}_i := \|t\|; \bar{\mathbf{x}}_i := |t|]\!]^{-1}|(t_1 \doteq t_2)| \qquad \Box$$

Case (ii).1: $\|[\![\mathbf{x}_i :=?]\!]^{\top}_{\mathcal{H}}(\mathbf{x}_j \doteq t')\| = [\![\mathbf{x}_i :=?]\!]^{-1}\|\mathbf{x}_j \doteq t'\|$ – we differentiate:
Case (ii).1.1: $\mathbf{x}_i \notin \mathbf{Vars}(t) \cup \{\mathbf{x}_j\}$:

$$\|\forall_{\mathbf{x}_i}.(\mathbf{x}_j \doteq t)\| = \|(\mathbf{x}_j \doteq t)\| = \forall_{\mathbf{x}_i,\bar{\mathbf{x}}_i}\|(\mathbf{x}_j \doteq t)\|$$

Case (ii).1.2: $\mathbf{x}_i \in \mathbf{Vars}(t) \cup \{\mathbf{x}_j\}$:
We have to show that $\|\forall_{\mathbf{x}_i}.(\mathbf{x}_j \doteq t)\| = \|\texttt{false}\| = (1 \doteq 0)$

$$\forall_{\mathbf{x}_i,\bar{\mathbf{x}}_i}.\|(\mathbf{x}_j \doteq t)\| = \forall_{\bar{\mathbf{x}}_i}.\|\mathbf{x}_j \doteq t\|[0/\mathbf{x}_i] \wedge \|\mathbf{x}_j \doteq t\|[1/\mathbf{x}_i]$$

since $\mathbf{x}_i$ occurs only linearly in $(\mathbf{x}_j \doteq t)$. From lemma 7, we know that a linear term is equal to the zero polynomial, if it has at least two consecutive roots.
Now, if $j = i$, we have $1 \doteq t \wedge 0 \doteq t$, which is `false`. Otherwise, i.e. $j \neq i$, $\mathbf{x}_i$ occurs (linearly) in $\|t\|$, and we can split the summands of $\|t\|$ in two: $\|t\| = p_0 + p_1 \cdot \mathbf{x}_i$ where p_i are two $\mathbf{x}_i$ free polynomials, with $p_1 \not\equiv 0$, as $\mathbf{x}_i$ occurs in t by assumption. This leads to:

$$= \forall_{\bar{\mathbf{x}}_i}.(\mathbf{x}_j \doteq p_0 + p_1 \cdot \mathbf{x}_i)[0/\mathbf{x}_i] \wedge (\mathbf{x}_j \doteq p_0 + p_1 \cdot \mathbf{x}_i)[1/\mathbf{x}_i] =$$

$$\forall_{\bar{\mathbf{x}}_i}.(p_1 \doteq 0)$$

which contradicts the assumption that $p_1 \not\equiv 0$, and closes case (ii).1. $\Box$

Case (ii).2: $|[\![\mathbf{x}_i :=?]\!]^{\top}_{\mathcal{H}}(\mathbf{x}_j \doteq t)| = [\![\mathbf{x}_i :=?]\!]^{-1}|\mathbf{x}_j \doteq t|$ – again, we differentiate:
Case (ii).2.1: $\mathbf{x}_i \notin \mathbf{Vars}(t) \cup \{\mathbf{x}_j\}$:

$$|\forall_{\mathbf{x}_i}.(\mathbf{x}_j \doteq t)| = |(\mathbf{x}_j \doteq t)| = \forall_{\bar{\mathbf{x}}_i}.|(\mathbf{x}_j \doteq t)|$$

Case (ii).2.2: $\mathbf{x}_i = \mathbf{x}_j$:

$$\forall_{\bar{\mathbf{x}}_i}.|(\mathbf{x}_i \doteq t)| = (\bar{\mathbf{x}}_i \doteq t)[1/\bar{\mathbf{x}}_i] \wedge (\bar{\mathbf{x}}_i \doteq t)[0/\bar{\mathbf{x}}_i] = (1 \doteq t) \wedge (0 \doteq t) = \texttt{false}$$

Case (ii).2.3: $\mathbf{x}_i \in \mathbf{Vars}(t)$:

$$\forall_{\bar{\mathbf{x}}_i}.|(\mathbf{x}_j \doteq t)| = \forall_{\bar{\mathbf{x}}_i}.(\bar{\mathbf{x}}_j \doteq |t|)$$

Since we know that $\mathbf{x}_i \in \mathbf{Vars}(t)$, we can extract a factor $\bar{\mathbf{x}}_i$ from $|t|$, leading to $|t'| \not\equiv 0$:

$$= \forall_{\bar{\mathbf{x}}_i}.(\bar{\mathbf{x}}_j \doteq \bar{\mathbf{x}}_i \cdot |t'|) = \bigwedge_i .(\bar{\mathbf{x}}_j \doteq i \cdot |t'|)$$

which implies `false`, and closes the case (ii).2. $\Box$

Finally, based on lemma 19 we extend α to transformers:
Definition 7 . *We extend α to map from transformers on Herbrand equalities to transformers on polynomial ideals:*

- $\alpha([\![\mathbf{x}_i := t]\!]^\top_{\mathcal{H}}) = [\![\mathbf{x}_i := \|t\|; \bar{\mathbf{x}}_i := |t|]\!]^{-1}$
- $\alpha([\![\mathbf{x}_i := ?]\!]^\top_{\mathcal{H}}) = [\![\mathbf{x}_i := ?]\!]^{-1}$

Example 30 . Let $p = 3 + 4\mathbf{x}_1 + 12\bar{\mathbf{x}}_1$ *and* $t = (\mathbf{x}_1 \oplus (\mathbf{x}_2 \ominus \mathbf{x}_1)) \oplus 1$*, with* $\omega = 4$*:*

$$\begin{aligned}\alpha([\![\mathbf{x}_1 := t]\!]^\top_{\mathcal{H}})\; p \;&= [\![\mathbf{x}_1 := \|t\|; \bar{\mathbf{x}}_1 := |t|]\!]^{-1} p = \\ p[|t|/\bar{\mathbf{x}}_1][\|t\|/\mathbf{x}_1] \;&= \\ p[\omega^4 \bar{\mathbf{x}}_1^2 \bar{\mathbf{x}}_2/\bar{\mathbf{x}}_1][\|(\mathbf{x}_1 \oplus (\mathbf{x}_2 \ominus \mathbf{x}_1)) \oplus 1\|/\mathbf{x}_1] \;&= \\ p[256\bar{\mathbf{x}}_1^2\bar{\mathbf{x}}_2/\bar{\mathbf{x}}_1][5 + 16\mathbf{x}_1 + 32\bar{\mathbf{x}}_1 + 64\bar{\mathbf{x}}_1\mathbf{x}_2 + 64\bar{\mathbf{x}}_1\bar{\mathbf{x}}_2\mathbf{x}_1 + 192\bar{\mathbf{x}}_1^2\bar{\mathbf{x}}_2/\mathbf{x}_1] &= \\ 20 + 64\mathbf{x}_1 + 128\bar{\mathbf{x}}_1 + 256\bar{\mathbf{x}}_1\mathbf{x}_2 + 256\bar{\mathbf{x}}_1\bar{\mathbf{x}}_2\mathbf{x}_1 + 3840\bar{\mathbf{x}}_1^2\bar{\mathbf{x}}_2&\end{aligned}$$

□

With this last component, we can now apply the abstraction function α to the whole constraint system $[\mathbf{C}]^\top$. This yields a system of weakest precondition transformers of polynomial ideals from $\mathbb{Z}[\mathbf{X} \cup \bar{\mathbf{X}}] \mapsto \mathbb{Z}[\mathbf{X} \cup \bar{\mathbf{X}}]$. Its greatest solution at the procedure start points characterises the weakest precondition transformers corresponding to calls to this procedure.

$[\mathrm{S1}]^\sharp$	$S^\sharp[\mathsf{r}_f] \supseteq \mathbf{id}$	for each procedure f
$[\mathrm{S2}]^\sharp$	$S^\sharp[u] \supseteq [\![\mathbf{x}_i := \|t\|; \bar{\mathbf{x}}_i := \|t\|]\!]^{-1} \circ S^\sharp[v]$	for each $(u, \mathbf{x}_i := t, v) \in E$
$[\mathrm{S3}]^\sharp$	$S^\sharp[u] \supseteq S^\sharp[\mathsf{st}_f] \circ S^\sharp[v]$	for each $(u, f(), v) \in E$

Likewise, we can apply α to $[\mathrm{C}]^\top$. The solution of the resulting constraint system represents an abstraction of the weakest preconditions for a Herbrand equality, expressed as polynomial ideals:

$[\mathrm{C1}]^\sharp$	$\mathbf{C}^\sharp[t] \subseteq \alpha(h_1 \doteq h_2)$,	the conjecture to verify at t
$[\mathrm{C2}]^\sharp$	$\mathbf{C}^\sharp[u] \subseteq [\![\mathbf{x}_i := \|t\|; \bar{\mathbf{x}}_i := \|t\|]\!]^{-1}\; \mathbf{C}^\sharp[v]$,	for each $(u, \mathbf{x}_i := t, v) \in E$
$[\mathrm{C3}]^\sharp$	$\mathbf{C}^\sharp[u] \subseteq S^\sharp[\mathsf{st}_f](\mathbf{C}^\sharp[v])$.	for each $(u, f(), v) \in E$

Theorem 16 . *The smallest solution of constraint systems* $[\mathrm{S}]^\sharp$ *and* $[\mathrm{C}]^\sharp$ *are exact abstractions of constraint systems* $[\mathrm{S}]^\top$ *and* $[\mathrm{C}]^\top$*, i.e.* $\alpha(\mathbf{C}^\top[u]) = \mathbf{C}^\sharp[u]$ *and* $\alpha(S^\top[u]) = S^\sharp[u]$*.*

Proof. In order to show for every program point u, $\alpha(\mathbf{C}^\top[u]) = \mathbf{C}^\sharp[u]$, as well as $\alpha(S^\top[u]) = S^\sharp[u]$, we apply the Transfer Lemma of general fixpoint theory [Cou97, AP86]. α is defined such that the least upper bound on polynomial ideals commutes with the greatest lower bound on conjunctions of Herbrand equalities. All the transfer functions on the right hand sides of the involved constraint systems are monotonic. We have shown in lemma 17 that the abstraction function α makes no overapproximation, applied to either reduced conjunctions of Herbrand equalities. Further, in lemma 19 we

showed that the transformers in $[S]^\sharp$ directly correspond to the transformers $[S]^\top$. The fact that α is $\wedge - \cup$-distributive completes the premises to apply the Transfer Lemma of general fixpoint theory, yielding that $[S]^\sharp$ and $[C]^\sharp$ are exact abstractions of constraint systems $[S]^\top$ and $[C]^\top$. □

The last two sections now have revealed, that computation of weakest preconditions of Herbrand equalities can be reduced to computing weakest preconditions of polynomial ideals. This leads to the idea of transfering the concepts of interprocedural analyses from polynomial ideals to Herbrand equalities.

3.2.3 Representing procedure effects on Herbrand equalities

Trying to compute the solution of the constraint systems $[S]^\sharp$ and $[C]^\sharp$ turns out to be problematic: The constraints of $[S]^\sharp$ are expressed as weakest precondition transformers of polynomial ideals. These in general do not have a concise representation, as infinitely many polynomials would have to be tabulated. Although we can characterise the weakest precondition transformers via the least solution of a constraint system, we can not compute it. In fact, we are confronted with the same problem as in sections 2.3.4ff: The problem of how to handle procedure calls precisely in Herbrand programs is in practice open.

Anyway, we can go similar ways as for analysing polynomials only, and i.e. only compute preconditions for a finite set of postconditions before giving up on procedure calls, as proposed in section 2.3.4. Anyway, following this approach is only suitable for procedures, that are not called so often, and thus only a solution for a minority of procedures.

Another idea for treating procedure calls, which we used in the approach for polynomial analysis, is to provide generic summary functions for procedure effects, which can be instantiated for each call site. For this, we proposed in section 2.3.6 to analyse the effect of a procedure by computing the precondition of a generic equality, and then use this relation to create a coefficient mapping for concrete equalities at each call site. This approach is capable of providing an exact procedure effect representation for concrete postconditions of up to a certain bound (the total degree in the case of polynomial analysis) and yields an overapproximated effect for all other postconditions. Now, that we deal with Herbrand equalities, we present in this section, how this approach can be applied for Herbrand equality analysis.

In order to provide an effective transformer for procedure call effects, we set up a modified constraint system S'_D which accumulates the weakest precondition transformer effects of a procedure on generic polynomial templates. The constraint system S'_D is intended to be parameterised with a polynomial template p_D, for which parametric preconditions are yielded by the constraint system S'_D. The procedure call effects are implemented by instantiating these parametric preconditions with the parameters, identified by matching the original template polynomial with the ideal representing the postcondition of the procedure call. Let $\mathbf{A}$ be a set of variables, disjoint from the program variables $\mathbf{X}$ or weight variables $\bar{\mathbf{X}}$ and D the set of exponent tuples $D \subseteq \mathbb{N}_0^{2k}$.

Recall from section 2.3.6, that then the template p_D is a polynomial over a finite set of monomials from $\bigcup_{(i_1,\ldots,i_{2k})\in D}(\mathbf{x}_1^{i_1}\ldots\mathbf{x}_{i_k}^{i_k}\bar{\mathbf{x}}_{i_1}^{i_{k+1}}\ldots\bar{\mathbf{x}}_{i_k}^{i_{2k}})$ and the template parameters from $\mathbf{A}$. In practice, we never instantiate polynomials with all possible monomials for Herbrand analyses, as for representing Herbrand terms, all monomials only feature a linear submonomial in the variables from $\mathbf{X}$. We denote the set of such polynomials with $\mathbb{Z}_{\mathbf{X}}^{\mathbf{A}}[\bar{\mathbf{X}}]$, each of which is of the form:

$$a_0 + a_1\mathbf{x}_{i_1} + a_2\mathbf{x}_{i_2}\bar{\mathbf{x}}_{i_1} + \ldots + a_n\mathbf{x}_{i_n}\bar{\mathbf{x}}_{i_1}\ldots\bar{\mathbf{x}}_{i_{n-1}} = a_0 + \sum_{j=1}^{n}(a_j\mathbf{x}_{i_j}\prod_{k=1}^{j-1}\bar{\mathbf{x}}_{i_k}) \in \mathbb{Z}_{\mathbf{X}}^{\mathbf{A}}[\bar{\mathbf{X}}]$$

Ideals of such polynomials are closed under the weakest precondition transformers that we presented in this section.

Considering a concrete procedure call site, it is our goal to define preconditions for each postcondition polynomial. Instead of providing preconditions for all polynomials in an ideal, it suffices for us to only compute preconditions for the generators of it. We denote the function, which provides us with a finite base of generators for an ideal I with $base(I)$. Assuming that monomials with exponents based on D are large enough to capture all the generators of the postcondition polynomial ideal at a procedure call site, we can define the precondition transformer for a procedure call to f as:

$$[\![f()]\!]^\sharp I \;=\; \langle\{q[a/\mathbf{a}] \mid p[a/\mathbf{a}] \in base(I), q \in S'[\mathsf{st}_f]\}\rangle$$

Substituting concrete values a for the template variables $\mathbf{a}$ instantiates the generic constraint system to one that is exactly tailored for the concrete polynomials p. In this case, the precomputed parametric precondition from $S'[\mathsf{st}_f]$ can be instantiated to the weakest precondition of the procedure call:

Lemma 20 . *Let $I \subseteq \mathbb{Z}_{\mathbf{X}}^{\mathbf{A}}[\bar{\mathbf{X}}]$ be an arbitrary ideal of polynomials and the polynomial $p_D \in \mathbb{Z}_{\mathbf{X}}^{\mathbf{A}}[\bar{\mathbf{X}}]$. If $T(q) \subseteq T(p_D)$ for each polynomial $q \in \mathbb{Z}_{\mathbf{X}}^{\mathbf{A}}[\bar{\mathbf{X}}]$ which occurs at any procedure call site during computation of S' and in I then $[\![f()]\!]^\sharp I \equiv S^\sharp[\mathsf{st}_f]I$.*

Unfortunately, due to the fact that each assignment with more than one single variable potentially enlarges the exponents of variables from $\bar{\mathbf{X}}$ in the dataflow value, one can not guarantee in general, that the monomials, of which a postcondition is composed are bounded in any way. Nevertheless, following the ideas from section 2.3.6 we can apply the operator $\mathbf{W}$ here to map an ideal, whose generators are composed of monomials which are not contained in those produced by the exponents from D to a larger ideal, whose generators are compatible to the exponent set D. In the context of analysing Herbrand equalities, not every exponent combination for the variables from $\mathbf{X} \cup \bar{\mathbf{X}}$ is really meaningfull. This is, because the encoding of Herbrand terms as polynomials produces only polynomial terms of a specific structure: Program variables from $\mathbf{X}$ contribute only linearly to the polynomial terms, from which a valid polynomial encoding of a Herbrand term is composed. However, as each program variable in a Herbrand term has a different specific position in the term, the width variables from $\bar{\mathbf{X}}$ may emerge in arbitrary degrees only bounded by the number of variables, which occured in the Herbrand term which they represent.

In particular $\mathbf{W}$ can be applied to enforce a bound the postconditions, for which the weakest precondition of procedure calls can be provided exactly by the sequence of program variables, that the postconditions are composed of. Applying $\mathbf{W}$ prior to the transformer for the procedure call decomposes postconditions, which consist of monomials other then from a particular set D into ideal-combinations of D-polynomials:

$$[\![f()]\!]^\sharp \mathbf{W}_D(I) \;=\; \langle\{q[a/\mathbf{a}] \mid \langle\{p_D[a/\mathbf{a}]\}\rangle \subseteq I, q \in S'[\mathsf{st}_f]\}\rangle$$

This yields a very important property of the approach of analysing the weakest preconditions of procedure calls w.r.t. Herbrand equalities via their encoding as polynomials:

Theorem 17 . *Let d be an integer constant and $I = \langle\{\|s_i\| - \|t_i\|, |s_i| - |t_i|\}\rangle = \alpha(\bigwedge_i s_i \doteq t_i)$ with $\mathsf{deg}(|s_i| \cdot |t_i|) \leq d$. For all such I there is a decomposition function $\mathbf{W}_D$, such that $[\![f()]\!]^\sharp I \equiv [\![f()]\!]^\sharp \mathbf{W}_D(I)$.*

Proof. If $\mathsf{deg}(|s|)$ is bounded by a constant d, then the number of occurrences of program variables is also bounded. Thus, the encoding functions $|.|, \|.\|$ may only produce polynomials from a finite set $\tilde{D}$ of monomials, in particular maximally $2 \cdot d + 2$. Choosing $\mathbf{W}_{\tilde{D}}$ to be a function, decomposing polynomials based on the monomials from $\tilde{D}$ thus is equivalent to the identity function on $|s|$ and $\|s\|$. □

In practice, this means that exact preconditions for procedure calls can be provided, if the postconditions for procedure calls are bounded in the number of occurrences of program variables. In particular, this also yields that the precision of the Herbrand equality analysis does not depend on the actual size of the Herbrand Terms, but merely on how many times variables occur within them. This permits the computation of preconditions for arbitrary length Herbrand equalities as long as their program variable occurrences remains bounded.

All that remains now is to set up the constraint system $[\mathrm{S}'_\mathrm{D}]$ which provides us with the preconditions of the generic D-polynomial for each procedure. We thus start with the ideal, generated by the generic polynomial at each procedure's return node, and set up constraints for their preconditions. Procedure calls are handled with the previously described combination of the abstraction function $\mathbf{W}$ and coefficient matching:

$[\mathrm{S1}'_\mathrm{D}]$	$S'_D[\mathsf{r}_f] \supseteq \langle p_D\rangle$	for each procedure f
$[\mathrm{S2}'_\mathrm{D}]$	$S'_D[u] \supseteq [\![\mathbf{x}_i := \|t\|; \bar{\mathbf{x}}_i := |t|]\!]^{-1} S'_D[v]$	for each $(u, \mathbf{x}_i := t, v) \in E$
$[\mathrm{S3}'_\mathrm{D}]$	$S'_D[u] \supseteq [\![f()]\!]^\sharp \mathbf{W}_D(S'_D[v])$	for each $(u, f(), v) \in E$

Introducing this concept for handling procedure calls in the constraint system $[\mathrm{C}]^\sharp$ leads to a slightly modified version $[\mathrm{C}'_\mathrm{D}]$, which describes the preconditions for the validity of a polynomial equality at a particular program point t. Note that $[\mathrm{C}_\mathrm{D}]'$ now is also parametric with respect to the set of monomials D:

$[\mathrm{C1}'_\mathrm{D}]$	$\mathbf{C}'_D[t] \subseteq \alpha(h_1 \doteq h_2)$,	the conjecture to verify at t
$[\mathrm{C2}'_\mathrm{D}]$	$\mathbf{C}'_D[u] \subseteq [\![\mathbf{x}_i := \|t\|; \bar{\mathbf{x}}_i := |t|]\!]^{-1}\ \mathbf{C}'_D[v]$,	for each $(u, \mathbf{x}_i := t, v) \in E$
$[\mathrm{C3}'_\mathrm{D}]$	$\mathbf{C}'_D[u] \subseteq [\![f()]\!]^\sharp \mathbf{W}_D(\mathbf{C}'_D[v])$.	for each $(u, f(), v) \in E$

Theorem 18 . *Let $D \subseteq \mathbb{N}_0^k$ be an arbitrary set of exponents, and $h_1 \doteq h_2$ the conjecture Herbrand equalitiy at program point t. If $[\mathrm{C}'_\mathrm{D}][\mathsf{st}_{main}] = \langle 0 \rangle$, then $h_1 \doteq h_2$ is valid at t.*
Proof. In lemma 19 we showed, that the transformers in $[\mathrm{S2}'_\mathrm{D}]$ and $[\mathrm{C2}'_\mathrm{D}]$ are equivalent to weakest precondition operators on Herbrand equalities. The abstraction function α is shown in lemma 17 to be an exact abstraction of conjunctions of Herbrand equalities. The function $\mathbf{W}$ at least provides sufficient preconditions for an ideal as shown by lemma 3, and together with lemma 20, we get that $[\mathrm{S3}'_\mathrm{D}]$ and $[\mathrm{C3}'_\mathrm{D}]$ also yield sufficient preconditions for their postconditions. Altogether, the smallest solution of the two constraint systems represent a sufficient precondition ideal for the validity of the conjecture Herbrand equality.

Lemma 18 finally enables us to reason about the validity of the conjecture from the fact whether the computed precondition ideal is equivalent to the zero ideal or not. □

Apart from statements based on the sufficient preconditions, we can extend this approach further – the proof of theorem 18 yields weakest preconditions, assuming that the constraint in the second part of lemma 3 is valid, i.e. all preconditions can be computed exactly by precomputed procedure effects:
Corollary 3 . *Let Q be the set of polynomials representing the postcondition of a callsite $f()$, and $T(q) \subseteq T(p_D)$ for all polynomials $q \in Q$ with $T(p)$ being the set of monomials of a polynomial p:*
Then $h_1 \doteq h_2$ is valid at t iff $[\mathrm{C}'_\mathrm{D}][\mathsf{st}_{main}] = \langle 0 \rangle$.

We have presented an approach which reduces checking the validity of conjunctions of Herbrand equalities to checking validity of polynomial identities. Having set up conditions, under which our approach yields exactly weakest preconditions, we continue with the aspect of inferring Herbrand equalities in the next section.

3.2.4 Interprocedurally inferring Herbrand equalities

Our next goal is to provide techniques for inferring valid Herbrand equalities at a particular program point. There has been interest in inferring this kind of invariants lately [SSS02, GT07b, GT07a, GT09, GGSST10]. Most of them approach the problem by directly computing weakest preconditions of generic Herbrand terms called context variables. In general, it is not known, whether the unification of such terms with context variables is decidable, but there is research going on by Schmidt-Schauss [SSS02] and Tivari [GGSST10] et al. for special cases of the context unification problem, which are decidable. In contrast, we present an other approach to this topic, which works without context-unification.

Apart from context-unification, there is also the idea from section 2.3.6, which consists in analysing the weakest precondition of a fixed template. Apart from the fact, that this template would again impose a bound on the shape of the inferred equalities, there is also the problem of solving the equation system which is yielded as precondition of the runs reaching t from program start. Solving these systems ordinarily over e.g. $\mathbb{Z}$ may not necessarily yield valid values for the generic parameters from $\mathbf{A}_D$, as they

originate from the encoding of a Herbrand equality as a polynomial. Instead of pursueing this approach, we rather come up with a different approach.

Computing candidates for invariants

In short terms, our approach consists in finding a particular program state $\sigma_t \in \mathbf{R}_t[\mathsf{st}_{main}]$ for a program point t. We then construct the set of concrete Herbrand equalities, which are satisfiable by σ_t. These can be generated from the Herbrand terms $\sigma_t(\mathbf{x}_j)$ for each program variable $\mathbf{x}_j$. All equalities, that we find are candidates, which must still be verified via weakest precondition computation. The next few paragraphs give a more detailed insight into this idea.

A program run is a path through the control flow graph, starting at program start, following the edges through the program's control flow graph. Procedure calls are considered as inlined by need. Whenever our run reaches a branch in the control flow graph, we have to make a choice, which branch to take. We decide, this choice by first looking, which of these branches ly on the path from the program start to t. If both of them lie on such a path but one of them leads to an already visited program point, we choose the other one, in order to avoid infinite looping. In any other case, we choose an arbitrary branch. All this leads to a single run through the control flow graph, reaching t.

In order to obtain a single program state, which is induced by such a program run at program point t, we apply the sequence of statements on an initial program state. Program states as defined in the Herbrand semantics consist in a mapping from program variables to Herbrand terms: $\Sigma : \mathbf{X} \mapsto \mathcal{T}_\Omega[\mathbf{X}]$. At program start, our concrete semantics defines the set of all possible states Σ_∞ to be valid. When computing a single state only, it suffices to take an arbitrary state σ_{start} from this set, in detail the state, that maps each variable to an arbitrary constant. We now apply the effect of the statements whose edges we pass on the way to t. When we are only interested in one of the states, which may be induced by the application of the Herbrand interpretation $[\![\mathsf{stm}]\!]_\mathcal{H}$ of our statements onto a single state σ, it suffices to choose an arbitrary state from the result of $[\![\mathsf{stm}]\!]_\mathcal{H}\{\sigma\}$. We thus define a may-effect on our statements according to their Herbrand semantics:

$$[\![\mathbf{x}_i := t]\!]\sigma \ni (\sigma \oplus \{\mathbf{x}_i \mapsto \sigma(t)\})$$

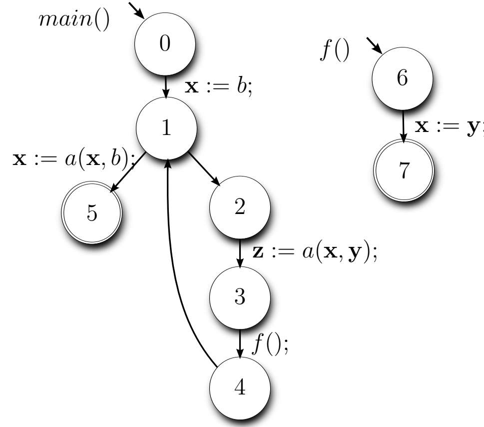
While for a deterministic assignment, the result is unambiguously another single program state, for non-determinstic assignments it is not. Anyway, as we only care for a single state, which may be induced by the statement, we can just pick an arbitrary one:

$$[\![\mathbf{x}_i :=?]\!]\sigma \ni (\sigma \oplus \{\mathbf{x}_i \mapsto a\})$$

where a is an arbitrary Herbrand constant.

Performing these interpretations of the program instructions successively on the initial assignment σ_{start} following the given run's trace finally yields a program state σ_t for t.

Example 31 . We are interested in a concrete program state for program point 4 *in the following example program:*



Let us start with $\sigma_{start} = \{\mathbf{x} \mapsto b, \mathbf{y} \mapsto b, \mathbf{z} \mapsto b\}$. *Starting from program point* 0*, we have to first apply the effect of an assignment:*

$$\sigma_1 = [\![\mathbf{x} := b]\!]\sigma_{start} = \{\mathbf{x} \mapsto b, \mathbf{y} \mapsto b, \mathbf{z} \mapsto b\}$$

For the branch, we choose to advance to 2 *and immidiately to* 3*, as* 5 *is not on the path to* 4*:*

$$\sigma_3 = [\![\mathbf{z} := a(\mathbf{x}, \mathbf{y})]\!]\sigma_1 = \{\mathbf{x} \mapsto b, \mathbf{y} \mapsto b, \mathbf{z} \mapsto a(b, b)\}$$

The call to function $f()$ *is inlined, leading to a single instruction* $\mathbf{x} := \mathbf{y}$*:*

$$\sigma_4 = [\![\mathbf{x} := \mathbf{y}]\!]\sigma_3 = \{\mathbf{x} \mapsto b, \mathbf{y} \mapsto b, \mathbf{z} \mapsto a(b, b)\}$$

So, we arrive at a possible program state for point 4*.* □

Now, that we have a particular program state σ_t for a program point, it is our goal to generate candidates for invariants. We know, that all valid Herbrand equalities at program point t must be valid for program state σ_t. So, we get:

$$C_{\mathcal{H}}[t] \subseteq \{(h \doteq h') \mid \sigma_t \models (h \doteq h')\}$$

Our goal is to verify particular invariant candidates by the means of the last section. Thus, it is crucial, to obtain a finite set of candidates, as compact as possible. So, based on the state σ_t, how can we obtain a finite set of equalities, which are valid for σ_t? Our idea is to apply the reverse mapping σ_t^{-1} transitively to the different subterms of the values of the program variables, in order to get candidates for invariants, as specified by the algorithm in figure 3.2.

Lemma 21 . *Let* $\sigma \in \mathbf{X} \mapsto \mathcal{T}_\Omega$*,* $t' \in \sigma(\mathbf{X})$ *and* $\mathbf{x}_i \doteq t \in \mathbb{E}$*. Then*

$$\sigma \models \mathbf{x}_i \doteq t \implies \sigma \models \mathbf{x}_i \doteq \tilde{t} \;\; \text{with} \;\; \tilde{t} \in t[\sigma^{-1}(t')//t']$$

Proof. In the case, that t' is a term, that does not occur in t, $t[\sigma^{-1}(t')//t']$ results in the singleton set $\{t\}$, for which our proposal holds.

```
algorithm CANDIDATES(σ_t)
forall (x_i ↦ h) ∈ σ_t do
   H ← H ∪ {(x_i ≐ h)}
   Q ← Q ∪ {(x_i ≐ h)}
od
while (Q ≠ ∅) do
    select (x_i ≐ h) ∈ Q
    Q ← Q \ {(x_i ≐ h)}
    if (∃_{t∈τ(h)}.t ∈ (σ_t(X))) do
        T ← h[σ_t^{-1}(t)//t]
        H ← H ∪ {(x_i ≐ h') | h' ∈ T}
        Q ← Q ∪ {(x_i ≐ h') | h' ∈ T}
    od
od
return H
```

where $\sigma_t(\mathbf{X})$ is the set of images of the set of inputs $\mathbf{X}$ under the function σ_t, and $\tau(h)$ is the set of subterms of a Herbrand term h. We denote with $t[a//b]$ the set of all terms, which result from substituting a for the particular occurrences of b in t.

Figure 3.2: CANDIDATES computes $H = \{(h \doteq h') \mid \sigma_t \models (h \doteq h')\}$

In the remaining case, t' is a subterm of t. Thus, one of its occurrences will be replaced by its preimage w.r.t. σ. Anyhow, as the definition of $\sigma \models \mathbf{x}_i \doteq t$ is $\sigma(\mathbf{x}_i) \doteq \sigma(t)$, this cancels the effect of the undertaken substitution, and thus the proposal follows. □

As the algorithm CANDIDATES from figure 3.2 extends the set H repeatedly only by equalities, produced by $[\sigma^{-1}(t)//t]$, we get by lemma 21:

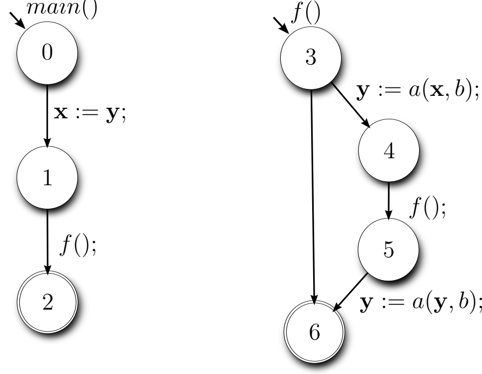
Lemma 22 . *Let H be the set of Herbrand equalities, returned by algorithm* CANDIDATES(σ_t) *in figure 3.2. Then $\sigma_t \models H$.* □

This algorithm produces finitely many equalities, as in each iteration of the algorithm's main loop, we extract one equality from Q and replace it with equalities which have less operator symbols. This can be done only finitely often. With this candidate generation, we have a sound way of inferring valid Herbrand equalities for particular program points:

Theorem 19 . *Let σ_t be a concrete program state at program point t. Then for all equalities $(h \doteq h')$ from* CANDIDATES(σ_t) *with $C'_D[\mathsf{st}_{main}] = \langle 0 \rangle$ we get that $\mathcal{C}_\mathcal{H}[t] \models (h \doteq h')$.*

Proof. If $\sigma_t \in \mathcal{C}_\mathcal{H}[t]$ then the set of all equalities H which are implied by $\mathcal{C}_\mathcal{H}[t]$ also have to be implied by σ_t. Then, considering lemma 22, we get that $H \subseteq$ CANDIDATES(σ_t). Theorem 18 then provides that $\mathcal{C}_\mathcal{H}[t] \models (h \doteq h')$ if $C'_D[\mathsf{st}_{main}] = \langle 0 \rangle$. □

Example 32 . Let us examine this whole approach in an example. We are interested in inferring the invariants for program point 6 *in the following program:*



For an encoding of our Herbrand semantics, we choose $\|a\| = 1, \|b\| = 2, \|c\| = 0$ *and* $\omega = 3$.

First of all, we start to compute a reachable program state for program point 6*. Therefore, we start with the arbitrary program state* $\sigma_{start} = \{\mathbf{x} \mapsto c, \mathbf{y} \mapsto b\}$

$$\sigma_1 = [\![\mathbf{x} := \mathbf{y}]\!]\sigma_{start} = \{\mathbf{x} \mapsto b, \mathbf{y} \mapsto b\}$$

Now, inlining procedure $f()$ *and following the skip-edge to* 6*, we get*

$$\sigma_6 = \sigma_3 = \sigma_1 = \{\mathbf{x} \mapsto b, \mathbf{y} \mapsto b\}$$

Algorithm CANDIDATES(σ_6) *yields the two invariant candidates* $\mathbf{x} \doteq \mathbf{y}$ *and* $\mathbf{y} \doteq \mathbf{x}$*. For our example, let's choose* $\mathbf{x} \doteq \mathbf{y}$ *for analysis.*

Verifying the validity of these candidates, we first need to choose a set of monomials D *and then compute the solution of the constraint system* $[\mathrm{S}'_{\mathrm{D}}]$*. We choose* $D = \{\mathbf{x}, \mathbf{y}, \bar{\mathbf{x}}, \bar{\mathbf{y}}\}$*. The instruction* $[\![\mathbf{y} := a(\mathbf{y}, b);]\!]^\sharp$ *results in a transformer* $[\![\mathbf{y} := 1 + 3\mathbf{y} + 6\bar{\mathbf{y}}; \bar{\mathbf{y}} := 9\bar{\mathbf{y}};]\!]^{-1}$ *on polynomial ideals.*

We start to compute the precondition for procedure $f()$ *for the polynomial*

$$d + e\mathbf{x} + f\mathbf{y} + g\bar{\mathbf{x}} + h\bar{\mathbf{y}}$$

with the parameter variables $\{d, e, f, g, h\}$.

$$S'_D[3] \supseteq S'_D[6] \subseteq \{d + e\mathbf{x} + f\mathbf{y} + g\bar{\mathbf{x}} + h\bar{\mathbf{y}}\}$$

$$S'_D[5] \supseteq [\![\mathbf{y} := a(\mathbf{y}, b);]\!]^\sharp S'_D[6] = \{d + e\mathbf{x} + f(1 + 3\mathbf{y} + 6\bar{\mathbf{y}}) + g\bar{\mathbf{x}} + h(9\bar{\mathbf{y}})\}$$

This leads to the evaluation of the recursive procedure call to $f()$*, iteratively refining* $S'_D[3]$*:*

$$S'_D[3] \supseteq [\![\mathbf{x} := a(\mathbf{x}, b);]\!]^\sharp [\![f]\!]^\sharp \mathbf{W}(S'_D[5])$$

which stabilizes after the re-evaluation to

$$\begin{aligned} S'_D[5] \quad = \langle \quad & d + e\mathbf{x} + f\mathbf{y} + g\bar{\mathbf{x}} + h\bar{\mathbf{y}} \quad , \\ & (e+g)\bar{\mathbf{x}} + (f+h)\bar{\mathbf{y}} \quad , \\ & e + f - 2d \quad \rangle \end{aligned}$$

Now, that we have a stable value for $S'_D[5]$, we can start to compute the smallest solution for $C'_D[\mathsf{st}]$. We start with $C'_D[6] = \alpha(\mathbf{x} \doteq \mathbf{y}) = \langle\{\mathbf{y} - \mathbf{x}, \bar{\mathbf{y}} - \bar{\mathbf{x}}\}\rangle$. We evaluate the respective transformers:

$$C'_D[5] \supseteq [\![\mathbf{y} := a(\mathbf{y}, b);]\!]^\sharp C'_D[6] = \langle\{1 + 3\mathbf{y} + 6\bar{\mathbf{y}} - \mathbf{x}, 9\bar{\mathbf{y}} - \bar{\mathbf{x}}\}\rangle$$

$$C'_D[4] \supseteq [\![f();]\!]^\sharp \mathbf{W}(C'_D[5]) = \langle\{1 + 3\mathbf{y} + 6\bar{\mathbf{y}} - \mathbf{x}, 9\bar{\mathbf{y}} - \bar{\mathbf{x}}\}\rangle$$

For $C'_D[3]$, we have to evaluate two constraints:

$$C'_D[3] \supseteq [\![\mathbf{x} := a(\mathbf{x}, b);]\!]^\sharp C'_D[4] = \langle\{\mathbf{y} - \mathbf{x}, \bar{\mathbf{y}} - \bar{\mathbf{x}}\}\rangle$$

$$C'_D[3] \supseteq C'_D[6] = \langle\{\mathbf{y} - \mathbf{x}, \bar{\mathbf{y}} - \bar{\mathbf{x}}\}\rangle$$

This results in a stable upper bound for $C'_D[3]$:

$$C'_D[3] = \langle\{\mathbf{y} - \mathbf{x}, \bar{\mathbf{y}} - \bar{\mathbf{x}}\}\rangle$$

This now gives rise to the evaluation of the constraint system $[\mathrm{C'_D}]$ for the caller $main$:

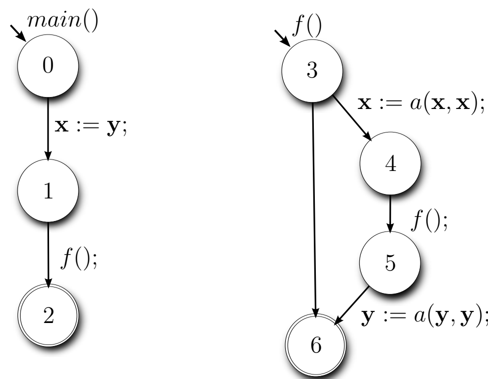
$$C'_D[0] = [\![\mathbf{x} := \mathbf{y};]\!]^\sharp C'_D[1] = \langle\{\mathbf{y} - \mathbf{y}, \bar{\mathbf{y}} - \bar{\mathbf{y}}\}\rangle = \{0\}$$

Thus, we get, that $\mathbf{x} \doteq \mathbf{y}$ is a valid invariant at program point 6. □

Inferring all Herbrand equalities

Although we have shown in the last section a way to infer valid Herbrand equalities, we have no statement about the completeness of this approach yet. In fact, there are examples for Herbrand programs, for which the approach is not able to yield all valid Herbrand equalities:

Example 33 . Let us have a look at a slightly modified version of the program from example 32. Again, we expect the equality $\mathbf{x} \doteq \mathbf{y}$ to be valid for program point 6:



This time, the approach fails at computing an exact representation of the procedure effect for $f()$ for any polynomial template. At start, everything is still ok:

$$S'_D[3] \supseteq S'_D[6] \subseteq \{d + e\mathbf{x} + f\mathbf{y} + g\bar{\mathbf{x}} + h\bar{\mathbf{y}}\}$$

$$S'_D[5] \supseteq [\![\mathbf{y} := a(\mathbf{y},\mathbf{y});]\!]^\sharp S'_D[6] = \{d + e\mathbf{x} + f(1 + 3\mathbf{y} + 3\bar{\mathbf{y}}\mathbf{y}) + g\bar{\mathbf{x}} + h(3\bar{\mathbf{y}}^2)\}$$

Here, we can see the problem: the transformer for $[\![\mathbf{y} := a(\mathbf{y},\mathbf{y});]\!]^\sharp$ generates polynomials of a degree higher than the orignal template, no matter which template we choose. This way, $\mathbf{W}$ enlarges the ideal for $S'_D[5]$ before applying $[\![f]\!]^\sharp$. In this case, we decompose by the term $\bar{\mathbf{y}}$:

$$\mathbf{W}(S'_D[5]) = \{d + e\mathbf{x} + f(1 + 3\mathbf{y}) + g\bar{\mathbf{x}},\ 3f\mathbf{y},\ 3h\}$$

The recursive call to $f()$ leads then to a second constraint for $S'_D[3]$:

$$S'_D[3] \supseteq [\![\mathbf{x} := a(\mathbf{x},\mathbf{x});]\!]^\sharp [\![f]\!]^\sharp \mathbf{W}(S'_D[5])$$

which finally stabilises as:

$$S'_D[3] \;=\; \langle d + e\mathbf{x} + g\bar{\mathbf{x}},\ (d+g)\bar{\mathbf{x}},\ e - 2d,\ f,\ h\rangle$$

Now, computing the weakest precondition of the equality $\mathbf{x} \doteq \mathbf{y}$, we obtain the following:

$$C'_D[5] \supseteq [\![\mathbf{y} := a(\mathbf{y},\mathbf{y});]\!]^\sharp C'_D[6] = \langle\{1 + 3\mathbf{y} + 3\bar{\mathbf{y}}\mathbf{y} - \mathbf{x}, 3\bar{\mathbf{y}}^2 - \bar{\mathbf{x}}\}\rangle$$

$$\mathbf{W}(C'_D[5]) = \langle 1\rangle$$

This is the top element of the lattice, which is propagated back to program start:

$$C'_D[0] = C'_D[1] = C'_D[3] = C'_D[4] = [\![f();]\!]^\sharp \mathbf{W}(C'_D[5]) = \langle 1\rangle$$

As we cannot show that $C'_D[0] \subseteq \langle 1\rangle$, we cannot show that $\mathbf{x} \doteq \mathbf{y}$ for program point 6.□

Essentially, the fact that we had to apply $\mathbf{W}$ to decompose the generators of the ideals generated slightly stronger preconditions, causing the verification process to lose too much information. Programs for which this decomposition is not necessary, can be analysed for all valid Herbrand equalities:

Theorem 20 . *Let Q be the set of generator polynomials, representing the postcondition of a callsite $f()$, and D be a finite fixed set of monomials, such that $T(q) \subseteq T(p_D)$ for each $q \in Q$ during the verification of a particular Herbrand equality.*
Then, we can infer all valid Herbrand equalities for a program point t.
Proof. If all postconditions of procedure calls are representable as polynomials with monomials from a fixed finite set D, then by corollary 3 we can decide whether a particular Herbrand equality is valid for a program point t. As by lemma 22 `CANDIDATES(t)` yields a finite superset of all valid Herbrand equalities for t, the invalid candidates can be filtered by proving their invalidity by fixpointiteration. □

However, theorem 20 provides us with a semantic property to decide, whether all valid equalities can be inferred, only. We need to have a more constructive property to characterise a program class, for which all valid Herbrand equalities can be inferred. Looking back at the last example, we see that the growth of degrees of the postconditions occur at specific assignments. E.g. $[\![\mathbf{y} := a(\mathbf{y}, \mathbf{y});]\!]^\sharp$ results in a transformer $[\![\mathbf{y} := 1 + 3\mathbf{y} + 3\bar{\mathbf{y}}\mathbf{y}; \bar{\mathbf{y}} := 3\bar{\mathbf{y}}^2;]\!]^{-1}$ - which leads to a substitution, which increases the degree for all polynomial terms which were composed of either $\mathbf{y}$ or $\bar{\mathbf{y}}$. The only assignments which lead to transformers which never increase the total degree of a polynomial are linear assignments. Interpreted as Herbrand program instructions, this represents assignments which do not have more than one occurrence of program variables on their right hand side. As the total degree of the generator polynomial in these programs can not increase, we can choose D such that it covers all monomials up to the total degree of the largest monomial in the conjecture. This disposes the need to apply $\mathbf{W}$ for any set of generators in the analysis, and thus leads to precise preconditions. Finally, we arrive at a syntactical program class, which can be analysed for all valid Herbrand equalities:

Theorem 21 . *Let P be a Herbrand program with procedure calls, whose assignment instructions have not more than one program variable on the right hand side. Then P can be analysed for all valid Herbrand equalities at each program point t.* □

Example 34 . Let us consider the program from example 32 again. Every assignment, which occurs in this program has exactly one program variable on the right hand side. We thus can be sure, that we do not miss any valid Herbrand equality with our approach.

3.3 Herbrand constants

Reducing the expressiveness of the representations of weakest precondition transformers seems to turn the problem of finding succinct representations for procedure effects more easy. One way of reducing the expressiveness of weakest precondition transformers consists in regarding the transformers from section 3.1.2 only with the goal of verifying or inferring *Herbrand constants*, i.e. equalities between program variables and ground Herbrand terms e.g. $\mathbf{x}_i \doteq a(b)$. This topic has been discussed by Müller-Olm, Seidl and Steffen, who show in [MOSS05] a particular analysis of *Herbrand constants*. They indicate, that their approach may yield in presence of procedure calls *all* Herbrand constants. Our goal in this subsection is to finally provide a solid approach to interprocedurally infer *all Herbrand constants* by instantiating our framework from section 3.1.2 appropriately.

Let us recall the form of weakest preconditions, which are obtained by our weakest precondition transformers $[\![\mathsf{stm}]\!]^\top_{\mathcal{H}}$ under the aspect of preconditions for the constancy of a particular variable $\mathbf{x}_i$. Therefore, we introduce an auxillary variable $\bullet$, disjoint from the program variables $\mathbf{X}$, which is meant to hold an arbitrary Herbrand constant:

$$\llbracket \mathbf{x}_j := t \rrbracket^{\top}_{\mathcal{H}} (\mathbf{x}_i \doteq \bullet) \quad = \quad \begin{cases} t \doteq \bullet & \text{if } i = j \\ \mathbf{x}_i \doteq \bullet & \text{otherwise} \end{cases}$$

$$\llbracket \mathbf{x}_j := ? \rrbracket^{\top}_{\mathcal{H}} (\mathbf{x}_i \doteq \bullet) \quad = \quad \begin{cases} \texttt{false} & \text{if } i = j \\ \mathbf{x}_i \doteq \bullet & \text{otherwise} \end{cases}$$

As these implementations are directly taken from the general weakest precondition transformer's definitions, we get the following specialisation from lemma 12:

Corollary 4 . *Let f be one of the program statements $\mathbf{x}_i := t;$ or $\mathbf{x}_i :=?;$. Then*

$$\llbracket f \rrbracket_{\mathcal{H}} \Sigma \models (\mathbf{x}_i \doteq t) \Leftrightarrow \Sigma \models \llbracket f \rrbracket^{\top}_{\mathcal{H}} (\mathbf{x}_i \doteq t)$$

We observe, that the weakest precondition transformers produce Herbrand equalities of the form $(t \doteq \bullet)$ with $t \in \mathcal{T}_\Omega(\mathbf{X})$. The least upper bound $\sqcup$ of two weakest precondition transformers $\llbracket f \rrbracket^{\top}$ and $\llbracket g \rrbracket^{\top}$ is their componentwise conjunction:

$$\llbracket f \sqcup g \rrbracket^{\top} (\mathbf{x}_i \doteq \bullet) = \llbracket f \rrbracket^{\top} (\mathbf{x}_i \doteq \bullet) \wedge \llbracket g \rrbracket^{\top} (\mathbf{x}_i \doteq \bullet)$$

When accumulating precondition transformers via constraint systems, each join node produces the least upper bound of two dataflow values. Concerning weakest precondition computations, this corresponds to conjunctions of preconditions. Thus we have to deal with mappings from constant equalities to conjunctions of equalities. We represent them by means of reduced conjunctions of Herbrand equalities $E \subseteq \mathbb{E}$, i.e. of the form $E = (t \doteq \bullet) \wedge \bigwedge_i (t_i \doteq t'_i)$ for $t, t_i, t'_i \in \mathcal{T}_\Omega(\mathbf{X})$. Recall that in section 3.1.1 we introduced reduced conjunctions of Herbrand equalities as normal form for conjunctions of equalities. In this normal form, we rely on the fact, that the conjunctions consist in maximally $k+1$ many equalities. Thus, when only tabulating preconditions for constant equalities $\mathbf{x}_i \doteq \bullet$, we end up with maximally $k \cdot (k+1)$ many equalities to represent the weakest precondition transformer.

What we also learned in section 3.1.1 is that reduced Herbrand equalities are partially ordered via implication. The componentwise application of this partial order yields a partial order for mappings of constant equalities to reduced Herbrand equalities:

$$\llbracket f \rrbracket^{\top} \subseteq \llbracket g \rrbracket^{\top} \qquad \Leftrightarrow \qquad \forall_i . \llbracket f \rrbracket^{\top} (\mathbf{x}_i \doteq \bullet) \Rightarrow \llbracket g \rrbracket^{\top} (\mathbf{x}_i \doteq \bullet)$$

This order creates a lattice where the least element is a mapping $\mathbf{x}_i \; \llbracket \bot \rrbracket^{\top} (\mathbf{x}_i \doteq \bullet) = \texttt{false}$ for every variable $\mathbf{x}_i$, respectively a greatest mapping $\llbracket \top \rrbracket^{\top} (\mathbf{x}_i \doteq \bullet) = \texttt{true}$.

The first notable point is now, that we can effectively compute the composition of two such variable constant equality precondition transformers. First, we have to introduce a technical lemma, extending the transformers from mapping only constant equalities $\mathbf{x}_i \doteq \bullet$ to their preconditions to mapping arbitrary Herbrand equalities $t \doteq t'$ to preconditions:

Definition 8 . *For a transformer f and each variable $\mathbf{x}_i$, let Φ^i_f be a reduced conjunction of $\bullet$-free Herbrand equalites with $\llbracket f \rrbracket^{\top} (\mathbf{x}_i \doteq \bullet) = \Phi^i_f \wedge (s_i \doteq \bullet)$. Then*

$$\llbracket f \rrbracket^{\top} (t \doteq t') = \begin{cases} \bot & \textbf{if } \exists_i . \llbracket f \rrbracket^{\top} (\mathbf{x}_i \doteq \bullet) = \bot \\ (\bigwedge_{i=1}^{k} \Phi^i_f) \wedge (t \doteq t')[s_1/\mathbf{x}_1, \ldots, s_k/\mathbf{x}_k] & \textbf{otherwise} \end{cases}$$

Lemma 23 . *Let Σ be a set of substitutions and $[\![f]\!]^\top(\mathbf{x}_i \doteq \bullet) = \Phi_f^i \wedge (s_i \doteq \bullet)$, with Φ_f^i a reduced conjunction of $\bullet$-free Herbrand Equalities. Then*

$$\Sigma \models [\![f]\!]^\top(t \doteq t') \quad \Leftrightarrow \quad [\![f]\!]\Sigma \models (t \doteq t')$$

Proof. Let us first look at the case, where $f()$; does not terminate. We get:

$$\Sigma \models [\![f]\!]^\top(t \doteq t') \Leftrightarrow \Sigma \models \bot \Leftrightarrow \texttt{true} \Leftrightarrow \emptyset \models (t \doteq t') \Leftrightarrow [\![f]\!]\Sigma \models (t \doteq t')$$

Now, let us look at the case, where $[\![f]\!]^\top(\mathbf{x}_i \doteq \bullet) = \Phi_f^i \wedge (s_i \doteq \bullet)$ for all i:

$$\Sigma \models (t \doteq t')[s_1/\mathbf{x}_1, \ldots, s_k/\mathbf{x}_k] \wedge \bigwedge_{i=1}^{k} \Phi_f^i \Leftrightarrow$$

$$\Sigma \models (t \doteq t')[\bullet_1/\mathbf{x}_1, \ldots, \bullet_k/\mathbf{x}_k] \wedge \bigwedge_{i=1}^{k} (s_i \doteq \bullet_i \wedge \Phi_f^i) \Leftrightarrow$$

Substitutions distribute over conjunctions:

$$\Sigma \models (t \doteq t')[\bullet_1/\mathbf{x}_1, \ldots, \bullet_k/\mathbf{x}_k] \wedge \bigwedge_{i=1}^{k} (\Sigma \models (s_i \doteq \bullet_i \wedge \Phi_f^i)) \Leftrightarrow$$

$$\Sigma \models (t \doteq t')[\bullet_1/\mathbf{x}_1, \ldots, \bullet_k/\mathbf{x}_k] \wedge \bigwedge_{i=1}^{k} (\Sigma \models [\![f]\!]^\top(\mathbf{x}_i \doteq \bullet_i)) \Leftrightarrow$$

$$\Sigma \models (t \doteq t')[\bullet_1/\mathbf{x}_1, \ldots, \bullet_k/\mathbf{x}_k] \wedge \bigwedge_{i=1}^{k} ([\![f]\!]\Sigma \models (\mathbf{x}_i \doteq \bullet_i)) \Leftrightarrow$$

Since in $(t \doteq t')[\bullet_1/\mathbf{x}_1, \ldots, \bullet_k/\mathbf{x}_k]$ all program variables are substituted by some $\bullet_i$, $f()$ has no effect:

$$[\![f]\!]\Sigma \models (t \doteq t')[\bullet_1/\mathbf{x}_1, \ldots, \bullet_k/\mathbf{x}_k] \wedge \bigwedge_{i=1}^{k} ([\![f]\!]\Sigma \models (\mathbf{x}_i \doteq \bullet_i)) \Leftrightarrow$$

$$[\![f]\!]\Sigma \models \left((t \doteq t')[\bullet_1/\mathbf{x}_1, \ldots, \bullet_k/\mathbf{x}_k] \wedge \bigwedge_{i=1}^{k} (\mathbf{x}_i \doteq \bullet_i) \right) \Leftrightarrow$$

$$[\![f]\!]\Sigma \models (t \doteq t')$$

□

Note that this yields particularly $[\![f]\!]^\top(t \doteq \bullet) = (\bigwedge_{i=1}^{k} \Phi_f^i) \wedge (t \doteq \bullet)[s_1/\mathbf{x}_1, \ldots, s_k/\mathbf{x}_k]$. Further, we know from lemma 11 that although $\bigwedge_{i=1}^{k} \Phi_f^i$ gives rise to k^2 many equalities,

we can reduce them to k many when re-establishing the reduced conjunction of Herbrand equalities.

Definition 9 . *Let the precondition transformer of g be $[\![g]\!]^\top(\mathbf{x}_i \doteq \bullet) = \bigwedge_j(\phi_j^i) \wedge (s_i \doteq \bullet)$, with $\bigwedge_j(\phi_j^i)$ a reduced conjunction of $\bullet$-free Herbrand Equalities ϕ_j^i. Then the composed precondition transformer of f and g is*

$$[\![f]\!]^\top \circ [\![g]\!]^\top(\mathbf{x}_i \doteq \bullet) \quad = \begin{cases} \bot & \textbf{if } \exists_i \,.\, [\![g]\!]^\top(\mathbf{x}_i \doteq \bullet) = \bot \\ ([\![f]\!]^\top(s_i \doteq \bullet)) \wedge \bigwedge_j([\![f]\!]^\top \phi_j^i) & \textbf{otherwise} \end{cases}$$

Lemma 24 . *Let $\Sigma \models [\![g]\!]^\top(\mathbf{x}_i \doteq \bullet) \Leftrightarrow [\![g]\!]\Sigma \models (\mathbf{x}_i \doteq \bullet)$ and $\Sigma \models [\![f]\!]^\top(\mathbf{x}_i \doteq \bullet) \Leftrightarrow [\![f]\!]\Sigma \models (\mathbf{x}_i \doteq \bullet)$. Then*

$$\Sigma \models [\![f]\!]^\top \circ [\![g]\!]^\top(\mathbf{x}_i \doteq \bullet) \quad \Leftrightarrow \quad [\![f;g]\!]\Sigma \models (\mathbf{x} \doteq \bullet)$$

Proof. Let us first look at the case, where $g()$; does not terminate. We get:

$$\Sigma \models [\![f]\!]^\top \circ [\![g]\!]^\top(t \doteq t') \Leftrightarrow \Sigma \models \bot \Leftrightarrow \texttt{true} \Leftrightarrow \emptyset \models (t \doteq t') \Leftrightarrow [\![f;g]\!]\Sigma \models (t \doteq t')$$

Now, let us look at the case, where $[\![g]\!]^\top(\mathbf{x}_i \doteq \bullet) = (s_i \doteq \bullet) \wedge \bigwedge_{j=1}^k(\phi_j^i)$, with $\bigwedge_{j=1}^k(\phi_j^i)$ a reduced conjunction of $\bullet$-free Herbrand Equalities for all i:

$$\Sigma \models [\![f]\!]^\top \circ [\![g]\!]^\top(\mathbf{x}_i \doteq \bullet) \Leftrightarrow$$

$$\Sigma \models [\![f]\!]^\top((s_i \doteq \bullet) \wedge \bigwedge_{j=1}^k [\![f]\!]^\top(\phi_j^i)) \Leftrightarrow$$

Since substitution is distributive over conjunctions:

$$\Sigma \models [\![f]\!]^\top(s_i \doteq \bullet) \wedge \bigwedge_{j=1}^k(\Sigma \models [\![f]\!]^\top(\phi_j^i)) \Leftrightarrow$$

By lemma 23 we get:

$$[\![f]\!]\Sigma \models (s_i \doteq \bullet) \wedge \bigwedge_{j=1}^k([\![f]\!]\Sigma \models (\phi_j^i)) \Leftrightarrow$$

Since substitution is distributive over conjunctions:

$$[\![f]\!]\Sigma \models \left((s_i \doteq \bullet) \wedge \bigwedge_{j=1}^k(\phi_j^i)\right) \Leftrightarrow$$

By definition of $[\![g]\!]^\top$

$$[\![f]\!]\Sigma \models \left([\![g]\!]^\top(\mathbf{x}_i \doteq \bullet)\right) \Leftrightarrow$$

By the precondition from the lemma, we get

$$[\![f;g]\!]\Sigma \models (\mathbf{x}_i \doteq \bullet)$$

This is what we had to show. □

Example 35 . Let us first have a look at a simple composition:
We start with an assignment:

$$[\![\mathbf{x}_1 := a(\mathbf{x}_1,\mathbf{x}_2);]\!]^{\top}_{\mathcal{H}} = \{(\mathbf{x}_1 \doteq \bullet) \mapsto (a(\mathbf{x}_1,\mathbf{x}_2) \doteq \bullet), (\mathbf{x}_2 \doteq \bullet) \mapsto (\mathbf{x}_2 \doteq \bullet)\}$$

$$\begin{aligned} [\![\mathbf{x}_1 := b;]\!]^{\top}_{\mathcal{H}} \circ [\![\mathbf{x}_1 := a(\mathbf{x}_1,\mathbf{x}_2);]\!]^{\top}_{\mathcal{H}} &= \begin{array}{ll} \{ & (\mathbf{x}_1 \doteq \bullet) \mapsto [\![\mathbf{x}_1 := b;]\!]^{\top}_{\mathcal{H}}(a(\mathbf{x}_1,\mathbf{x}_2) \doteq \bullet), \\ & (\mathbf{x}_2 \doteq \bullet) \mapsto [\![\mathbf{x}_1 := b;]\!]^{\top}_{\mathcal{H}}(\mathbf{x}_2 \doteq \bullet)\} \end{array} \\ &= \begin{array}{ll} \{ & (\mathbf{x}_1 \doteq \bullet) \mapsto (a(b,\mathbf{x}_2) \doteq \bullet), \\ & (\mathbf{x}_2 \doteq \bullet) \mapsto (\mathbf{x}_2 \doteq \bullet)\} \end{array} \end{aligned}$$

A more complex example comes here. Consider the following two WP transformers:

$$[\![f]\!]^{\top} = \{(\mathbf{x}_1 \doteq \bullet) \mapsto (a(b,b) \doteq \bullet) \wedge (b \doteq \mathbf{x}_2), (\mathbf{x}_2 \doteq \bullet) \mapsto (\mathbf{x}_2 \doteq \bullet)\}$$

$$[\![\mathbf{x}_1 := a(\mathbf{x}_2,b);]\!]^{\top}_{\mathcal{H}} = \{(\mathbf{x}_1 \doteq \bullet) \mapsto (a(\mathbf{x}_2,b) \doteq \bullet), (\mathbf{x}_2 \doteq \bullet) \mapsto (\mathbf{x}_2 \doteq \bullet)\}$$

$$[\![f]\!]^{\top} \circ [\![\mathbf{x}_1 := a(\mathbf{x}_2,b);]\!]^{\top}_{\mathcal{H}} = \begin{array}{ll} \{ & (\mathbf{x}_1 \doteq \bullet) \mapsto [\![f]\!]^{\top}(a(\mathbf{x}_2,b) \doteq \bullet), \\ & (\mathbf{x}_2 \doteq \bullet) \mapsto [\![f]\!]^{\top}(\mathbf{x}_2 \doteq \bullet)\} \end{array}$$
$$= \{(\mathbf{x}_1 \doteq \bullet) \mapsto (a(b,b) \doteq \bullet) \wedge (b \doteq \mathbf{x}_2), (\mathbf{x}_2 \doteq \bullet) \mapsto (\mathbf{x}_2 \doteq \bullet)\}$$

□

Being able to effectively represent weakest precondition transformers and to compute their composition, we can set up an abstract interpretation system [WP] of the constraint system $[\mathrm{S}^{\top}]$. This abstract system accumulates the weakest precondition transformers for mappings from $\mathbf{x}_i \doteq \bullet$ to conjunctions of Herbrand equalities. The smallest solution of this constraint system characterises for each procedure f the weakest precondition transformers for the validity of Herbrand equalities of the form $\mathbf{x}_i \doteq \bullet$. We achieve this by replacing the general transformers in the constraint system $[\mathrm{S}^{\top}]$ from section 3.1.2 with our more limited ones:

[WP1]	$\mathbf{WP}[\mathsf{r}_f] \supseteq \mathbf{Id}$	for each procedure f and each $\mathbf{x}_i$
[WP2]	$\mathbf{WP}[u] \supseteq [\![\mathsf{stm}]\!]^{\top}_{\mathcal{H}} \circ \mathbf{WP}[v]$	for each $(u, \mathsf{stm}, v) \in E$
[WP3]	$\mathbf{WP}[u] \supseteq [\![f]\!]^{\top} \circ \mathbf{WP}[v]$	for each $(u, f(), v) \in E$
[WP4]	$[\![f]\!]^{\top} \supseteq \mathbf{WP}[\mathsf{st}_f]$	f's effect is accumulated at procedure start

[WP1] marks that at all procedure return points r_f we start with $\mathbf{Id}$, which essentially maps every $\mathbf{x} \doteq \bullet$ to itself. In [WP2], a program instruction's weakest precondition transformer is composed to the so far accumulated weakest precondition transformer. Finally, [WP3] yields that the cumulated effect of a procedure is obtained via the weakest precondition at the procedure's start. We denote the greatest solution of system $\mathbf{WP}$ for a procedure f by $[\![f]\!]^\top$ in [WP4].

Now, having characterised the effects of each procedure, we can embed them into the system $\mathbf{C}$ to characterise the weakest precondition of the template $\mathbf{x}_i \doteq \bullet$, which enables us to infer a concrete constant value for $\mathbf{x}_i$ at t:

$[\mathrm{C1}^\top]$	$\mathbf{C}^\top[t] \supseteq (\mathbf{x}_i \doteq \bullet)$	for inferring a constant for $\mathbf{x}_i$ at t
$[\mathrm{C2}^\top]$	$\mathbf{C}^\top[u] \supseteq [\![\mathsf{stm}]\!]^\top_{\mathcal{H}}\, \mathbf{C}^\top[v]$	for each $(u, \mathsf{stm}, v) \in E$
$[\mathrm{C3}^\top]$	$\mathbf{C}^\top[u] \supseteq [\![f]\!]^\top \mathbf{C}^\top[v]$	for each $(u, f(), v) \in E$
$[\mathrm{C4}^\top]$	$\mathbf{C}^\top[u] \supseteq \mathbf{C}^\top[\mathsf{st}_f]$	for each $(u, f(), _) \in E$

Theorem 22 . $\mathcal{C}_{\mathcal{H}}[t] \models (\mathbf{x}_i \doteq \bullet)$ *iff* $\Sigma_\infty \models \mathbf{C}^\top[\mathsf{st}_{main}]$.

Proof. As $\mathcal{C}_{\mathcal{H}}[\mathsf{st}_{main}] = \Sigma_\infty$, $\mathbf{C}^\top[\mathsf{st}_{main}]$ is a tautology iff $\mathcal{C}_{\mathcal{H}}[\mathsf{st}_{main}] \models \mathbf{C}^\top[\mathsf{st}_{main}]$. By corollary 4 and the lemmas 23 and 24, we get that $[\![f]\!]\Sigma \models (\mathbf{x}_i \doteq \bullet) \Leftrightarrow \Sigma \models [\![f]\!]^\top(\mathbf{x}_i \doteq \bullet)$, especially:

$$\mathcal{C}_{\mathcal{H}}[\mathsf{st}_{main}] \models \alpha(\mathbf{R}_t(\mathsf{st}_{main})) \Leftrightarrow \mathcal{C}_{\mathcal{H}}[t] \models \alpha(\mathbf{R}_t(t))$$

We now prove via the fixpoint transfer lemma (c.f. [Cou97, AP86]), that the least fixpoint of $[\mathrm{C}^\top]$ is exact, i.e. $\mathbf{C}^\top[\mathsf{st}_{main}] = \alpha(\mathbf{R}_t(\mathsf{st}_{main}))$. The precondition to turn this lemma applicable requires that α is monotonic and distributive for arbitrary least upper bound operators. Luckily, this is the case, as shown above. The further requirement for the fixpoint transfer lemma is, that the transformer functions are exact – which they are due to corollary 4 and the lemmas 23 and 24.
As a conjecture, we get:

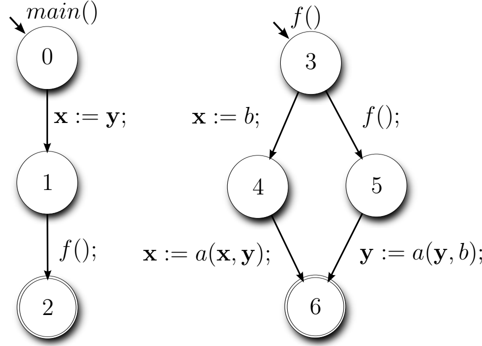
$$\mathcal{C}_{\mathcal{H}}[\mathsf{st}_{main}] \models \mathbf{C}^\top[\mathsf{st}_{main}] \Leftrightarrow \mathcal{C}_{\mathcal{H}}[t] \models \mathbf{C}^\top[t]$$

which is just another representation of the theorem, as $\mathcal{C}_{\mathcal{H}}[\mathsf{st}_{main}] = \Sigma_\infty$ and $\mathbf{C}^\top[t] = (\mathbf{x}_i \doteq \bullet)$. □

This provides us with the means to infer valid constant Herbrand equalities at t:

Corollary 5 . $C^\top[\mathsf{st}_{main}]$ *provides a* $\bullet = h$ *with* $h \in \mathcal{T}_\Omega$*, if* $\mathbf{x}_i \doteq h$ *is valid at t.*

Example 36 . Let $\mathbf{X} = \{\mathbf{x}, \mathbf{y}\}$, $\Omega_0 = \{b\}, \Omega_2 = \{a\}$ *and* $\|a\| = 1, \|b\| = 2$*. Consider the following Herbrand program:*



We are interested in the constant value of program variable $\mathbf{x}$ *at program point* 6*. We first compute the least solution for* $\mathbf{WP}[3]$*:*

$$\mathbf{WP}[6] = \{(\mathbf{x} \doteq \bullet) \mapsto (\mathbf{x} \doteq \bullet), (\mathbf{y} \doteq \bullet) \mapsto (\mathbf{y} \doteq \bullet)\}$$

$$\mathbf{WP}[5] = [\![\mathbf{x} := a(\mathbf{y}, b);]\!]_{\mathcal{H}}^{\top} \circ \mathbf{WP}[6] = \{(\mathbf{x} \doteq \bullet) \mapsto (a(\mathbf{y}, b) \doteq \bullet), (\mathbf{y} \doteq \bullet) \mapsto (\mathbf{y} \doteq \bullet)\}$$

$$\mathbf{WP}[4] = [\![\mathbf{x} := a(\mathbf{x}, \mathbf{y});]\!]_{\mathcal{H}}^{\top} \circ \mathbf{WP}[6] = \{(\mathbf{x} \doteq \bullet) \mapsto (a(\mathbf{x}, \mathbf{y}) \doteq \bullet), (\mathbf{y} \doteq \bullet) \mapsto (\mathbf{y} \doteq \bullet)\}$$

$$[\![f]\!]_{\mathcal{H}}^{\top} \circ \mathbf{WP}[5] = \{(\mathbf{x} \doteq \bullet) \mapsto (a(\mathbf{y}, b) \doteq \bullet), (\mathbf{y} \doteq \bullet) \mapsto (\mathbf{y} \doteq \bullet)\}$$

$$[\![\mathbf{x} := b;]\!]_{\mathcal{H}}^{\top} \circ \mathbf{WP}[4] = \{(\mathbf{x} \doteq \bullet) \mapsto (a(b, \mathbf{y}) \doteq \bullet), (\mathbf{y} \doteq \bullet) \mapsto (\mathbf{y} \doteq \bullet)\}$$

Applying the least upper bound on both constraints yields

$$\mathbf{WP}[3] = [\![f]\!]_{\mathcal{H}}^{\top} \circ \mathbf{WP}[5] \ \sqcup \ [\![\mathbf{x} := b;]\!]_{\mathcal{H}}^{\top} \circ \mathbf{WP}[4] =$$

$$= \{(\mathbf{x} \doteq \bullet) \mapsto (a(b, b) \doteq \bullet) \wedge (b \doteq \mathbf{y}), (\mathbf{y} \doteq \bullet) \mapsto (\mathbf{y} \doteq \bullet)\}$$

This stabilises after the second reevaluation. Now we can start to compute the precondition for $(\mathbf{x}_i \doteq)$*:*

$$\mathcal{C}^{\top}[6] = (\mathbf{x} \doteq \bullet)$$

$$\mathcal{C}^{\top}[5] = [\![\mathbf{x} := a(\mathbf{y}, b);]\!]_{\mathcal{H}}^{\top} \mathcal{C}^{\top}[6] = (a(\mathbf{y}, b) \doteq \bullet)$$

$$\mathcal{C}^{\top}[4] = [\![\mathbf{x} := a(\mathbf{x}, \mathbf{y});]\!]_{\mathcal{H}}^{\top} \mathcal{C}^{\top}[6] = (a(\mathbf{x}, \mathbf{y}) \doteq \bullet)$$

$$\mathcal{C}^{\top}[1] = \mathcal{C}^{\top}[3] \supseteq (a(b, \mathbf{y}) \doteq \bullet) \sqcup ((a(b, b) \doteq \bullet) \wedge (b \doteq \mathbf{y}))$$

$$\mathcal{C}^{\top}[0] = [\![\mathbf{y} := b;]\!]_{\mathcal{H}}^{\top} \mathcal{C}^{\top}[1] = (a(b, b) \doteq \bullet)$$

So, we inferred, that at program point 6*, the constant equality* $\mathbf{x} \doteq a(b, b)$ *is valid.* $\square$

3.4 Local variables

The semantics of Herbrand programs as specified in section 3.1.2 is based on the concept, that all program variables are accessible globally. However, we can extend this framework with a notion of local variables, similar to the extension, which we provided in section 2.3.8 to polynomial analysis. We leave the principles for procedure calls as they are, which means, that one may realise parameter passing, by fixing a subset of the global variables as parameter exachange variables. This way, copies of values may be passed to procedures.

We choose $\mathbf{Y} = \{\mathbf{y}_1, \ldots, \mathbf{y}_m\}$ to be the set of local variables of all procedures. $\mathbf{X}$ remains the set of all globally accessible program variables. This causes of course, that we have to extend the program states for the collecting semantics to the local variables from $\mathbf{Y}$. A program state thus is now a ground substitution $\sigma : (\mathbf{X} \cup \mathbf{Y}) \mapsto \mathcal{T}_\Omega$. Likewise, the transformers $\mathcal{S}_\mathcal{H}[u]$ are now sets of transformers of substitutions from $2^{((\mathbf{X}\cup\mathbf{Y})\mapsto\mathcal{T}_\Omega)\mapsto((\mathbf{X}\cup\mathbf{Y})\mapsto\mathcal{T}_\Omega)}$.

First, we define a helper function, which separates the name spaces of caller and callee by resetting the local variables of the caller's locals:

$$\mathit{enter}(\Sigma) = \{\sigma \oplus \{\mathbf{y}_i \mapsto h \mid 1 \leq i \leq m, h \in \mathcal{T}_\Omega\} \mid \sigma \in \Sigma\}$$

Second, we need a function, which determines how the effect of a procedure call influences the caller, given the set of effects of a procedure's body F:

$$H(F)(\Sigma) = \{\sigma \oplus \{\mathbf{x}_i \mapsto f(\sigma')(\mathbf{x}_i) \mid \sigma' \in \mathit{enter}(\{\sigma\}), f \in F\} \mid \sigma \in \Sigma\}$$

Note that parameter passing, though not explicitly specified for procedure calls can be implemented in this framework via assignments to and from global variables.

To obtain a concrete Herbrand collecting semantics with support for local variables, we have to modify the constraint systems $[\mathbf{R}]$ and $[\mathcal{C}_\mathcal{H}]$ with the following constraints:

$$\begin{array}{llll}
[\mathbf{R}3'] & \mathbf{R}_t[u] & \supseteq H(\mathbf{R}_{\mathsf{r}_f}[\mathsf{st}_f]) \circ \mathbf{R}_t[v] & \text{for each } (u, f(), v) \in E \\
[\mathcal{C}_\mathcal{H}3'] & \mathcal{C}_\mathcal{H}[v] & \supseteq H([\![\mathbf{R}_{\mathsf{r}_f}[\mathsf{st}_f]]\!]_\mathcal{H})(\mathcal{C}_\mathcal{H}[u]) & \text{for each } (u, f(), v) \in E \\
[\mathcal{C}_\mathcal{H}4'] & \mathcal{C}_\mathcal{H}[\mathsf{st}_f] & \supseteq \mathit{enter}(\mathcal{C}_\mathcal{H}[u]) & \text{for each } (u, f(), _) \in E
\end{array}$$

For the weakest precondition calculus on Herbrand equalities, we now also have to handle local variables from $\mathbf{Y}$. First, we have to ensure that all preconditions which have to be satisfied at procedure start have to be valid for arbitrary values of the callee's locals. To only pass such equalities, we introduce the following helper function:

$$\mathit{enter}^\top(E) = \forall_{i=1}^m.\forall_{\mathbf{y}_i}.E = \begin{cases} \top \text{ if } \mathbf{Vars}(E) \cap \mathbf{Y} \neq \emptyset \\ E \text{ otherwise} \end{cases}$$

for which we can show, that it is the weakest precondition transformer corresponding to its counterpart enter:

Lemma 25 . *For $\Sigma : \mathbf{X} \mapsto \mathcal{T}_\Omega$ and $E \in \mathbb{E}$:*

$$enter(\Sigma) \models E \quad \Leftrightarrow \quad \Sigma \models enter^\top(E)$$

Proof. We treat two cases here:
Case 1 – $\mathbf{Vars}(E) \cap \mathbf{Y} \neq \emptyset$: Here, we have to show that:

$$\{\sigma \oplus \{\mathbf{y}_i \mapsto h \mid 1 \leq i \leq m, h \in \mathcal{T}_\Omega\} \mid \sigma \in \Sigma\} \models E \quad \Leftrightarrow \quad \Sigma \models \top$$

This is true, as both expressions evaluate to false: The right one is defined to yield false. The left one's reduced equalities E contain some local variable $\mathbf{y}_i$. This is why E turns out to be unsatisfiable by the program states as they are defined to have arbitrary many different values for $vary_i$. The left hand side expression thus also evaluates to false.
Case 2 – $\mathbf{Vars}(E) \cap \mathbf{Y} = \emptyset$: Here, we have to show that:

$$\{\sigma \oplus \{\mathbf{y}_i \mapsto h \mid 1 \leq i \leq m, h \in \mathcal{T}_\Omega\} \mid \sigma \in \Sigma\} \models E \quad \Leftrightarrow \quad \Sigma \models E$$

This holds obviously, as the variables from $\mathbf{Y}$ are the only ones, which are manipulated on the left hand side, and E does not constrain any values of variables $\mathbf{Y}$ in this case. □

At procedure calls, we have to make sure, that the name spaces of caller and callee remain separated, while applying the precondition transformer's effect. Thus we introduce another set $\tilde{\mathbf{Y}}$ with the same cardinality as $\mathbf{Y}$ to temporarily rename local variables of the caller, while the effect of the callee is carried out. If $F^\top$ is the set of effects of the procedure body of f, $H^\top(F^\top)(E)$ is the effect of a call to this procedure to a conjunction of Herbrand equalities equalities E, where

$$H^\top(F^\top)(E) = (enter^\top(F^\top(E[\tilde{\mathbf{Y}}/\mathbf{Y}])\,))[\mathbf{Y}/\tilde{\mathbf{Y}}]$$

and $[\tilde{\mathbf{Y}}/\mathbf{Y}]$ is short for $[\tilde{\mathbf{y}}_1/\mathbf{y}_1, \ldots, \tilde{\mathbf{y}}_{|\mathbf{Y}|}/\mathbf{y}_{|\mathbf{Y}|}]$. As the concrete semantics require that local variables may hold arbitrary values, all preconditions induced by the procedure call must hold for all values of the callee's local variables in the precondition, which leads to the universal quantification: $\forall_{i=1}^m \forall_{\mathbf{y}_i}$. This quantification is implemented exactly as non-determinstic assignments.

The constraint systems $[\mathrm{C}^\top]$ and $[\mathrm{S}^\top]$ must be modified to reflect these changes, which leads to the following modified constraints:

$[\mathrm{S2}'^\top]$	$S^\top[u] \supseteq H^\top([\![s]\!]_\mathcal{H}^\top) \circ S^\top[v]$	for each $(u, s, v) \in E$
$[\mathrm{C3}'^\top]$	$\mathbf{C}^\top[u] \supseteq H^\top(S^\top[\mathsf{st}_f])(\mathbf{C}^\top[v])$	for each $(u, f(), v) \in E$
$[\mathrm{C4}'^\top]$	$\mathbf{C}^\top[u] \supseteq enter^\top(\mathbf{C}^\top[\mathsf{st}_f])$	for each $(u, f(), _) \in E$

What we still have to show is:

Theorem 23 . *Let $F(\Sigma) \models E \Leftrightarrow \Sigma \models F^\top(E)$ with $\Sigma \subseteq \mathbf{X} \mapsto \mathcal{T}_\Omega(\mathbf{X})$ and $E \in \mathbb{E}$. Then*

$$\Sigma \models H^\top(F^\top)(E) \quad \Leftrightarrow \quad H(F)(\Sigma) \models E$$

Proof.

$$\Sigma \models H^\top(F^\top)(E) \quad \Leftrightarrow$$

$$\Sigma \models (enter^\top(F^\top(E[\tilde{\mathbf{Y}}/\mathbf{Y}])))[\mathbf{Y}/\tilde{\mathbf{Y}}] \quad \Leftrightarrow$$

$$\Sigma[\tilde{\mathbf{Y}}/\mathbf{Y}] \models (enter^\top(F^\top(E[\tilde{\mathbf{Y}}/\mathbf{Y}]))) \quad \Leftrightarrow$$

$enter^\top()$ distributes over $\wedge$:

$$\Sigma[\tilde{\mathbf{Y}}/\mathbf{Y}] \models enter^\top(F^\top(E[\tilde{\mathbf{Y}}/\mathbf{Y}])) \quad \Leftrightarrow$$

By lemma 25, we get:

$$enter(\Sigma[\tilde{\mathbf{Y}}/\mathbf{Y}]) \models F^\top(E[\tilde{\mathbf{Y}}/\mathbf{Y}]) \quad \Leftrightarrow$$

By applying the precondition, we obtain:

$$F(enter(\Sigma[\tilde{\mathbf{Y}}/\mathbf{Y}])) \models E[\tilde{\mathbf{Y}}/\mathbf{Y}] \quad \Leftrightarrow$$

$$F(enter(\Sigma[\tilde{\mathbf{Y}}/\mathbf{Y}]))[\mathbf{Y}/\tilde{\mathbf{Y}}] \models E \quad \Leftrightarrow$$

$$\{f(\sigma') \mid \sigma' \in enter(\Sigma[\tilde{\mathbf{Y}}/\mathbf{Y}]), f \in F\}[\mathbf{Y}/\tilde{\mathbf{Y}}] \models E \quad \Leftrightarrow$$

As f is not defined on variables from $\tilde{\mathbf{Y}}$, this leaves $\sigma(\mathbf{Y})$ unchanged:

$$\{\sigma \oplus \{\mathbf{x}_i \mapsto f(\sigma')(\mathbf{x}_i) \mid \sigma' \in enter(\{\sigma\}), f \in F\} \mid \sigma \in \Sigma\} \models E \quad \Leftrightarrow$$

$$H(F)(\Sigma) \models E$$

□

With this theorem, we get that $H^\top$ is the exact weakest precondition transformer w.r.t. the concrete semantics. Therefore, we merge the old constraints with the modified versions to new systems [$C'^\top$] and [$S'^\top$], and obtain as theorem, analogously to theorem 15:

Theorem 24 . *Let the Herbrand equality $h_1 \doteq h_2$ be the conjecture to verify at a program point t of a program with local variables, i.e. $\mathbf{C}'^\top[t] \supseteq (h_1 \doteq h_2)$. Then*

$$\mathcal{C}'_{\mathcal{H}}[t] \models (h_1 \doteq h_2) \quad \textit{iff} \quad \Sigma_\infty \models \mathbf{C}'^\top[\mathsf{st}_{main}]$$

Proof. [Sketch] Analogously as the proof to theorem 15, but this time also considering theorem 23 for exactness of procedure call with local variables.. □

3.5 Conclusion

Summarising, in this section, we presented a general framework for computing the precondition for the validity of Herbrand equalities in the context of Herbrand programs with procedure calls. Based on this, we concentrated on special instances of this framework, for which all equalities can be inferred precisely. We presented, how to encode Herbrand terms into polynomials, and Herbrand programs into polynomial programs. This enabled us to apply generic polynomial ideal transformers for representing procedure effects. With this kind of encoding, we are able to infer all Herbrand equalities in programs, whose assignments have only one variable on their right-hand side.

Our second proposal was to consider the programs only for their behaviour concerning the validity of constant equalities. We observed, that it would only be necessary to tabulate the effect of a transformers for a very bounded set of equalities and showed, that the composition function which we provided for this representation conserves precision for constant equalities. As a consequence, this analysis allowed to infer all valid Herbrand constants.

Finally, we demonstrated how to extend our framework to differentiate local and global variables. The means to achieve this are proven to be exact with respect to their counterparts on the concrete semantics.

Summarising, the approaches in this part of the thesis yield notable results in decidability problems, complementing and confirming the results from Müller-Olm, Steffen and Seidl [MOSS05] concerning equality with Herbrand constants. Also, it achieves results in precise analysis of Herbrand equalities in programs with procedure calls based on a different understanding of procedure call effects than context unification as done e.g in [GT07b] which is another major research field that is currently explored for subclasses which allow precise procedure call handling.

Chapter 4

Implementing a framework for program analyses

In this chapter we face the needs and success criteria for a framework for program analysis in science and education. First, we have a look on typical situations in need for a better tool support, then present an analysis of demands on the framework, elaborating on a list of key success criteria for frameworks in this environment. After a rating of common existing tools, considering our criteria, we advance to compile a set of ideas to counter the deficiencies existing tools have in our environment. We then describe how these ideas have been implemented and finally evaluate to which degree the elaborated criteria can be fulfilled.

4.1 Usage scenarios

The urge for a new framework to handle program analyses and optimisations arises from several scenarios for which there was no appropriate solution with existing tools. The situations in which there was demand were related to teaching program analysis techniques as well as rapidly developing prototypes for new analysis approaches in science.

Scenario 1: Frontend development

The first usage scenario is settled in teaching compiler construction in a practical course. The goal is to connect program code, written in an input language, to the given compiler framework. The relevant connection to the compiler framework is its internal representation of program code, i.e. its intermediate representation of a programming language. The students develop a parser that first creates an abstract syntax tree from their language and then design means to extract the control flow from their syntax tree and shape it with the help of provided tools in such a way that it complies to the intermediate language of the framework.

Scenario 2: Teaching program analysis and transformation

The second usage scenario is settled in teaching design of program analyses and transformations in an exercise course. For the instructor the goal here is to communicate the setup of typical flow analyses and provide an easy way to implement standard analyses as exercise. Thus, the work for a student here concentrates on the development of an analysis based on a given simple intermediate language. The interaction with the program analysis framework comprises the interface to obtaining the control flow graph from a given program and means to inspect, annotate and manipulate it. These actions are responded by the framework in form of a visualisation of the information gathering and graph transformation process.

Scenario 3: Rapid prototyping of analysis chains

This scenario is focused on a usage of the framework with a scientific background. The framework has to serve as a platform to support the rapid, yet comfortable development and empirical evaluation of newly designed program analyses. In this case, the researcher designs either extensions to parts of the provided standard tools or designs new analyses achievable with available techniques and possibly depending on previous analyses. Quite often students are employed to carry out the implementation of the novel approach. In order to achieve practical results and to evaluate new approaches the researcher carries out analyses of practical examples or wants to evaluate the practical feasibility of his ideas and thus wants to run the implementation of his analysis on large examples.

Scenario 4: Comfortable evaluation of analysis results

The final use case is settled in the review of the analysis results. One focus is put on the way how to locate, filter and display the computed data flow values in a clearly arranged way. In a larger input file it is crucial not to be overwhelmed by the mass of gathered information which threatens to drown the specific information the instructor is after. The user thus interacts with a graphical visualisation of the control flow graph of the input program with comfortable means to navigate in this structure to locally and logically connected program points. The focus is the understanding of the settlement of a specific data flow value in the course of the analysis. This time, the user should be able to inspect the process of the analysis run.

4.2 Demands on tool support

Reviewing the usage scenarios, we identify several interfaces which have to be present for a homogeneous toolkit. From scenario 1, we deduce that there is the need for an intelligible intermediate language model which is language-independent, yet expressive enough that several real-world languages could be connected to the framework. Yet, as the time for a student's project is very limited quite often, it is important to have a concise intermediate language. So, we come to our first criterion:

Criterion 1 . *The framework must provide an expressive, concise intermediate language to represent arbitrary programming languages.*

Especially in use case 2, the author of a program analysis appreciates if the formalisms that are used in a lecture have direct counterparts when implementing a program analysis. Students solving exercises or employed for implementation work as well benefit from the fact that they recognise the structure of the theoretical concepts within the practical framework. In this way, the program analysis author is supported in concentrating on functional matters only. This minimises the extra effort, that the analysis developer has, when he moves from specifying the theoretical concept to providing experimental results:

Criterion 2 . *The semantics of the analysis ought to be declared in a compact way, i.e. it should be as near as possible to the common formal approach to specify flow analyses.*

As in science (cf. use case 3), the analysis developer often relies on the fact that program analyses can be realised as fixpoint iterations on lattices, an ideal experimental environment should enable to use arbitrary lattice based program analyses. The benefit of such a framework rises, if the programmer is able to use a consistent interface for all routine jobs in such analyses and transformations. This is also true for educational purposes, as time is too limited in classes to learn more than one framework. Gathered from scenarios 2 and 3, we thus obtain:

Criterion 3 . *The framework must provide a generic analysis interface which allows an easy access to the flow graph in order to obtain, traverse and modify it.*

The deployment of the framework in an exercise course as use case 2 dictates is that the analysis interface has to be easily understandable and extensible, while keeping the focus more on the aspects of the specific program analysis or optimisation than on e.g. the generation of flow graphs or fixpoint iteration algorithms. Thus, it is important that the framework provides for each of these tasks a decent default module that can be used generally. Contrarily, use case 3 suggests that the analysis interface ought to be flexible enough to allow to exchange the default modules by default implemented modules with user-provided versions if needed. In summary, this leads to:

Criterion 4 . *The analysis framework must both provide an adequate set of default modules and allow for vital modules to be dynamically replaced by user defined ones.*

Especially use case 3 as well as use case 2 explicitly require that students implement analyses with this framework. In order to avoid long tool training, at least the programming language for interfacing the flow graph should be one most students know well. Furthermore, the simplicity and easiness of development of program analyses is facilitated by the choice of a programming environment that offers advanced tools for debugging and tracing of the produced code. Another crucial paradigm is that for prototypes and educational exercises the focus shifts from creation of top-speed programs to rapid development and concentration on matter more than on technique. Hence, elaborated manual memory management issues tend to distract more from the main matter than they acquire. The application of common patterns in analysis templates is encouraged in order to exploit on experiences with similar programming problems.

Thus, we come up with:

Criterion 5 . *Program analyses must be implemented in a common standard high-level programming language that offers convenient ways for debugging and tracing. The usage of common patterns in interface declaration is considered worthwhile.*

Furthermore, use case 3 also accounts for the need to interconnect an analysis, currently in development, with preceding supporting analyses. Hence, on the one hand, it must be easy to combine several analysis phases into a sequel. On the other hand, the results produced by former runs of analyses in the same framework or even provided from external sources must be easily accessible from consecutive analysis phases:

Criterion 6 . *The framework must support analyses composed of several phases each of which must be able to access results from previous phases easily.*

Use case 3 as well as use case 4 impose the possibility to import realistic examples as test objects into the analysis framework, either as benchmark or as feasibility proof. As these inputs have to be taken from existing programs, it is important for an analysis framework to have a frontend interface to some wide-spread standard representation of programs:

Criterion 7 . *There must be a frontend for realistic input of test objects, concerning size and standard conformity.*

Especially when dealing with real-world examples, analysis output may be extremely large. Throughout use cases 2 and 3, it is crucial for the developers to be able to inspect the results that their analyses obtain at certain program points as a feedback about the correctness of their implementation. The vast amount of computed control flow values imposes to find some visually appealing and quickly intuitively inspectable representation forms for analysis results. Hence, we tackle this as:

Criterion 8 . *There must be a way to visualise and navigate through the results of analyses on possibly large control flow graphs intuitively, quickly and accurately.*

Considering the fact, that flow analyses based on fixpoint iterations compute their values in several iterations with intermediate results, an ordinary output of the final result of the computation for each program point may not satisfy the developer in use case 2 to 4 to comprehend the accomplishment of the final value. Thus, the ability to monitor the evolution of the flow values represents an integral advantage for the developer of a program analysis – debugging of implementation errors is only reasonable, when instead of only looking at the final result there is a possibility to observe the changes that are induced by state transitions in every analysis pass, the localisation of an implementation error is much easier. Even more, it offers insights for the educated user of a program analysis as in use case 4. This is recorded in:

Criterion 9 . *There must be a way to observe the changes of flow values during fixpoint computation.*

Uniquely for use case 2, the framework is intended to be used in the context of teaching automatic program transformation techniques. Demonstrating these transformations by visual effects on the representation of a control flow graph is especially helpful in

teaching how program optimisations actually work. Thus, we state:

Criterion 10 . *The transformations on the control flow graph must be reflected in its visual representation.*

Summarising, we have gathered the following criteria from the use cases:

(i) The framework must provide an expressive, concise intermediate language to represent arbitrary programming languages.

(ii) The semantics of the analysis ought to be declared in a compact way, i.e. it should be as near to the common formal approach to specify flow analyses as possible.

(iii) The framework must provide a generic analysis interface which allows an easy access to the flow graph in order to obtain, traverse and modify it.

(iv) The analysis framework must both provide an adequate set of default modules and allow for vital modules to be dynamically replaced by user-defined ones.

(v) Program analyses must be implemented in a common high-level programming language that offers convenient ways for debugging and tracing. The usage of common patterns in interface declaration is considered worthwhile.

(vi) The framework must support analyses composed of several phases, each of which must be able to access results from previous phases easily.

(vii) There must be a frontend for realistic input of test objects concerning size and standard conformity.

(viii) There must be a way to visualise and navigate through the results of analyses on possibly large control flow graphs intuitively, quickly and accurately.

(ix) There must be a way to observe the changes of flow values during fixpoint computation.

(x) The transformations on the control flow graph must be reflected in its visual representation.

In the following, these criteria will serve in rating the tools on-hand, inspiring design propositions and evaluating the success of the developed toolkit.

Comparison of other toolkits

Of course, in the area of program analysis, there are already existing toolkits which offer frameworks to develop program analyses. Each of them is a little different in its features and limitations. The tools which came closest to the needs of a program analysis developer are compared here:

- PAG is the engine behind the Absint binary code analysers. It offers a small functional language named FULA to specify domains and transfer functions for analyses. The results, that are produced by PAG can be visualised with the tool Aisee. The drawbacks of the PAG framework transpire, when the simple data structures within FULA do not suffice to implement an efficient analysis. The fallback strategy is to specify the analysis in C, which is error prone due to the undocumented C interface and hard to debug. Further, PAG analyses are in practise limited to call-string based interprocedural analyses, while its constraint solver

is limited to a few notions of fixpoint iterators. Furthermore, it is not so easy to implement succeeding and communicating analyses with this framework.

- LLVM is a widely used compiler framework with a very clean and well-specified intermediate representation of low-level code. It is maintained by a large community and already offers a large number of optimisations which are already implemented in this framework. As it originates from compiler suites, it has a lot of different practical machine architectures, which are supported by LLVM. The drawbacks for using LLVM in our case is that although it has an interesting representation of low-level code, it does not directly come with a generic constraint-solver architecture. Interprocedural analysis and control flow extraction still has to be done in this framework, although on a concise intermediate language only. With neither a direct representation of the control flow of a program nor a constraint solver, there is naturally no animation of the analysis progress. Further, intermediate code is kept in SSA-form, leading to a vast usage of different variables – which does not serve well for algorithms, which are based on polynomial ideals, as their performance depends on the number of occurring variables.
- CIL is a frontend, providing access to the abstract annotated syntax tree of C programs via Ocaml. Same as for LLVM, the drawback of this solution is that it is not equipped with a direct concept of control flow extraction, different concepts for interprocedural analysis, or a visualisation of the analysis progress.

4.3 Developing VoTUM

Now, we focus on the development of the new program analysis framework called VoTUM (*Visualisation of optimisations at the TUM*). I first present a brainstorming for ideas how to realise an implementation of a program analysis framework, complying to the criteria from the previous section. Then, I show how these ideas were implemented in the concrete program framework.

4.3.1 Ideas for implementation

In this section I show the ideas that stick behind VoTUM to master the challenges, that are imposed on tools for program analysis in the context of the criteria from section 4.2. On the basis of these ideas and experiences we made during the development of the toolkit we came to the architecture for the framework, that we will present in the next section.

The first, and very basic design idea is the choice of the Java platform not only as programming language but also as interface to the framework. I deliberately turned down the development of an own macro language for the specification of program analyses. This is due to the experiences which we had when using the PAG framework's [AM95] internal program analysis specification languages DATLA/FULA. There are several problems with an own macro language: First, the framework has to maintain an own

parsing/compiling infrastructure, which can be a quite overwhelming task, considering that languages evolve and new concepts are continuously added to common established programming languages. Second, the program analysis designer is repelled by the perspective to learn yet another language. Furthermore, as a reasonably implemented macro language for sure has to live with limits, it interferes with the import of legacy analyses or libraries, or even complicates the development of new ones due to its limits. We thus implemented the new framework in a fashion that permits to use Java code for analysis implementation. This choice permits an easy way to import existing library code into program analyses. At the same time, the program analysis designer benefits from the availability of syntax highlighting, IDEs and debugging facilities of the Java platform. Also, Java is an actively maintained language, which is widely taught at universities. It preserves the programmer from the concerns of manual memory management, which concentrates all efforts on the specification of the analysis. As Java's class loader permits easy class reloading at runtime, the development of modular program analyses and a separation of program analyses and analysis framework becomes possible. With a high-level object-oriented language as Java, the programming interface of the program analysis framework may also adhere to common wide-spread programming patterns for object-oriented languages, as e.g. in Gamma et. al. [GHJ+93].

The functional core of VoTUM is the framework to organise the collaboration of arbitrary program analyses, fixpoint iteration strategies and machine models. For the execution of a program analysis, there are three major modules that need to be provided: First, there is the class of analysed programs, ranging from abstract meta languages like IML [App02] to concrete machine architectures, as e.g. PPC[SB04]. The second need is to switch between different strategies for fixpoint algorithms, ranging from semi-naive propagation [FS98] to Min-Max-strategies [GS07]. Since the abstract semantics for each program analysis varies, the most common component that will be replaced in a concrete application will be the definition of the semantics of the transition functions as well as the underlying lattice.

The general vision of VoTUM is to provide an object-oriented approach to perform program analyses with a default set of tools, allowing to replace these three modules with custom implementations at will. VoTUM furthermore offers a reasonable set of default implementations for each of the *infrastructure* modules like meta language, machine model or fixpoint strategy.

Concerning the design of the VoTUM core, there are common properties which can be shared between all the concrete implementations of modules. There is the visualisation and animation of control flow graphs, the global address space for data flow values and the fixpoint engine itself. Around these core components of the VoTUM framework, a common object-oriented interface is necessary for all externally contributed modules by using patterns i.e. from Gamma et. al. [GHJ+93] for interface declaration. Interfacing this core, there are different machine architectures for which control flow analyses are developed. The provision of a semantically arbitrary program architecture for analyses is possible via a combination of the builder pattern with a specific visitor pattern. With this help, VoTUM offers to dispatch the calls to the external analysis module, whenever the computation of a transition-effect is necessary. The transitions thus are given as an

implementation of a specific visitor, delegating the callbacks for different edges and nodes to concrete operations on the lattice, while the lattice itself is given as a child class of an abstract lattice template. Together with the definition of the complete lattice, which forms the basic data type for the flow values, these customised transition functions allow for implementation of arbitrary program analyses on custom machine architectures.

While processing a fixpoint iteration, the VoTUM core is offered the choice between multiple flow-equations which could be re-computed to obtain new flow values or incremental updates. The performance of certain program analyses depends on hinting the VoTUM core which program state to evaluate next. This advice is provided to the VoTUM core by modules of fixpoint strategies, connected to the core via the strategy pattern. Another aspect of interest is the way, VoTUM organises the store for the data flow values that are belonging to each program state. VoTUM thus only declares the interface that it uses for storing and retrieving data flow values, allowing for the storage related code to be implemented in a module, connected via the delegation pattern [GHJ+93].

As control flow is forming a graph, it can be displayed as a set of nodes, connected with edges, both labelled with relevant data, with the help of Java's graphic library Swing [Gea99]. The layout of the graph's components is a particular problem which is disposed to the well-established and more sophisticated graph drawing tool Graphviz [EGK+01]. Anyway, the option to implement a pure Java- based layouting solution is still left open, as a possible future extension to add a hierarchical graph layouting algorithm as described e.g. in [GKNV93]. The VoTUM framework transparently connects the imported control flow graphs to their graphical representation without extra effort for the program analysis creator. As the fixpoint-engine itself is also part of the VoTUM core, the course of the fixpoint algorithm can be monitored by the graph representation module, which is thus able to transparently indicate the hot spots of the transfer function evaluation in the graph. This is done by changing e.g. the colour of the nodes or edges in the hot spot of the iteration in the graph. For transformations of the control flow graph, VoTUM offers a centralised interface to perform changes in the central graph model and communicates them to the graph display module, which animates the changes as e.g. movements of the involved nodes or edges. Concerning the overview in large control flow graphs, there are several measurements to facilitate navigation. From the *aisee* control flow graph display tool [Abs], we know the concept of folding procedure bodies or basic blocks. There is also the idea of a bird's eye view, originally from Fukata et. al. [FKMK98]. A little more conventional is the idea of a substring search in the edge or node label's text.

Atop of the VoTUM core and its machine architecture interface is the task of connecting different language and machine architectures via front-ends. As this topic is quite often picked up in practical courses with students, there emerged quite diverse demands on the front-end interface of VoTUM, depending on the purpose of the actual implementation. This lead to the need for a tiered model of intermediate representations from high-level to low-level. This hierarchical model is also reflected in Appel [App02]. His structure of intermediate languages provides a reasonable template for designing the intermediate languages for VoTUM. Following this approach, it turns out to be

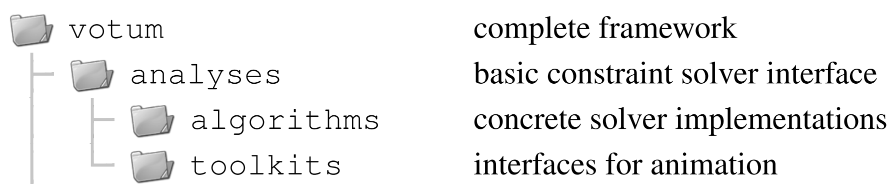
beneficial to have an intermediate language which is suitable as target for high-level source languages as well as another one which suites for performing flow analyses together with a transformation facility between them. Those components are provided by a default implementation in VoTUM together with a transformation from high-level to low-level code. As in education, the VoTUM framework is also used in the context of compiler construction labs it is beneficial to have a framework with preliminarily provided multiple tiers to which students can connect in their labs. Usually, such lessons concentrate on either development of a frontend for an arbitrary language, thus need an abstract high-level interface or on the other hand on the development of a back-end for an arbitrary target architecture, depending on an abstract low-level interface.

Regarding the more scientific application of VoTUM, the focus shifts from developing new frontends to developing new program analyses. Thus, for scientific applications it is more important to have at least one reliable, realistic and mature frontend. Nevertheless, in order to keep intermediate code's complexity low and thus development speed high, it is important to concentrate on a frontend which produces a compact and clear input model. Considering the fact that pure high-level languages have too many complex features to follow, the idea for VoTUM is to stick to assembly files instead. As there are still different choices for languages at this point, we were looking for an assembly model that is although sufficiently concise, yet common in the real-world. Our choice is thus one of the most common RISC-architectures, the PowerPC [IBM93], increasingly popular in the development of embedded systems [BTR02]. The basic instruction set is resumable in about 20 kinds of 3-address-code instructions, and thus lends itself quite well as programming model for program analyses. As the PowerPC platform is a major target architecture for the GNU compiler collection [Sta10], a compiler-infrastructure for generation and disassembly of PowerPC programs is freely accessible. Moreover, as PowerPC is an architecture, which is used for embedded programs, there are many programs which are conceptually build to gather all their executed code completely in a single file – which is beneficial for analyses that require the complete code of the analysed binary. It is also beneficial for student groups to have pre-implemented standard analyses like common sub-expression elimination or dead store elimination ready to speed up their own implementation.

4.3.2 Technical Realisation

Package overview

VoTUMs classes are distributed over several packages, providing solutions to specific problems. The main packages are `analyses`, `compilers`, `graph`, `gui` and `utils`, as the following figure illustrates:



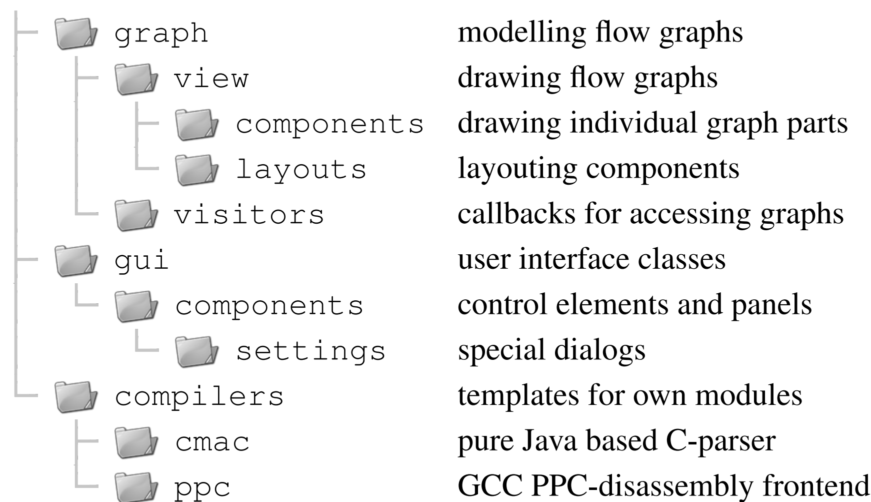


Figure 4.1: Sketch of global package structure in VoTUM

The fundamental classes for managing control flow graphs of any kind are provided by the `graph` package. Typical concepts for graphs like nodes, edges and basic visiting facilities are shared in all conceivable control flow graphs and thus are represented by classes in this package. Encapsulated in a sub-package view, there reside the classes for drawing graphs directly into Swing components and layouting graphs on a plane.

The classes from the `gui` package provide interaction between the user and the analysis framework. They contain the main class for running an analysis in VoTUM. Furthermore, all other user interface objects as e.g. settings dialogs, window components as the source code editor, the tabbed graph view and the analysis control panel are represented in own classes in this package. The classes from this package also provide an implementation of animations of graph changes, which come in handy for program transformations.

The package `analyses` gathers classes that contain common basic aspects of control flow analyses based on constraint solving. This package provides an interface for defining complete lattices, which is compatible to the provided abstract analysis phase. Besides establishing the connection to animations and display toolkits, this class also serves as basis for the concrete constraint solvers in the `algorithms` package.

The package `compilers` serves as initial point for extending the analysis framework with user content as own analyses or an own semantics for the edges in the control flow graph. In the sub-packages `cmac` and `ppc` two already implemented concrete architectures can be found. The `cmac` package provides an own pure Java implementation of a C-frontend generating a meta-language like in [App02]. The `ppc` package implements a complete representation of the PowerPC instruction set [SB04], generated by the GCC disassembler.

Finally the package `utils` gathers useful internal additions that are used throughout the VoTUM project. Since they do not have a global impact their description is omitted here.

Packages in detail

In this section we give an outline on the most important classes in the packages mentioned above, sketch their use and show their relations with each other.

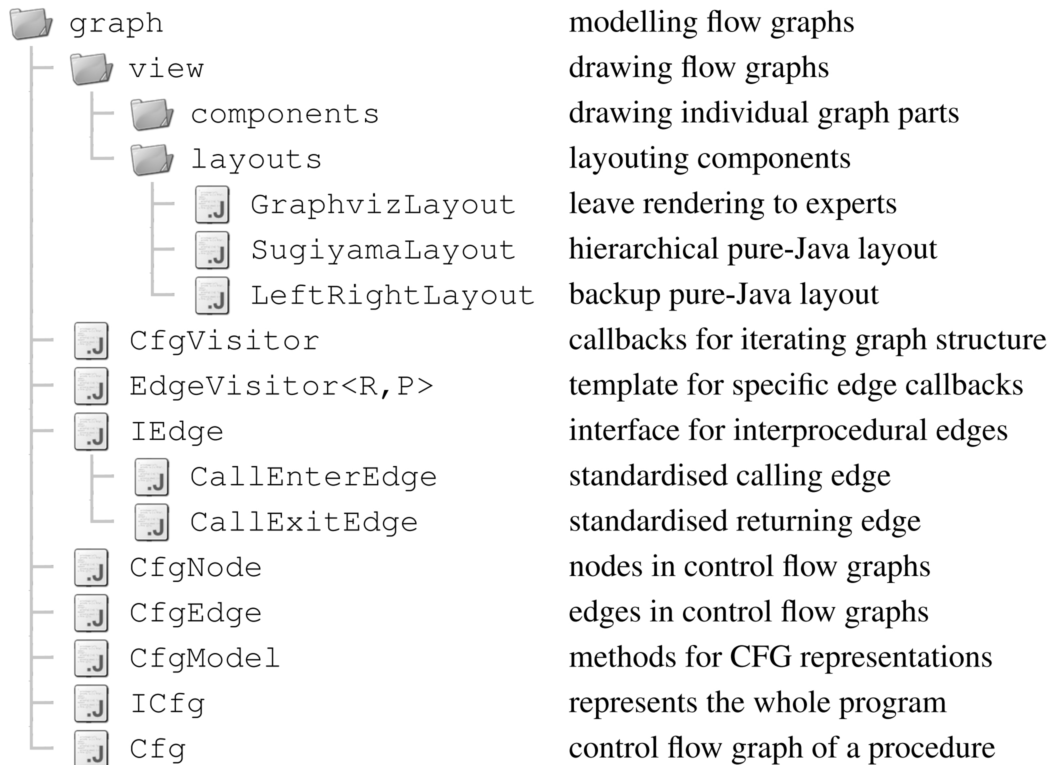


Figure 4.2: outline of package `graph`

In the `graph` package (see figure 4.2), the most central classes are those, assembling the control flow graphs of a program. In VoTUM, a program is represented by its interprocedural control flow graph, an instance of the class `ICfg`. It consists of a set of control flow graphs for each of its procedures, which are represented as instances of `Cfg`. Each of these graphs is a mesh-work of `CfgEdge` and `CfgNode` objects and provides the means to iterate through and change the structure of a control flow graph. Access to the structure of the control flow graph is not directly obtained through the nodes and edges themselves, but has to be queried from the parent `Cfg` or `ICfg`, which by implementing `CfgModel` offer the appropriate methods for this. This enables VoTUM to centrally manage the structure of the control flow graphs with the graph component classes acting as proxy objects to the underlying Swing components. This leads to a separation of the display logic and the user data, introducing a kind of model-view separation. In order to support analyses of programs with multiple procedures, there are special classes offered for edges that are intended to point from a procedure call site to the callee's `Cfg` and back: `IEdge.CallEnterEdge` and `IEdge.CallExitEdge`.

The `ICfg` provided by the backend is supposed to blend in these types of edges to signal jump destinations. Furthermore, the package offers with `CfgVisitor` a *visitor pattern* interface of callback routines for the classes occurring in a `Cfg`'s structure. Concerning the view part of the *model-view-controller* pattern, an `ICfg` manages to draw the control flow graph with the help of the Swing components from the `view` sub-package. The layouting of the components is sourced out to certain layouting modules which provide coordinates for the nodes via the *strategy pattern*. The classes in this package provide the general part of control flow graphs which does not depend on a special concrete computer architecture or intermediate language. In fact, the provision of `CfgEdge` subclasses with custom transfer functions is the way, how arbitrary semantics should be integrated into the framework. The only obligation for these custom edge subclasses is to accept visitation by an appropriate subclass of `EdgeVisitor` that provides the appropriate callbacks with a parameter of type `P` and return type `R` for this kind of architecture.

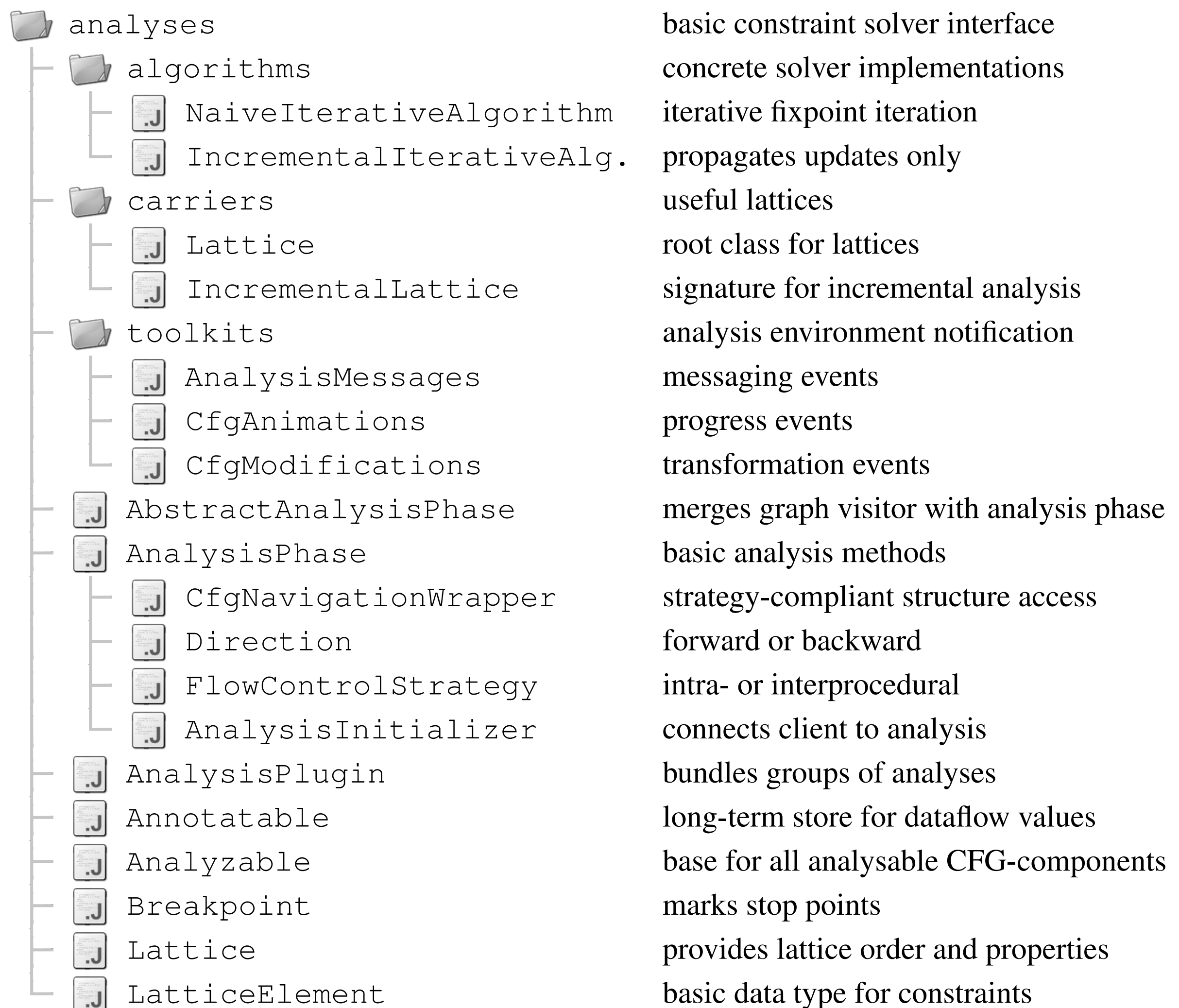


Figure 4.3: outline of package `analyses`

Based on control flow graphs, represented with classes from the `graph` package as in figure 4.3, the package `analyses` allows for analyses via abstract interpretation on control flow graphs. Analyses of control flow graphs via fixpoint iterations are based on constraints between the nodes of the graph. The structure of the control flow graph determines which nodes are related via constraints. The exact instance of constraint which relates to nodes is determined by the instruction which is annotated at the edges of the control flow graph. Eventually, the results of the evaluations of several constraints have to be joined, in order to provide an update for the data-flow value at a program point. If this re-evaluation leads to a new data-flow value, this change leads to other re-evaluations in the graph. At this point, the constraints which all edges, connecting the latest node with other nodes, have to be re-evaluated. This proceeding is reflected in the classes and interfaces, defined in the `analyses` package.

The class `AbstractAnalysisPhase` concentrates on the basic functionality to represent the structure of the program to analyse, which is reflected in the interface `CfgNavigationWrapper`: It answers these queries for the graph structure after consulting the internal control flow graph representations of the common interface type `CfgModel`, but other than naively reproducing the structure, it respects the actual analysis' *iteration strategy* and *direction*. Analyses in VoTUM can choose between whether analyses exhibit *forward* or *backward* semantics. The iteration strategy decides how to handle procedure calls: At the moment, one can choose the strategies `Intraproc`, `InterprocCallString` and `InterprocFunctional` for handling procedure calls. This affects how an `AbstractAnalysisPhase`-child views the connection of the nodes at the call site and the called procedure – e.g. the edges `CallEnter` and `CallExitEdges` can be blended in, depending on whether they confirm to the currently selected `FlowControlStrategy`, as can be seen in figure 4.4. To access the "real" unaltered program structure instead of the analysis' view, it is possible to directly query the `CfgModel`-instance instead, which can be obtained via `getICfg()`.

Anyway, rather than directly traversing the control flow graph via the `getSourceEdge/Node()` or `getTargetEdge/Node()` methods, the job of managing the sequence of items to be analysed can be left to the VoTUM framework by using `NaiveIterativeAlgorithm`. `NaiveIterativeAlgorithm` implements the interface from `AbstractAnalysisPhase`, providing a configurable fixpoint iteration on control flow graphs. All that is needed for an analysis are implementations of the callback methods `evaluateEdge()` and `evaluateNode()`, which are called everytime the fixpoint iteration decides to re-evaluate the constraints for a node or edge. In order to realise architecture-specific callbacks, this requires to implement a concrete subclass of the `EdgeVisitor` interface. This allows to distinguish different instructions of the analysed program, without restricting the fixpoint engine to a single program architecture.

The aspect of different analysis domains is approached in VoTUM via the `Lattice` interface, which has to be used as base for dataflow values. This interface establishes the

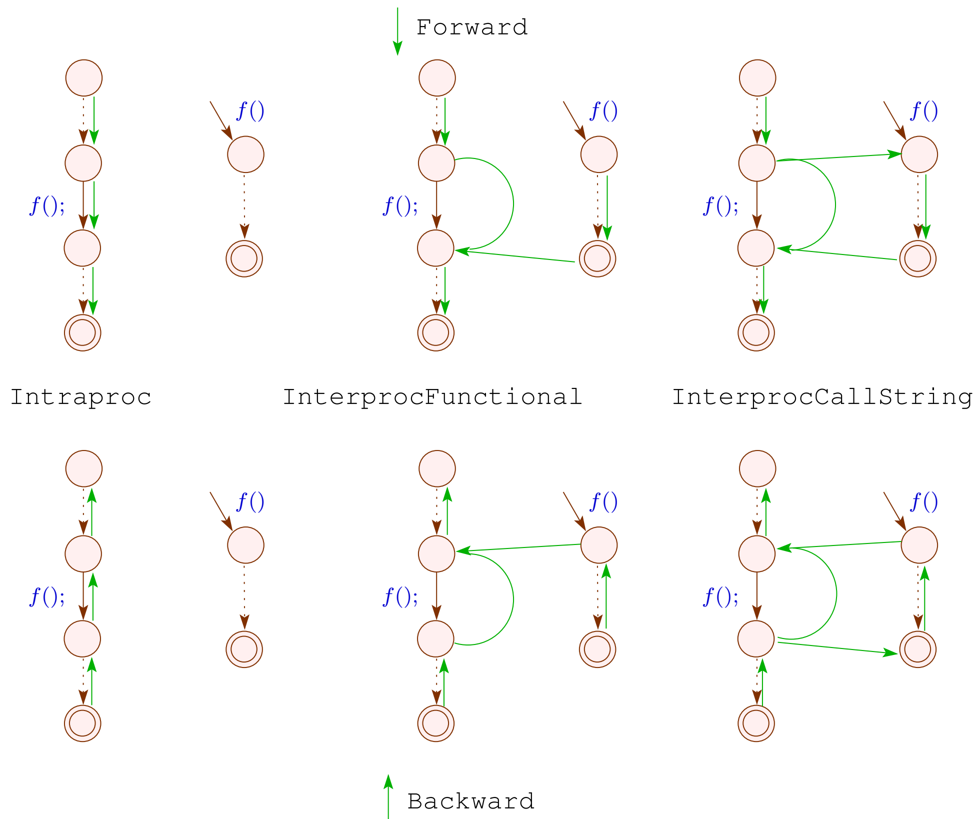


Figure 4.4: Control flow at procedure calls, complying to different `FlowControlStrategys` and `Directions` marked in green

common methods of interest `lessOrEqual()`, `merge()` and `widen()`. Special domains to analyse programs in VoTUM can be realised by an own implementation of this interface – semantically, these domains comply to complete lattices, featuring a partial order with `lessOrEqual()` and a least upper bound `merge()`. For analyses that are based on *widening*-techniques, there is a method `widen()`, which can be overwritten as needs arise, but as default may simply delegate to `merge()`. The framework counts, how often branches are re-evaluated and offers to call the `widen()`-method, as soon as a configurable threshold is exceeded. This threshold can be set via `enableWidening()`. An action diagram, illustrating in which combination these methods are called can be seen in figure 4.13.

Apart from the standard naive fixpoint iteration as implemented in `NaiveIterativeAlgorithm`, there are also more advanced ideas for performing analyses on control flow graphs. One of these ideas which is particularly interesting for analysing e.g. polynomial invariants is the concept of the semi-naive incremental fixpoint iteration, as presented by Fecht and Seidl in [FS98].

The persistence of the dataflow values is ensured by storing them for each node

separately. Each graph component implements the `Annotatable`-interface which defines routines to store and retrieve arbitrary objects via a string identifier. The `NaiveIterativeAlgorithm` automatically saves the novel value associated with a graph component after executing its callback method based on the identifier string defined via each analysis' `getAnnotationKey()` method. To allow for different user interfaces to perform abstract interpretation analyses, the information about the progress of the analysis is not directly hard-coded into the analysis framework, but instead capsuled as calls to the classes from the `toolkit` sub-package. Each time an analysis is loaded, the framework initialises it by handing over an `AnalysisInitializer` object to the analysis' `initialize()` method. This initialisation object carries references to implementations of the toolkit interfaces, establishing the connection from the analysis to arbitrary user interfaces. Furthermore, the actual control flow graph model `ICfg` is passed from the framework to the analysis class via this initialisation object.

Implementing own custom program analyses thus make it necessary to provide an own subclass to the `NaiveIterativeAlgorithm`, as well as a class with callback functions for the desired architecture and the implementation of the desired complete lattice as subclass of `Lattice`. An analysis author provides his analysis to the VoTUM framework within a `jar`-file, which he deploys into the `analysis` subfolder of the VoTUM application folder. This `jar`-file must be equipped with the Java manifest entries `Analysis-Plugin-Name`, `Analysis-Plugin-IR-Type` and `Analysis-Plugin-Main-Class`, informing VoTUM about the main analysis class which has to be an `AnalysisPlugin`, and the machine architecture, for which this analysis is written.

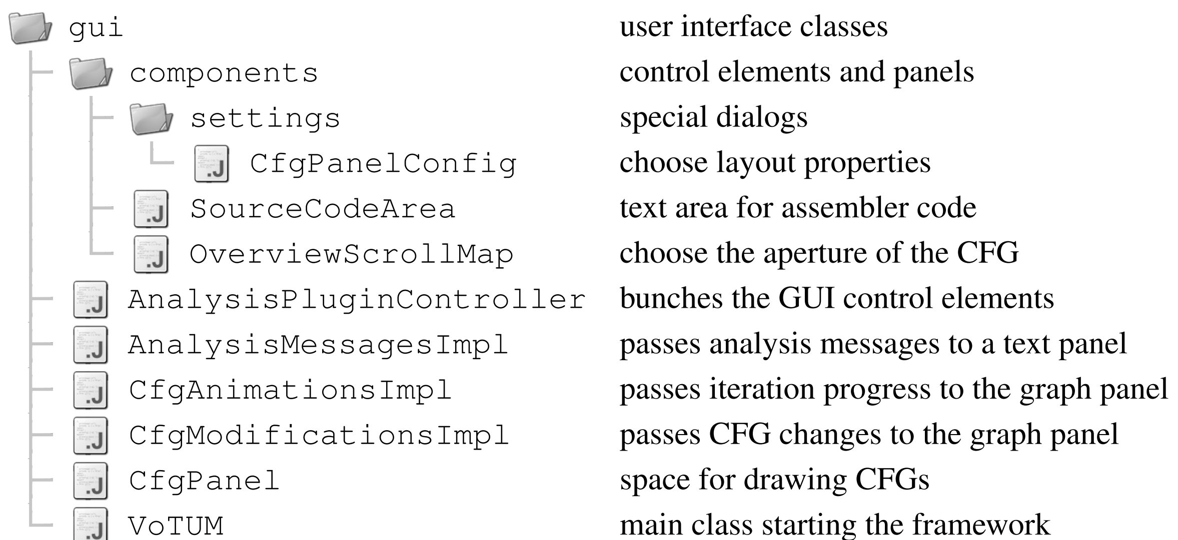


Figure 4.5: outline of package `gui`

The `gui` package provides the graphical user interface for performing visually animated program analyses and transformations. The central class for this package is `VoTUM`, which starts the graphical client for performing animated program analyses

and transformations, which can be seen in figure 4.8. It offers a source code view via `SourceCodeArea`, tabs with `CfgPanels` for the particular control flow graphs themselves, a panel for the analysis messages via `AnalysesMessagesImpl` and a toolbar related to basic control flow graph management. This upper toolbar (as in figure 4.6) offers to load and store control flow graphs, gives a handle to the configuration dialog, can be used for zooming into the flow graph and offers a bird's eye overview control via `OverviewScrollMap` (c.f. figure 4.7).

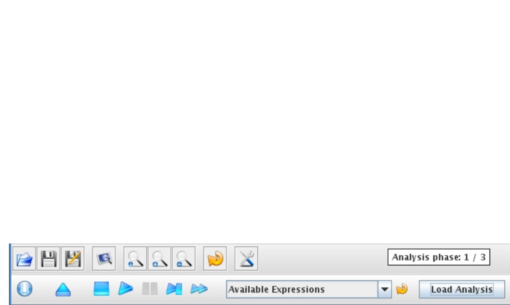


Figure 4.6: CFG management and analysis control toolbars

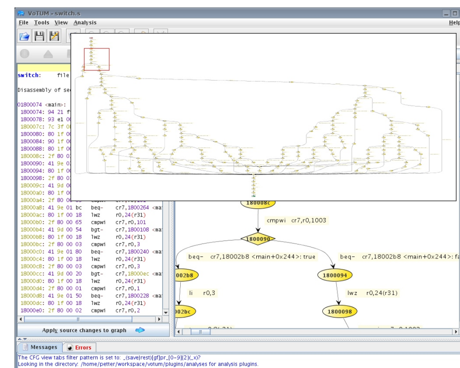


Figure 4.7: Bird's eye control

The main window's `AnalysisPluginController` takes over the job of loading and communicating with the analysis itself. Therefore it provides a specific toolbar with play and pause buttons like the lower one in figure 4.6. It relays the control commands received via this toolbar to the embedded `AnalysisPhase` object. All plugins concerning back-end architectures or program analyses and transformations are brought together via this user interface. The `AnalysisPluginController` will load the analyses from the application's `analysis` sub-folder, identifying the analysis classes with the help of the manifest files of each analysis plugin's `jar` container. Such an `AnalysisPlugin` consists of a sequence of analyses in the form of `AnalysisPhase` classes.

Finally, there is the `compilers` package, containing concrete representations of machine architectures for VoTUM and the interface files for providing own machine architecture plugins. The entry point for an externally provided analysis consists in `CfgCompilerPlugin`, which essentially consists in a method, providing an `ICfg` class, which represents the program, which was loaded from a file. Concerning architecture plugins, the initial class is the `CfgCompilerPlugin` which has to be overwritten in order to provide custom edges in a control flow graph. The central method to override is the `load()` method which is responsible for parsing the committed file for a set of `Cfg`-objects and join them into an `ICfg`, which is returned by `getICfg()`. Besides this rudimentary requirement, architecture plugins provide own subclasses of `CfgEdge` and an `EdgeVisitor`-derived own class of callback methods for these instructions. Following this scheme, we find two pre-provided architectures in the sub-packages

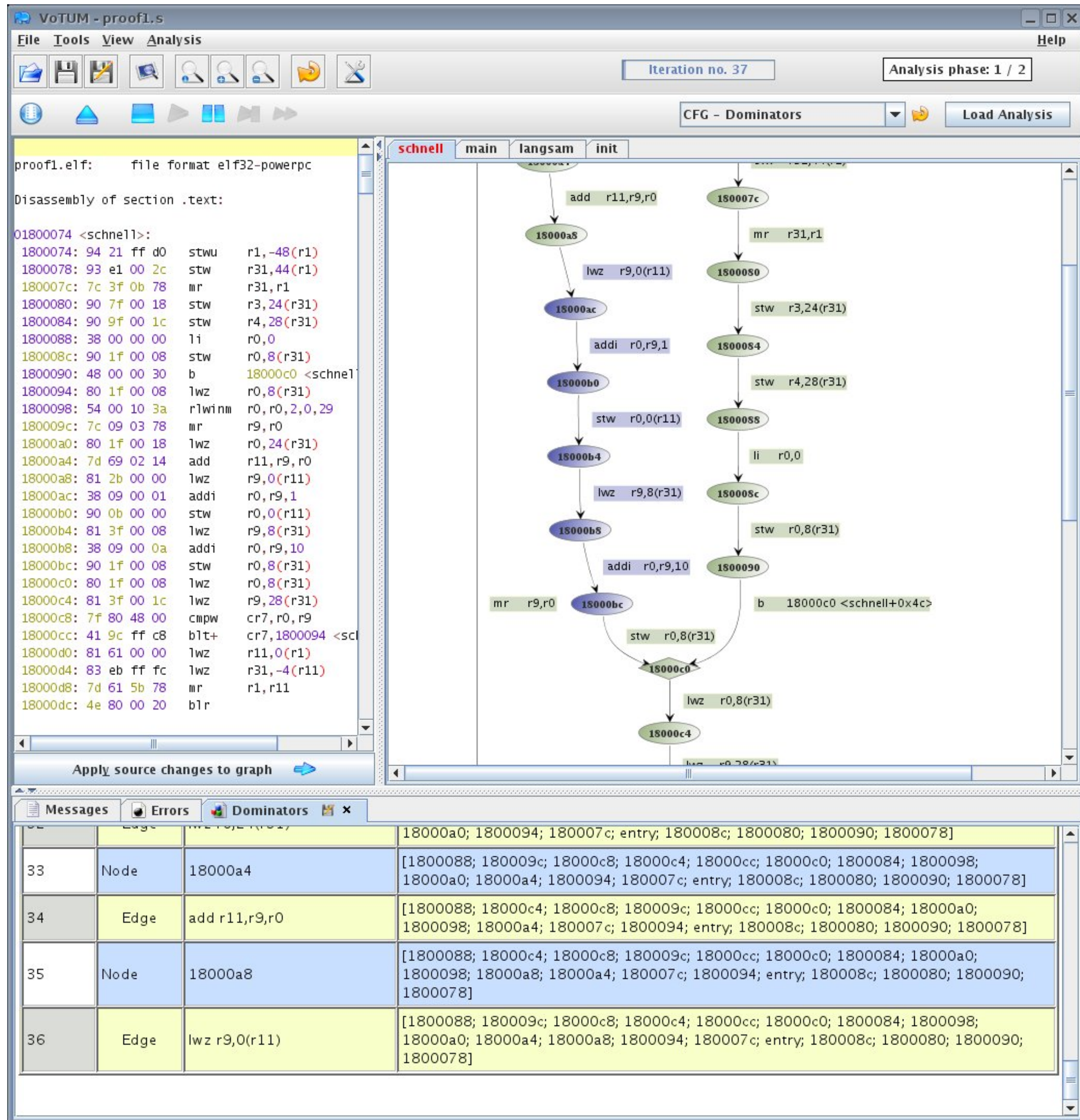


Figure 4.8: VoTUMs main screen

`cmac` and `ppc`. The first architecture *Cmac* extracts the control flow from an ANSI C-compiler [ANS89], developed within student labs [ANS89]. Its machine architecture is geared to the one used for a lecture in program optimisation at TUM, and still work in progress at the time of writing.

The second architecture *PPC* is based on the real hardware architecture *PowerPC* designed by Apple, IBM and Freescale [Fre05]. The class `PowerPCDisassemblyPlugin` implements a proper PPC disassembly parser based on the hexadecimal code sections that are output by the GCC objdump disassembler

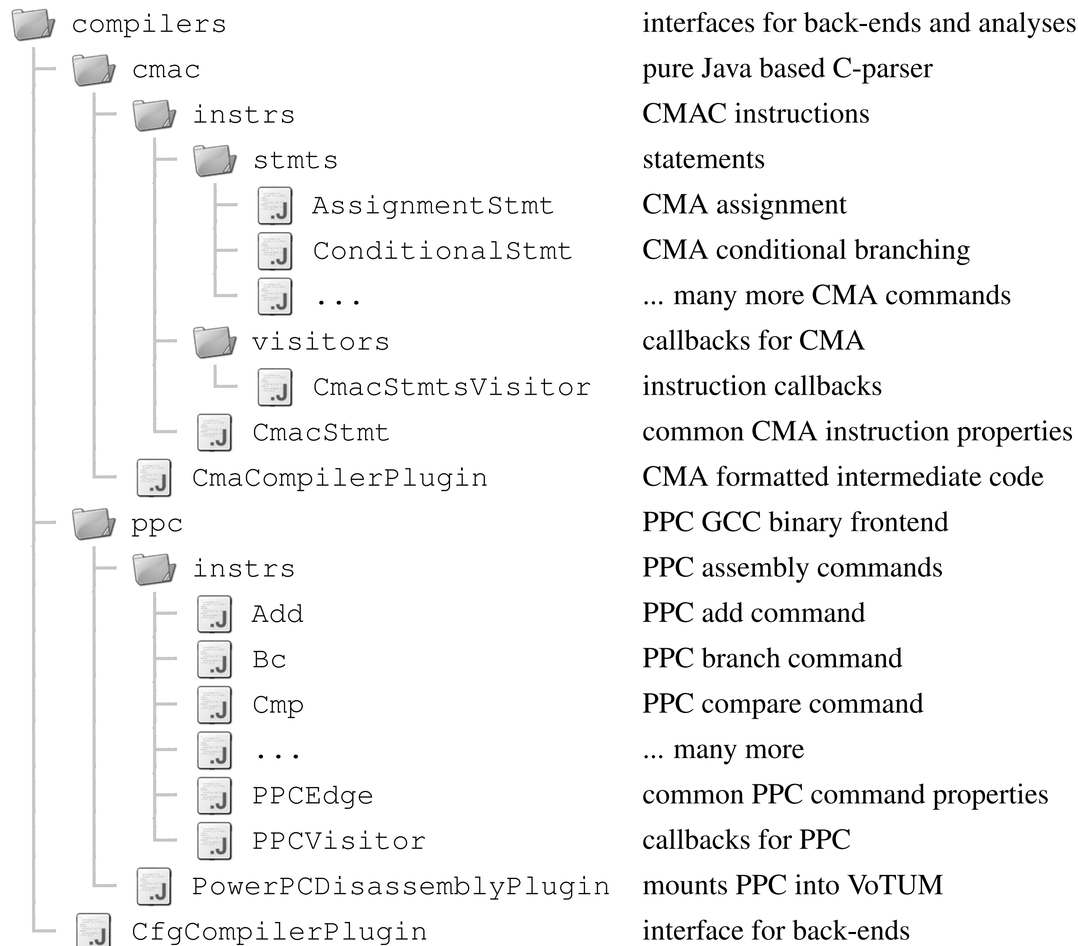


Figure 4.9: outline of package `plugins`

which is used as an ELF parser [TIS95] in this context. Again, this package comes with an implementation of the different instructions as subclasses of `CfgEdge` already providing the common attributes within the `PPCEdge`, and a callback interface via `PPCVisitor`. Since in this architecture one has to cope with all the strange phenomenons of real-world code, as indirect jumps or jump tables, this frontend cannot produce an exact unambiguous control flow graph, but instead an `ICfg` over-approximation, guaranteed to comprehend the concrete program run. This plugin is developed together with Mihaila for his diploma theses [Mih09].

Fixpoint analysis in detail

Implementing analyses in VoTUM is mostly based on deriving an own class from the `NaiveIterativeAlgorithm` class. This class encapsulates the code that traverses the control flow graph, updates and propagates the data-flow values along its path. It is

in fact crucial to know, how in detail the particular elements of the control flow graph are traversed and which callback methods are called when, especially as they can be configured via strategies and analysis directions. It is thus beneficial to have another closer look at the detailed relations between the `NaiveIterativeAlgorithm` class and its collaborating classes:

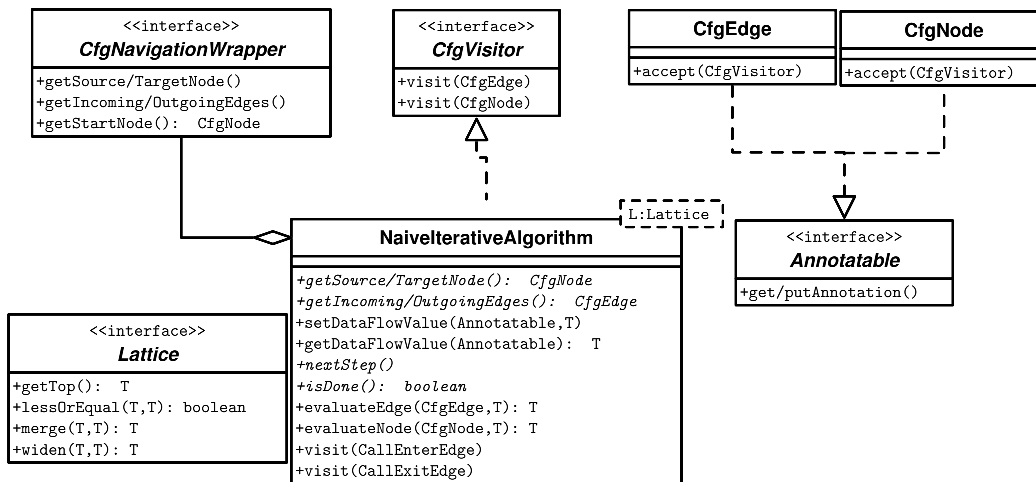


Figure 4.10: Internal details of the fixpoint analysis in VoTUM

The main fixpoint iteration is performed via a simple loop in the framework which calls the `NaiveIterativeAlgorithm`'s `nextStep()` method as long as its `isDone()` method returns **`false`**. Internally, this interface encapsulates a worklist, managing nodes in the graph, which have to be re-evaluated. The class `NaiveIterativeAlgorithm` finds and fetches the next control flow graph element to reevaluate next via the `CfgNavigationWrapper` which regards the analysis' strategy and direction. The `NaiveIterativeAlgorithm` by itself delegates the queries about the control flow graph's structure to its embedded `CfgModel` which links the analysis to a concrete graph structure. The graph component which was obtained from the `CfgNavigationWrapper` triggers through its `accept` method a call to `NaiveIterativeAlgorithm`'s particular `visit()` method. The `visit()` methods are in essence responsible for receiving incoming data-flow values, calling the transformers for this value and performing the merge or widening with the former data-flow value that was associated with this control flow element before. The `visit()` method first calls the `evaluateEdge()`, respectively `evaluateNode()`, methods to permit the user to specify code, to transform the data-flow value according to the kind of the control flow graph element which is actually traversed. It returns the user generated data-flow value which is now managed by the framework. By comparing this new data-flow value with its predecessor via `Lattice`'s `lessOrEqual()` method, it is decided whether the current data-flow value represents a contributing update and whether the dependent control flow elements have to be re-evaluated w.r.t. this update. In case that the currently re-evaluated object is a `CfgNode`, the framework checks

whether the node is a confluence node, and if so, calls the `widen()` method instead of the `merge()` method to attain a new data-flow value. Finally, in order to make the results accessible from the outside, this updated data-flow value is stored associated to the particular control flow element via its `Annotatable`-interface. Abstract pseudo code of the fixpoint iteration as it is performed by the VoTUM framework can be found in figure 4.11.

```
queue.enqueue ( startnode, top )
while (! queue.empty() ) do
  (node,newdata) := queue.poll()
  newdata := evaluateNode ( node , newdata )
  olddata := getDataflowValue ( node )
  if ( ! olddata ≤ newdata )
    if ( wideningthreshold < count++ )
      newdata := widen ( olddata, newdata )
    else
      newdata := merge ( olddata, newdata )
    fi
    setDataflowValue ( node, newdata )
    forall ( edge ∈ outgoing ( node ) ) do
      edgedata := evaluateEdge ( edge, newdata )
      queue.enqueue ( targetNode( edge ), edgedata )
    od
  fi
od
```

Figure 4.11: pseudo code for fixpoint iteration in VoTUM

In the actual implementation especially the `visit()` methods perform even more tasks, e.g. triggering notifications of the graphical animation component to show the progress of the iteration, but this is omitted here, as they do only affect the graphical animation but do not change any vital part of the analysis semantics. Altogether, the VoTUM fixpoint analysis core sports a very complex structure and the call sequence is quite long and crosses many object boundaries. As long as there is no need to change details of the internal interaction of the different components of this framework, there is only a small subset of classes and methods with which one comes in contact. This interface is presented in the following section.

Typical usage patterns

VoTUM's typical usage is in most cases focused on developing new analyses for a specific instruction set with standard naive iterative fixpoint iteration. The typical interaction which happens when using VoTUM for performing dataflow analyses in control flow graphs can be found in figure 4.12. The classes typically provided by an

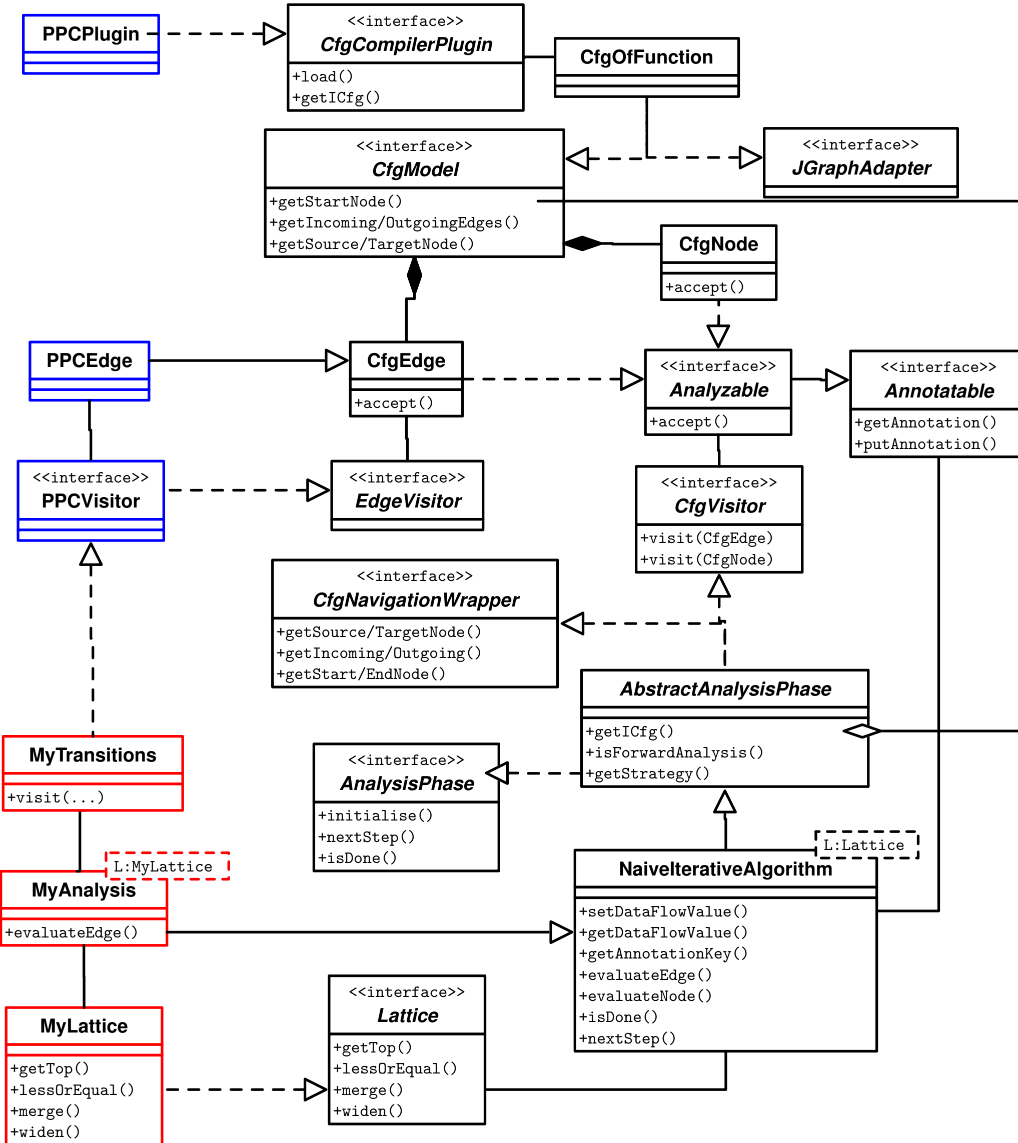


Figure 4.12: Interaction in the VoTUM framework

author of a program analysis are marked in red. Usually, one has to provide an own specific lattice subclass, defining an own data structure as well as an appropriate least upper bound operation `merge()` and a partial order relation. In case that the data structure needs to be widened in the course of the analysis, one also has to provide a `widen()` method which can in every other case simply delegate to the `merge()` method. The connection between VoTUM's fixpoint analysis and own transfer functions is established via a subclass of `NaiveIterativeAlgorithm`. There, one can override the general `evaluateEdge()` method and delegate the transformation of the dataflow value to a specialised subclass of the `EdgeVisitor`, where each particular instance of the target architecture's instructions features a callback routine. By overriding

the callbacks in this visitor, one integrates his own transfer functions for the chosen architecture into the fixpoint analysis. Optionally, if more precise control over the process of selecting the next control flow element is necessary, one may implement an own subclass of `AbstractAnalysisPhase`, but in most cases a default strategy as the ones from `FlowControlStrategy` will be fine.

In order to pre- or post-process the control flow graph and the analysis environment appropriately, it is important to understand the sequence of calls which happen in the framework. A rough sketch of the most important calls is provided in figure 4.13. It shows the order in which control is transferred to the different methods, mainly of `NaiveIterativeAlgorithm`. The first block serves for initialisation of the environment, as a prequel to each fixpoint iteration. First, there is the lattice, which is queried for its top and bottom elements. These elements are then distributed all over the control flow graph via the `initEdge()` and `initNode()` methods, just to be followed by the initialisation of the start node – by default initialised with the top element. After this preparation phase, the main loop of the analysis starts, re-evaluating dataflow values for particular program points, and queueing each dependent control flow graph element for re-evaluation, if the recently evaluated dataflow value is not less or equal than its predecessor. In this diagram, the methods that lend themselves to be overridden are depicted in red. The choice which callback to override is crucial as for example at the beginning of the initialisation phase the control flow graph will neither be fully equipped with a bottom dataflow value at each node or edge nor will the analysis have been specified the lattice which is to be used for the iteration itself.

In case that a new architecture is to be included into the VoTUM framework, it is usually a completely different set of classes which is in the centre of interest. In figure 4.12, the classes that provide a specific instruction set are marked in blue. A typical machine architecture plugin implements the `getICfg()`-method such that it returns a control flow graph that is composed of standard `CfgNode`s and custom `CfgEdge`s that are specific for this architecture. In order to provide program analyses with a handle to the specific instructions, it is common for architecture plugins to provide an own interface for callback methods which can be used in a visitor pattern to hook up to the different instruction callbacks in a concrete analysis. In this way, the VoTUM framework imposes no boundaries on the kind of architectures, which may be analysed with it, but only fixes the kind of graph structures, which may be achieved with the help of the given classes.

4.3.3 Example analysis

In this section, a small example of an analysis of PPC assembly code should demonstrate the use of the VoTUM framework for an interval analysis. For analysing intervals, we use a naive fixpoint iteration, as implemented in `NaiveIterativeAlgorithm`. Thus, we start with an adapted subclass, `IntervalAnalysis`. It configures:

- a purely intraprocedural analysis
- analysis direction as forward
- widening to be used, when a loop has been traversed more than 5 times

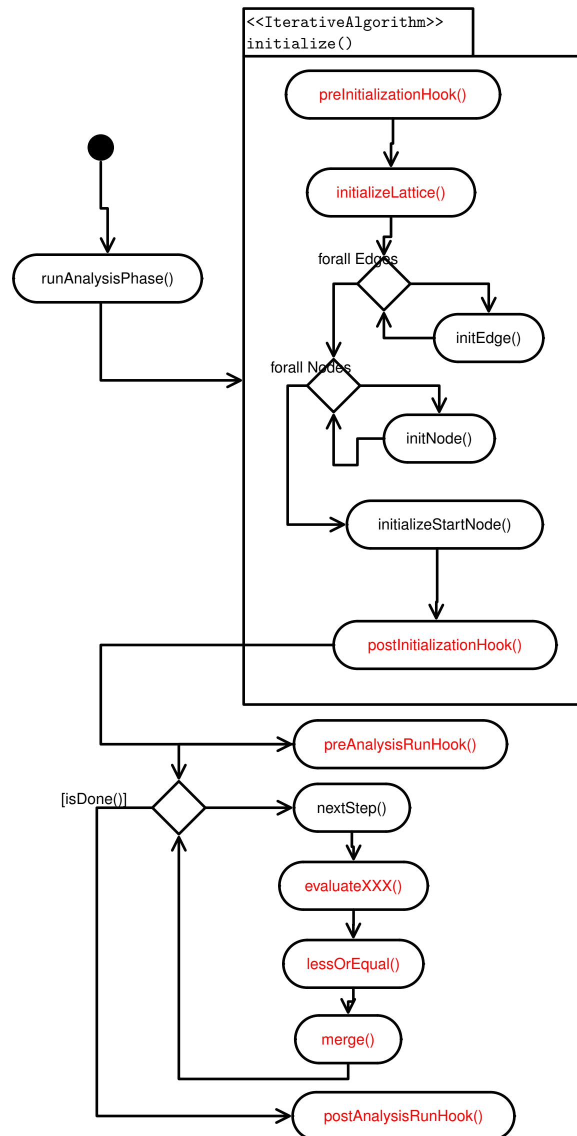


Figure 4.13: Sequence of customisable callbacks

- `RegIntervalLattice` as lattice for this analysis
- initialisation at program start with the top element
- the delegation to `PPCCallbacks` for all transfer functions

The respective Java code to specify such an analysis is provided here, omitting the exact implementation of the transfer functions from `PPCCallbacks`, which can be found in detail in the VoTUM source code:

```
1  /**
2   * Interval analysis on PPC
3   */
4  package myanalysis;
5  import java.util.*;
6  import ...votum.analyses.algorithms.NaiveIterativeAlgorithm;
7  import ...votum.analyses.carriers.Lattice;
8  import ...votum.analyses.carriers.interval.Interval;
9  import ...votum.analyses.ppc.interval.carriers.*;
10 import ...votum.compilers.ppc.machine.Register;
11 import ...votum.graph.*;
12
13 public class IntervalAnalysis extends NaiveIterativeAlgorithm<RegInterval> {
14    private final PPCCallbacks m_transferFunctions = new PPCCallbacks();
15    private RegIntervalLattice m_lattice;
16    /**
17     * Set the widening threshold, direction and strategy
18     * for this analysis.
19     */
20    public IntervalAnalysis() {
21      super(Direction.Forward,FlowControlStrategy.Intraproc);
22      enableWidening(5);
23    }
24
25    @Override public String getPhaseDescription() {
26      return "finds safe intervals for register values." ;
27    }
28    @Override public String getPhaseName() {
29      return "Interval analysis" ;
30    }
31    @Override protected String getDataFlowValueKey() {
32      return IntervalAnalysis.s_annotationKey;
33    }
34
35    /**
36     * Initialize & set lattice with all relevant PowerPC registers
37     */
38    @Override protected RegIntervalLattice initializeLattice() {
39      // First the GPRs
40      Set<Register> regs = EnumSet.range(Register.GPR0, Register.GPR31);
41      // Now the special purpose registers
42      regs.add(Register.LR);  // Link register
43      regs.add(Register.CTR); // Counter register
44      regs.add(Register.ZERO);// Constant zero register
45      m_lattice = new RegIntervalLattice(regs);
46      return m_lattice;
47    }
48
49    /**
50     * Delegate to the transfer functions for the PPC instructions.
51     */
52    @Override protected RegInterval evaluateEdge(CfgEdge edge, RegInterval inData) {
53            return edge.accept(m_transferFunctions, inData);
54    }
55
56    /**
57     * Set the mappings for all the registers to top at program start.
58     */
59    @Override protected RegInterval initializeStartNode(CfgNode node) {
60            return getLattice().getTop();
61    }
62
63    /**
64     * implements interval transfer functions for the PPC architecture
65     */
66    private class PPCCallbacks extends PPCSimpleVisitorAdapter<RegInterval> {
67      ...
68    }
69 }
```

Apart from configuring the fixpoint iteration, one needs to provide the lattice for the interval analysis, which is a mapping from registers to intervals of integers. Mappings are essentially provided by the framework class `MapLattice`, from which we have derived the subclass `RegIntervalLattice`. For each register, this class manages an interval, with the help of `IntervalLattice`, which manages merging, widening and comparison of particular intervals. The implementation of the `Interval` class itself can be found in the framework, it contains an implementation of standard interval arithmetics. All we need to specify for VoTUM, apart from the interval arithmetics itself can be found in the `IntervalLattice` class:

```
1  package myanalysis;
2  import ..votum.analyses.carriers.Lattice;
3
4  public class IntervalLattice implements Lattice<Interval> {
5
6      @Override public Interval getBottom() {
7              return Interval.s_bottom;;
8      }
9
10     @Override public Interval getTop() {
11             return Interval.s_top;
12     }
13
14     @Override public int getHeight() {
15             return 0;
16     }
17
18     @Override public Interval merge(Interval first, Interval second) {
19             return first.union(second);
20     }
21
22     @Override public Interval widen(Interval widener, Interval element) {
23             return element.widenToNextPowerOf2(widener);
24     }
25
26     @Override public boolean lessOrEquall(Interval first, Interval second) {
27             return second.contains(first);
28     }
29 }
```

Operating the GUI

A fixpoint iteration in VoTUM can be controlled via the buttons of the main window's toolbars (c.f. figure 4.6 on page 106). The upper bar provides general control elements for navigation within a control flow graph, zooming in and out of a graph, as well as configuring the whole application. Then there is a second toolbar dedicated to controlling the progress of the particular analyses. A drop-down menu offers all available analyses, while the VCR-like control elements operate the steps of the analysis – offering a single step mode as well as a continuous execution mode, either of them with or without animations.

While an analysis is running, the main window, as e.g. in figure 4.14 displays the current state of the control flow graph w.r.t. the progress of the analysis. Colours encode,

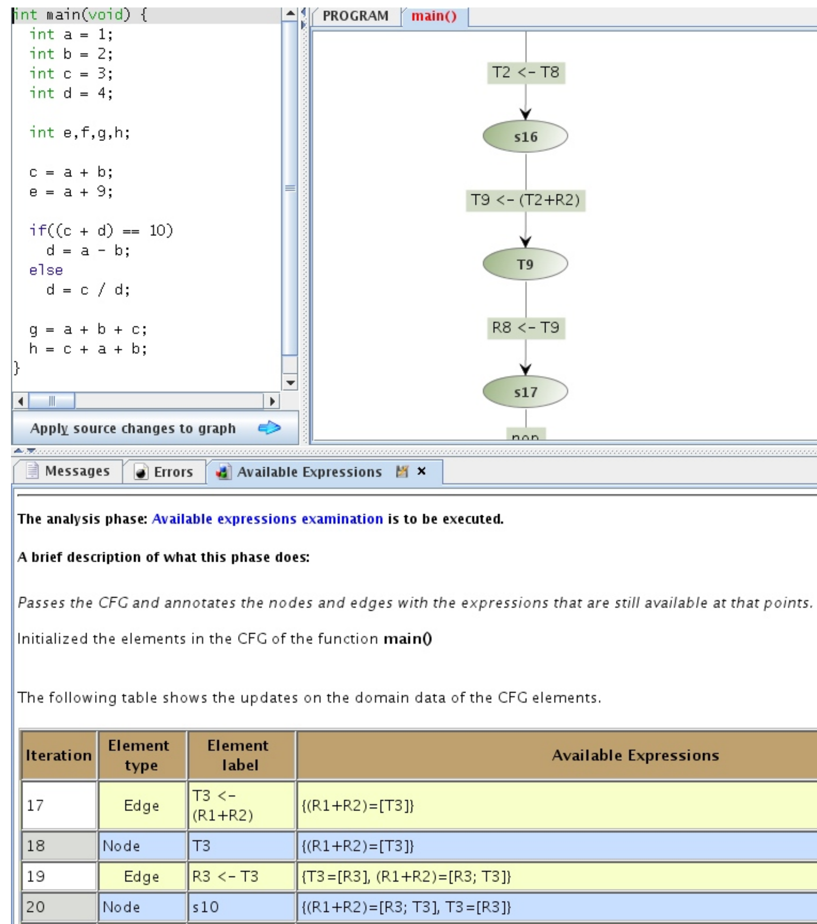


Figure 4.14: Source code vs. control flow graph

whether a node or edge of the graph already has been evaluated, as well as it shows pending updates which wait for processing. In case that there are several sub-procedures in one loaded file, they can be found as different tabs stacked onto each other in the middle part of the main window.

At the bottom, one can find more tabs which provide textual output of the analysis results. One tab records e.g. the protocol of the analysis, showing which dataflow value emerged at which control flow graph element. Apart from the mandatory errors and message tabs, there is also the possibility to add an own custom tab here.

Apart from marking the currently updated program state, the main control flow graph part also provides means to inspect the last passed dataflow value by hovering the mouse cursor over an edge or a node. As can be seen in figure 4.15, a pop-up window then shows all available information (a.k.a. annotation) about this element. Control flow element specific operations can be performed by right-clicking an element. Then, a context menu appears which enables to control e.g. break points for the current analysis as well as it offers to scroll and mark in the active code window the instruction

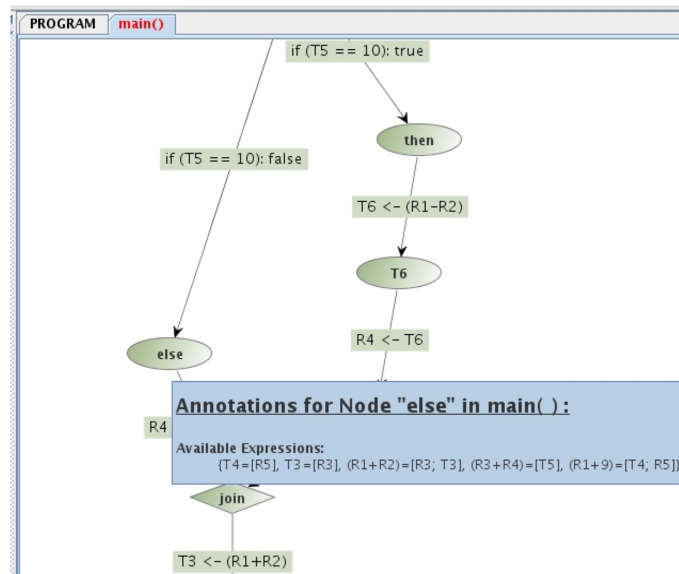


Figure 4.15: Controlling results via mouseover

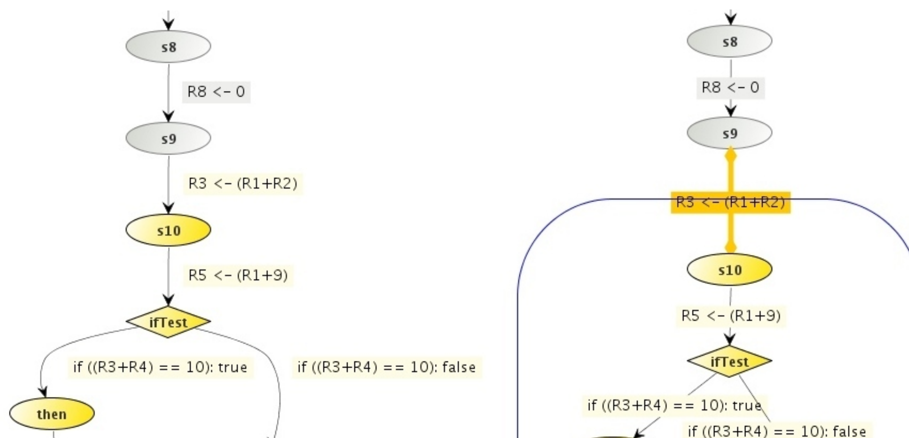


Figure 4.16: Making room for two edges

corresponding to the particular edge.

When it comes to transformations of control flow graphs, the control flow graph view supports the animation of the changes which are performed on particular elements. It features the animation of deletion, creation and union of elements. In figure 4.16, one can see parts of the animation of an insertion of an edge. At first, the sub-graph below the edge which is split will be framed by a blue box and moved out of the way for the manipulation.

Afterwards, as can be seen in figure 4.17 there is a new node *T3* inserted which gets equipped with two new edges to the rest of the control flow graph. Currently active

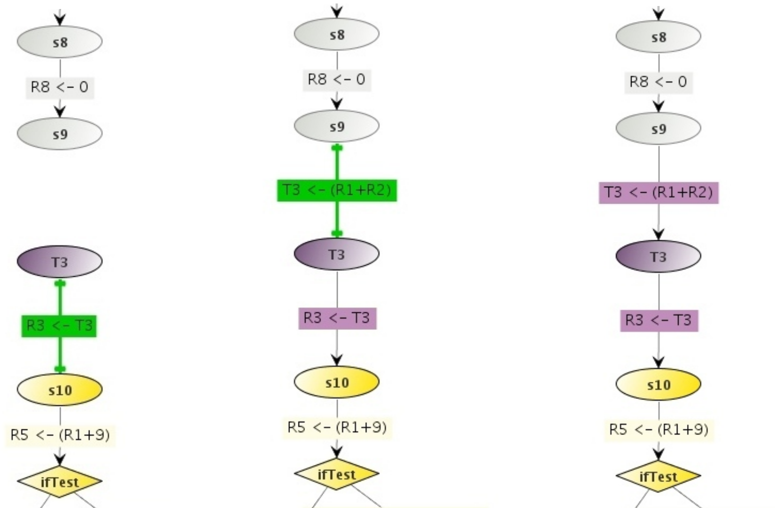


Figure 4.17: Insertion of two edges

edges are marked in green while recently modified ones appear in purple.

4.4 Further Improvements

Of course there is sill a lot of work going on to improve VoTUM. One of the big open points is still the aspect of architecture plugins. Although VoTUM has a stable model for representing programs as control flow graphs and offers an interface to tie concrete machine plugins to VoTUM what is still missing is a broader support of different concrete machine architectures. After the completion of the PowerPC plugin, for which there is still work going on, the addition of plugins for x86 compatible code as well as a plugin for LLVM code is planned.

Apart from that, recent developments in the Java sector allow the use of new different languages to be used on the JVM. This would enable to mix functional and object oriented languages in use within VoTUM, and offer e.g. a functional specification for transfer functions, which would make the rather verbose type declarations of Java obsolete.

Chapter 5

Conclusion

We have presented several techniques and applications related to the analysis of polynomial equalities and program analysis for programs with procedures. We now summarise the results that are provided within this work and give a perspective of further work, which is connected to the approaches in this thesis.

Summary

We achieved several goals in this thesis. In the first part of the thesis, we gave an overview over all techniques which we applied to analyse programs for polynomial equalities in presence of procedure calls. We furnish our program class with procedure calls, differentiated scoping rules between caller's and callee's local and global variables as well with treatments for particular guards. We come up with a forward oriented characterisation of all valid polynomials, and then acquire an effective algorithm to infer all valid polynomial equalities of a bounded degree in a practical relevant program class.

In the second part, we focused more on the differences which arise if switching from arithmetics over $\mathbb{Q}$ to arithmetics over $\mathbb{Z}_{2^w}$. After clarifying the arithmetic properties of $\mathbb{Z}_{2^w}$, we come up with the concept of a normal-reduced generator system, which allows to perform approximate membership tests good enough to establish finite increasing chains. We further provide a practically feasible algorithm to infer polynomial equalities in $\mathbb{Z}_{2^w}$. Theoretically we also come up with a polynomial time algorithm to infer all polynomial equalities in a program.

In the third part, we presented a general framework for analyses of Herbrand equalities featuring procedure calls with global and local variables. For this framework, we provide two implementations of procedure effects which allow to precisely infer all equalities for two different practically relevant program classes.

In the last part, we present the implementation of a practical program analysis framework for the development of interprocedural analyses and the visualisation of the analyses' progress.

Further work

What we have not covered within this thesis are a few topics which might be worth investigating.

Concerning the analysis of polynomial invariants in $\mathbb{Z}_{2^w}$, we have not yet examined the implications of modular arithmetics to forward analyses. Due to the fact that in $\mathbb{Z}_{2^w}$ there are no infinite decreasing chains any more, we expect the forward analysis to yield precise results in this case. However, in order to make this approach attractive, there has to be work done concerning the performance of the ELIMINATION algorithm for residue class rings.

Concerning Herbrand analyses, it might be worth to examine the practical performance of an implementation of the different approaches. Especially for the polynomial based approach, it is not clear whether the observed performance is practically acceptable or not.

For the VoTUM framework, especially the development of the binary frontend has brought interesting new problems up. Crucial for the analysis of binary codes is the safe knowledge about relations between machine registers and stack memory. So far, we still lack a safe interprocedural analysis, certifying which areas of the local stack frame have been corrupted by a procedure call. Another open topic is an analysis of function pointers on low-level code, which is quite exhausting as on low-level code, all integer values may turn out to be addresses or not.

Bibliography

[Abs] AbsInt, Angewandte Informatik GmbH. aiSee Graph Layout Software. `http://www.aisee.com`.

[AM95] Martin Alt and Florian Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *2nd Int. Symp. on Static Analysis (SAS)*, pages 33–50, 1995.

[ANS89] ANSI/ISO. *American National Standard for Programming Languages - C*, 1989. ANSI X3.159-1989.

[AP86] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, 1986.

[App02] Andrew W. Appel. *Modern Compiler Implementation in Java, 2nd edition*. Cambridge University Press, 2002.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Symposium on Principles of Programming Languages (POPL)*, pages 1–11, 1988.

[BBM97] Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87–, 1997.

[BLS96] Saddek Bensalem, Yassine Lakhnech, and Hassen Saidi. Powerful techniques for the automatic generation of invariants. In *8th Int. Conf. on Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*. Springer, 1996.

[BS01] Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.

[BTR02] Douglas C. Bossen, Joel M. Tendler, and Kevin Reick. Power4 system design for high reliability. *IEEE Micro*, 22(2):16–24, 2002.

[BW93] Thomas Becker and Volker Weispfenning. *Gröbner Bases – a computational approach to commutative algebra*. Springer Verlag, New York, 1993.

[CC77] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Recursive Procedures. In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts*, pages 237–277. North-Holland, 1977.

[CC95] Cliff Click and Keith D. Cooper. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196, 1995.

[Col04] Michael Colon. Approximating the algebraic relational semantics of imperative programs. In *11th Int. Symp. on Static Analysis (SAS)*, pages 296–311. Springer-Verlag, LNCS 3146, 2004.

[Col07] M. Colon. Polynomial approximations of the relational semantics of imperative programs. *Science of Computer Programming*, 64:76–96, Elsevier, 2007.

[Cou97] Patrick Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theor. Comput. Sci*, 6, 1997.

[Cou05] Patrick Cousot. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *6th International Conference, Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 1–24, 2005.

[CS69] John Cocke and J. T. Schwartz. *Programming languages and their compilers*. Courant Institute of Mathematical Sciences, New York University, 1969.

[EGK+01] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. In *Graph Drawing*, pages 483–484, 2001.

[FKMK98] Shinji Fukatsu, Yoshifumi Kitamura, Toshihiro Masaki, and Fumio Kishino. Intuitive control of ”bird’s eye” overview images for navigation in an enormous virtual environment. In *VRST*, pages 67–76, 1998.

[Fre05] Freescale Semiconductor. *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture – MPCFPE32B*, 2005. `http://www.freescale.com/files/product/doc/MPCFPE32B.pdf`.

[FS98] Christian Fecht and Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. *European Symposium on Programming (ESOP)*, 1381:90–104, 1998.

[Gar02] Karthik Gargi. A Sparse Algorithm for Predicated Global Value Numbering. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 45–56, 2002.

[Gea99] David M. Geary. *Graphic Java 2, Mastering the JFC: Volume II: Swing, 3rd Edition*. Prentice-Hall, 1999.

[GGSST10] A. Gascón, G. Godoy, M. Schmidt-Schauß, and A. Tiwari. Context unification with one context variable. *J. Symbolic Computation*, 45(2):173–193, February 2010.

[GHJ+93] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Taligent Inc. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of 7th European Conference on Object-Oriented Programing (ECOOP)*, pages 406–431. Springer-Verlag, 1993.

[GKNV93] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Trans. Software Eng.*, 19(3):214–230, 1993.

[GN03] Sumit Gulwani and George C. Necula. Discovering Affine Equalities Using Random Interpretation. In *30th Ann. ACM Symp. on Principles of Programming Languages (POPL)*, pages 74–84, 2003.

[GN04] Sumit Gulwani and George C. Necula. Global value numbering using random interpretation. In *Symposium on Principles of Programming Languages (POPL)*, pages 342–352, 2004.

[GN05] Sumit Gulwani and George C. Necula. Precise Interprocedural Analysis using Random Interpretation. In *32th Ann. ACM Symp. on Principles of Programming Languages (POPL)*, pages 324–337, 2005.

[GN07] Sumit Gulwani and George C. Necula. A polynomial-time algorithm for global value numbering. *Sci. Comput. Program.*, 64(1):97–114, 2007.

[Gol81] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theor. Comput. Sci.*, 13:225–230, 1981.

[Gra91] Philippe Granger. Static Analysis of Linear Congruence Equalities among Variables of a Program. In *Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, pages 169–192. LNCS 493, Springer-Verlag, 1991.

[GS07] Thomas Gawlitza and Helmut Seidl. Precise Relational Invariants Through Strategy Iteration. In *21th Computer Science Logic (CSL)*, pages 23–40, 2007.

[GT07a] Sumit Gulwani and Ashish Tiwari. Assertion checking unified. *Verification, Model Checking and Abstract Interpretation (VMCAI)*, 4349:363–377, 2007.

[GT07b] Sumit Gulwani and Ashish Tiwari. Computing Procedure Summaries for Interprocedural Analysis. *16th European Symposium on Programming (ESOP)*, 4421:253–267, 2007.

[GT09] Guillem Godoy and Ashish Tiwari. Invariant checking for programs with procedure calls. *16th Static Analysis Symposium (SAS)*, 5673:326–342, 2009.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, 1977.

[HS06] Norbert Hungerbühler and Ernst Specker. A generalization of the smarandache function to several variables. *Integers: Electronic Journal Combinatorial Number Theory*, 6, 2006.

[IBM93] IBM. *PowerPC Architecture - The official manual for the PowerPC architecture. Three parts: instruction set architecture, virtual environment architecture, and operating environment architecture, IBM book number SR28-5124-00*, 1993.

[IPS99] Russell Impagliazzo, Pavel Pudlák, and Jiri Sgall. Lower bounds for the polynomial calculus and the gröbner basis algorithm. *Computational Complexity*, 8(2):127–144, 1999.

[Kar76] Michael Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Symposium on Principles of Programming Languages (POPL)*, pages 194–206, 1973.

[KM96] Klaus Kühnle and Ernst W. Mayr. Exponential space computation of gröbner bases. *International Conference on Symbolic and Algebraic Computation (ISSAC)*, pages 63–71, 1996.

[KRS98] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Code motion and code placement: Just synonyms? *7th European Symposium on Programming (ESOP)*, 1381:154–169, 1998.

[KU76] J.B. Kam and J.D. Ullman. Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.

[LD03] Cullen Linn and Saumya K. Debray. Obfuscation of executable code to improve resistance to static disassembly. *ACM Conference on Computer and Communications Security (CCS)*, pages 290–299, 2003.

[Mat89] Hideyuki Matsumura. *Commutative Ring Theory*. Cambridge University Press, 1989.

[Mih09] Bogdan Mihaila. Control flow reconstruction from PowerPC binaries. Master's thesis, Technische Universität München, München, 2009.

[MOPS06] Markus Müller-Olm, Michael Petter, and Helmut Seidl. Interprocedurally analyzing polynomial identities. In *23rd Ann. Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 50–67, 2006.

[MOR01] M. Müller-Olm and O. Rüthing. On the complexity of constant propagation. *10th European Symposium on Programming (ESOP)*, 2001.

[MORS05] M. Müller-Olm, O. Rüthing, and H. Seidl. Checking Herbrand Equalities and Beyond. In *Verification Meets Model-Checking and Abstract Interpretation (VMCAI)*, pages 79–96, 2005.

[MOS02] M. Müller-Olm and H. Seidl. Polynomial constants are decidable. In *9th Static Analysis Symposium (SAS)*, pages 4–19. LNCS 2477, Springer-Verlag, 2002.

[MOS04a] Markus Müller-Olm and Helmut Seidl. Computing polynomial program invariants. *Information Processing Letters (IPL)*, 91(5):233–244, 2004.

[MOS04b] Markus Müller-Olm and Helmut Seidl. A note on Karr's algorithm. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2004.

[MOS04c] Markus Müller-Olm and Helmut Seidl. Precise Interprocedural Analysis through Linear Algebra. In *31st ACM Symp. on Principles of Programming Languages (POPL)*, pages 330–341, 2004.

[MOS05a] M. Müller-Olm and H. Seidl. Analysis of Modular Arithmetic. In *14th European Symposium on Programming (ESOP)*, 2005.

[MOS05b] M. Müller-Olm and H. Seidl. A Generic Framework for Interprocedural Analysis of Numerical Properties. In *12th Static Analysis Symposium (SAS)*, pages 235–250, 2005.

[MOS07] M. Müller-Olm and H. Seidl. Analysis of Modular Arithmetic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5), 2007.

[MOSS05] M. Müller-Olm, H. Seidl, and B. Steffen. Interprocedural Herbrand Equalities. In *14th European Symposium on Programming (ESOP)*, pages 31–45, 2005.

[Pet04] Michael Petter. Berechnung von polynomiellen Invarianten. Master's thesis, Technische Universität München, München, 2004.

[PMBG06] Mila Dalla Preda, Matias Madou, Koen De Bosschere, and Roberto Giacobazzi. Opaque Predicates Detection by Abstract Interpretation. *11th International Conference on Algebraic Methodology and Software Technology (AMAST)*, pages 81–95, 2006.

[RCK04a] E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *Int. ACM Symposium on Symbolic and Algebraic Computation 2004 (ISSAC04)*, pages 266–273, 2004.

[RCK04b] Enric Rodríguez-Carbonell and Deepak Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In *International Symposium on Static Analysis (SAS)*, 2004.

[RCK07] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64:54–75, Elsevier, 2007.

[RKS99] Oliver Rüthing, Jens Knoop, and Bernhard Steffen. Detecting Equalities of Variables: Combining Efficiency with Precision. In *Symposium of Static Analysis (SAS)*, pages 232–247, 1999.

[RL77] John H. Reif and Harry R. Lewis. Symbolic Evaluation and the Global Value Graph. In *Symposium on Principles of Programming Languages (POPL)*, pages 104–118, 1977.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. *Symposium on Principles of Programming Languages (POPL)*, pages 12–27, 1988.

[SB04] Steven Sobek and Kevin Burke. *PowerPC Embedded Application Binary Interface (EABI): 32-Bit Implementation*, 2004. http://www.freescale.com/files/32bit/doc/app_note/PPCEABI.pdf.

[Sin74] David Singmaster. On polynomial functions (mod m). *Journal of Number Theory*, 6:345–352, 1974.

[SKEG05] Namrata Shekhar, Priyank Kalla, Florian Enescu, and Sivaram Gopalakrishnan. Equivalence verification of polynomial datapaths with fixed-size bitvectors using finite ring algebra. In *International Conference on Computer-Aided Design (ICCAD)*, pages 291–296, 2005.

[SKR90] B. Steffen, J. Knoop, and O. Rüthing. The Value Flow Graph: A Program Representation for Optimal Program Transformations. In *3rd European Symp. on Programming (ESOP)*, pages 389–405. Springer-Verlag, LNCS 432, 1990.

[SKR91] Bernhard Steffen, Jens Knoop, and Oliver Rüthing. Efficient code motion and an adaption to strength reduction. *International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Vol.2*, 494:394–415, 1991.

[SSM04] Sriram Sankaranarayanan, Henry. B. Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases. In *ACM SIGPLAN Principles of Programming Languages (POPL)*, 2004.

[SSS02] Manfred Schmidt-Schauß and Klaus U. Schulz. Solvability of context equations with two context variables is decidable. *Journal of Symbolic Computation*, 33(1):77–122, 2002.

[Sta10] Richard M. Stallman. *Using and Porting the GNU Compiler Collection*, 2010. `http://www.skyfree.org/linux/references/gcc-v3.pdf`.

[Ste87] Bernhard Steffen. Optimal run time optimization proved by a new look at abstract interpretation. In *TAPSOFT, Vol.1*, pages 52–68, 1987.

[TIS95] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, 1995.

[Wie07] Oliver Wienand. The groebner basis of the ideal of vanishing polynomials. *to appear in Journal of Symbolic Computation (JSC)*, 2007. Preprint in arxiv math.AC/0801.1177.