Variations on Constants

Als Habilitationsschrift am Fachbereich Informatik
der Universität Dortmund vorgelegt von

Markus Müller-Olm

August 2002
# Contents

1 Motivation .............................................................................................................. 1

2 Introduction .............................................................................................................. 5

3 A Hierarchy of Constants ....................................................................................... 17
   3.1 A Taxonomy of Constants ................................................................................. 20
      3.1.1 Flow Graphs ........................................................................................... 20
      3.1.2 May- and Must-Constants ....................................................................... 21
      3.1.3 Weakened Constant Detection Problems .................................................. 23
      3.1.4 Classes of Integer Constants .................................................................... 25
   3.2 Known Results ................................................................................................. 26
   3.3 New Undecidability Results ............................................................................. 28
   3.4 New Intractability Results ............................................................................... 29
   3.5 Summary .......................................................................................................... 33

4 Deciding Constants ................................................................................................. 35
   4.1 Presburger and Polynomial Constants ............................................................. 36
   4.2 Presburger-Constant Detection ....................................................................... 37
   4.3 A Generic Algorithm ....................................................................................... 40
   4.4 Detection of Presburger Constants .................................................................. 43
   4.5 A Primer on Computable Ideal Theory ........................................................... 47
   4.6 More About \( \mathbb{Z}[x_1, \ldots, x_n] \) ..................................................................... 48
      4.6.1 \( \mathbb{Z}[x_1, \ldots, x_n] \) as a Complete Lattice ................................................... 48
      4.6.2 Zeros ...................................................................................................... 49
      4.6.3 Substitution ............................................................................................. 50
      4.6.4 Projection ............................................................................................... 50
   4.7 Detection of Polynomial Constants ................................................................. 51
   4.8 Conclusion ....................................................................................................... 53

5 Limits of Parallel Flow Analysis .............................................................................. 57
   5.1 A Motivating Example ..................................................................................... 59
   5.2 Parallel Programs ............................................................................................. 60
   5.3 Interprocedural Copy-Constant Detection ....................................................... 61
## 5.3.1 Two-Counter Machines .............................................. 61
5.3.2 Constructing a Program ............................................. 63
5.3.3 Correctness of the Reduction ...................................... 66
5.4 Intraprocedural Copy-Constant Detection ............................. 67
5.5 Copy-Constant Detection in Loop-Free Programs ..................... 70
5.6 Beyond Fork/Join Parallelism ........................................... 71
5.7 Owicki/Gries-style Program Proofs .................................... 72
5.8 Correctness of the Reduction in Section 5.3 ......................... 73
5.8.1 Enriching the Program ............................................... 73
5.8.2 The Proof Outlines .................................................. 73
5.8.3 Interference Freedom ............................................... 76
5.9 Correctness of the Reduction in Section 5.4 ......................... 77
5.9.1 Enriching the Program ............................................... 77
5.9.2 An Auxiliary Predicate .............................................. 78
5.9.3 Proof Outline for $\pi_0$ ............................................. 79
5.9.4 Proof Outline for $\pi_i(r)$ ........................................... 80
5.9.5 Proof Outline for $Main$ ............................................. 81
5.9.6 Interference Freedom ............................................... 82
5.10 Conclusion ............................................................... 83

### 6 Parallel Flow Graphs .................................................. 85
6.1 Parallel Flow Graphs .................................................... 86
6.2 Operational Semantics ................................................. 88
6.3 Atomic Runs ............................................................ 90
6.4 The Run Sets of Ultimate Interest ..................................... 91
6.5 The Constraint Systems ................................................. 92
6.5.1 Same-Level Runs .................................................... 92
6.5.2 Inverse Same-Level Runs .......................................... 94
6.5.3 Two Assumptions and a Simple Analysis .......................... 95
6.5.4 Reaching Runs ...................................................... 97
6.5.5 Terminating Runs ................................................... 98
6.5.6 Bridging Runs ....................................................... 99
6.5.7 The General Case ................................................... 100
6.6 Discussion ............................................................ 102

### 7 Non-Atomic Execution ................................................. 105
7.1 Modeling Non-Atomic Execution ...................................... 107
7.2 A Motivating Example .................................................. 109
7.3 The Domain of Non-Atomic Run Sets ................................ 110
7.3.1 Base Statements ..................................................... 111
7.3.2 Sequential Composition ............................................ 112
7.3.3 Interleaving Operator .............................................. 112
7.3.4 Pre-Operator ....................................................... 112
CONTENTS

7.3.5 Post-Operator .................................................. 113
7.4 Conclusion ....................................................... 113

8 Dependence Traces .............................................. 115
  8.1 Transparency and Dependences ............................. 117
  8.2 Dependence Traces ........................................... 118
  8.3 Implication Order ............................................ 120
  8.4 Subsumption Order .......................................... 121
  8.5 A Lattice of Antichains ..................................... 122
  8.6 Short Dependence Traces ................................... 125
  8.7 The Abstract Domain ....................................... 128
  8.8 Pre-Operator ................................................ 130
  8.9 Post-Operator ............................................... 133
  8.10 Sequential Composition ................................... 133
  8.11 Interleaving ................................................. 135
    8.11.1 Complementary Dependence Traces ................... 136
    8.11.2 Interleaving Operator ................................ 137
    8.11.3 Soundness Lemmas ..................................... 137
    8.11.4 Completeness Lemmas ................................ 141
    8.11.5 Proof of Theorem 8.35 ................................ 144
  8.12 Base Edges .................................................. 146
  8.13 Run-Time .................................................... 147
  8.14 Conclusion .................................................. 148

9 Detecting Copy Constants and Faint Code .................. 149
  9.1 Copy-Constant Detection .................................. 150
  9.2 Faint-Code Elimination .................................... 152
  9.3 Run-Time .................................................... 155
  9.4 Conclusion .................................................. 157

10 Complexity in the Non-atomic Scenario .................... 159
  10.1 The SAT-Reduction ......................................... 160
  10.2 Towards Stricter Lower Bounds ............................ 163
    10.2.1 Assignment Statements that Propagate Twice ....... 164
    10.2.2 Propagating Runs of Exponential Length .......... 165
  10.3 Summary .................................................... 166

11 Conclusion ..................................................... 167
  11.1 Future Research ............................................ 169

A Constraint-Based Program Analysis ........................ 181
Chapter 1

Motivation

Computer science is concerned with design of programs for a wide range of purposes. We are, however, not done once a program is constructed. For various reasons, programs need to be analyzed and processed after their construction. First of all, we usually write programs in high-level languages and before we can execute them on a computer they must be translated into machine code. In order to speed up computation or save memory, optimizing compilers perform program transformations relying heavily on the results of program analysis routines. Secondly, due to their ever increasing complexity, programs must be validated or verified in order to ensure that they serve their intended purpose. Program analysis (in a broad sense) is concerned with techniques that automatically determine run-time properties of given programs prior to run-time. This includes flow analysis, type checking, abstract interpretation, model checking, and similar areas.

By Rice’s theorem [68, 27], every non-trivial semantic question about programs is undecidable in a universal programming language. At first glance, this seems to imply that automatic analysis of programs is impossible. However, computer scientists have found at least two ways out of this problem. Firstly, we can use weaker formalisms than universal programming languages for modeling systems such that interesting questions become decidable. Important examples are the many types of automata studied in automata theory and Kripke structures (or labeled transition systems) considered in model checking. Secondly, we can work with approximate analyses that do not always give a definite answer but may have weaker (but sound) outcomes. Approximate analyses are widely used in optimizing compilers.

It is an interesting problem to assess the precision of an approximate analysis, i.e., how exact the delivered answers are. One approach is to define an abstraction of programs or program behavior that gives rise to weaker but sound information and to prove that the analysis is exact with respect to this abstraction (cf. Fig. 1.1). The loss of precision can then be attributed to and measured by the employed abstraction. This scheme has been used in the literature in a number of scenarios, e.g., [36, 72, 39, 73, 74, 22].
Figure 1.1: Using an abstraction to assess the precision of an approximate analysis.

The scheme of Fig. 1.1 allows us to make meaningful statements on approximate analyses independently of specific algorithms: by devising abstractions of programs, we obtain well-defined weakened analysis problems and we can classify these problems with the techniques of complexity and recursion theory. The purpose of such research is twofold: on the theoretical side, we get insights on the trade-off between efficiency and precision in the design of approximate analyses; on the practical side, we hope to uncover potential for the construction of more precise (efficient) analysis algorithms.

In this thesis we study weakened versions of constant propagation. The motivation for this choice is threefold. Firstly, the constant-propagation problem is easy to understand and of obvious practical relevance. Hence, uncovering potential for more precise constant-propagation routines is of intrinsic interest. Secondly, there is a rich spectrum of natural weakened constant-propagation problems. On the one hand, we can vary the set of algebraic operators that are to be interpreted by the analysis. On the other hand, we can study the resulting problems in different classes of programs (sequential or parallel programs, with or without procedures, with or without loops etc.). Finally, results for the constant-propagation problem can often be generalized to other analysis questions. For instance, if as part of the abstraction we decide not to interpret algebraic operators at all, which leads to a problem known as copy-constant detection, we are essentially faced with analyzing transitive dependences in programs. Hence, results for copy-constant detection can straightforwardly be adapted to other problems concerned with transitive dependences, like faint-code elimination or program slicing.

In this thesis we combine techniques from different areas like linear algebra, computable ring theory, abstract interpretation, program verification, complexity theory, etc. in order to come to grips with the considered variants of the constant-propagation problem. More generally, we believe that combination of techniques is the key to further progress in automatic analysis and constant-propagation allows us to illustrate this point in a theoretical study.
Let us briefly outline the main contributions of this thesis:

**A hierarchy of constants in sequential programs.** We explore the complexity of constant propagation for a three-dimensional taxonomy of constants in sequential imperative programs that work on integer variables. The first dimension is given by means of restrictions on the set of interpreted integer operators. The second dimension distinguishes between *must*- and *may-constants*; may-constants appear in two variations: single- and multiple-valued. In the third dimension we distinguish between programs with or without loops. We succeed in classifying the complexity of the problems almost completely (Chapter 3). Moreover, we develop (must-)constant propagation algorithms that interpret completely all integer operators except of the division operators by using results from linear algebra and computational ring theory (Chapter 4). These algorithms are far more precise than existing constant-propagation algorithms.

**Limits for the analysis of parallel programs.** We study propagation of copy constants in parallel programs. Assuming that base statements execute atomically, a standard assumption in the program verification and analysis literature, we show that exact copy-constant propagation is undecidable, PSPACE-complete, and NP-complete if we consider programs with procedures, without procedures, and without loops, respectively (Chapter 5). These results indicate that it is very unlikely that recent results on efficient exact analysis of parallel programs can be generalized to richer classes of dataflow problems.

**Abandoning the atomic execution assumption.** We then explore the consequences of abandoning the atomic execution assumption for base statements in parallel programs, which is the more realistic setup in practice (Chapters 6 to 10). Surprisingly, it turns out that this makes exact copy-constant propagation, exact faint-code elimination and, more generally, exact dependence analysis decidable for programs with procedures (Chapter 9) although it remains intractable (NP-hard) (Chapter 10). In order to show decidability we develop a precise abstract interpretation of sets of runs (program executions) (Chapter 8). While the worst-case running time of the developed algorithms is exponential in the number of global variables, it is polynomial in the other parameters describing the program size. As well-designed parallel programs communicate on a small number of global variables only, there is thus the prospect of developing practically relevant algorithms by refining our techniques.

These three contributions constitute self-contained parts of this thesis that can be read independently of each other. Figure 1.2 shows the assignment of the chapters to these three contributions and indicates dependences between the chapters. For clarity transitive relationships are omitted. Before we turn to
the technical presentation we motivate and describe these contributions in more
detail in the next chapter.
Throughout this thesis we assume that the reader is familiar with the basic
techniques and results from the theory of computational complexity [60, 32],
program analysis [58, 2, 26, 51], and abstract interpretation [12, 13]. A brief
introduction to constraint-based program analysis is provided by Appendix A.

**Acknowledgements**

I would like to thank Bernhard Steffen, head of the research group on Programming
Systems at the University of Dortmund, in which I worked from 1996, for
continual advice and support in many ways. I am also grateful to Oliver Rüthing
and Helmut Seidl for our joint work. I thank all three of them and Jens Knoop
for many helpful discussions and Hardi Hungar for his insightful comments on a
draft version.

From October 2001 until March 2002 I worked at Trier University, which
enabled me to elaborate the third part undisturbed from teaching duties. I thank
Helmut Seidl and the DAEDALUS project, that was supported by the European
FP5 programme (RTD project IST-1999-20527), for making this visit possible.

Finally, I would like to thank Bea Bollig for urging me at the right moment
to start working seriously on this thesis and Claudia Herbers for ensuring termina-
tion of this process about a year later.
Chapter 2

Introduction

Constant propagation is one of the most widely used optimizations in practice (cf. [2, 26, 51]). Its goal is to replace expressions that always yield a unique constant value at run-time by this value. This transformation can both speed up execution and reduce code size by replacing a computation or memory access by a load-constant instruction. Often constant propagation enables powerful further program transformations. An example is branch elimination: if the condition guarding a branch of a conditional can be identified as being constantly false, the whole code in this branch is dynamically unreachable and can be removed.

The term constant propagation is somewhat reminiscent of the technique used in early compilers: copying the value of constants in programs (like in $x := 42$) to the places where they are used. The associated analysis problem, to identify expressions in the programs that are constant at run-time, is more adequately called constant detection. However, in the literature the term constant propagation is also used to denote the detection problem. We use the term constant propagation in informal discussions but prefer the term constant detection in more formal contexts.

Constant propagation is an instance of an automatic program analysis. There are fundamental limitations to program analysis deriving from undecidability. In particular, constant detection in full generality is undecidable. Here is a simple reduction for a prototypic imperative programming language. Suppose we are given a program $P$ and assume that new is a variable not appearing in $P$. Consider the little program:

\[
\text{read(new)} ; P ; \text{write(new)}. 
\]

If $P$ does not terminate, new can be replaced by any constant in the write statement for trivial reasons, otherwise this transformation is unsound because the read-statement can read an arbitrary value. Thus, in order to solve the constant detection problem in its most general form, we have to solve the halting problem.

Similar games can be played in every universal programming language and for almost any interesting analysis question. Hence, the best we can hope for is approximate algorithms. An approximate analysis algorithm does not always give a
definite answer. An approximate constant-detection algorithm, for instance, detects some but in general not all constants in a program. The standard approach to constant propagation called *simple constant propagation*, for instance, does not detect that \( z \) is a constant of value 5 at node 7 in the flow graph in Fig. 2.1; cf. Appendix A. It is important that an approximate analysis algorithm only errs on one side and that this is taken into account when the computed information is exploited. This is called the *soundness* of the algorithm. We take soundness for granted in the discussion that follows.

Undecidability of the halting problem implies that it is undecidable whether a given program point can be reached in an execution of the program or not. We have seen above by the example of constant detection that this infects almost every analysis question. It is therefore common to abstract guarded branching to non-deterministic branching in order to ban this fundamental cause of undecidability. This abstraction is built into the use of the MOP-solution (see Appendix A) as the semantic reference point in dataflow analysis. This is: instead of the ‘real’ executions, we take all executions into account that at each branching point choose an arbitrary branch irrespective of the guard. Clearly, this abstraction makes reachability of program points decidable. Most analysis questions encountered in practice (and all the ones we are interested in in this thesis) ask for determining a property valid in all executions of the programs. For such questions information that is determined after guarded branching is abstracted to non-deterministic branching is valid, because more executions are considered. Adopting this abstraction, we work with non-deterministic programs in this thesis. Non-deterministic programs represent deterministic programs in which guarded branching has been abstracted to non-deterministic branching.
A Hierarchy of Integer Constants in Sequential Programs

The abstraction to non-deterministic branching does not solve all the problems with undecidability. Constant detection, for instance, remains undecidable for programs working on integer variables and a full signature of integer operators. Independent proofs of this fact have been given by Hecht [26] and by Reif and Lewis [67]. We briefly recall the construction of Reif and Lewis. It is based on a reduction from Hilbert’s famous tenth problem, whether a multi-variate polynomial has a zero in the natural numbers. This is known to be an undecidable problem [45]. Assume given a polynomial \( p(x_1, \ldots, x_n) \) in \( n \) variables \( x_1, \ldots, x_n \) with natural coefficients different from the zero polynomial and consider the (non-deterministic) program in Figure 2.2. The initialization and the loop choose an arbitrary natural value for the \( x_i \). If the chosen values constitute a zero of \( p(x_1, \ldots, x_n) \), then \( p(x_1, \ldots, x_n)^2 + 1 = 1 \) and \( r \) is set to 1. Otherwise, \( p(x_1, \ldots, x_n)^2 + 1 \geq 2 \) such that \( r \) is set to 0. Therefore, \( r \) is a constant of value 0 at the end of the program if and only if \( p(x_1, \ldots, x_n) \) does not have a natural zero. This result shows us that we cannot even hope for algorithms that detect all constants in non-deterministic programs.

On the other hand there are well-known and well-defined classes of constants that can be detected, even efficiently. A simple example are copy constants [18]. Roughly speaking, a variable \( x \) is a copy constant either if it is assigned a constant value (e.g., through \( x := 42 \)) or if it is assigned the value of another copy constant (e.g., in \( y := 42 ; x := y \)). All other forms of assignments (e.g. \( x := y + 1 \))
are (conservatively) assumed to make \( x \) non-constant \([70]\). Copy constants can efficiently be detected by a standard dataflow analysis; cf. Appendix A. Also if we restrict attention to programs without loops, even general constant detection is clearly decidable because there are only finitely many execution paths reaching any given program point and we can inspect all paths in succession. But even in this setting the problem is intractable; recently it has been shown to be \text{co-NP}-hard \([38]\). Another decidable class of constants are \textit{finite constants} \([75]\).

These results motivate our considerations in Chapter 3 and 4 where we examine the borderline of intractability and undecidability more closely. To this end, we investigate the constant propagation problem for integers with respect to a three-dimensional taxonomy. The first dimension is given by the distinction between arbitrary and loop-free flow graphs.

The second dimension introduces a hierarchy of weakened versions of the constant-propagation problem. In copy-constant propagation only non-composite expressions are interpreted on the right hand side of assignments; all other expressions are assumed to produce non-constant values. We are interested in the question how far we can go in posing less drastic restrictions on the expressions that are interpreted exactly. A natural way of posing a restriction is to fix a sub-signature of integer operators and to interpret just the expressions built from operators of this sub-signature. All but one of the classes studied in Chapter 3 are given in this way. More specifically, we investigate the following natural sub-signatures of the full integer signature and use the following names for the corresponding classes of constants:

1. the empty signature gives rise to \textit{copy constants};
2. the signature \{+, −\} gives rise to \textit{Presburger constants}.
3. the signature \{+, −, \*\} gives rise to \textit{polynomial constants}; and
4. the full integer signature \{+, −, \*, \texttt{div, mod}\} gives rise to \textit{full integer constants}.

The one remaining class is the class of \textit{linear constants} which is added because it has previously been studied in the literature \([70]\). It lies between the classes of copy constants and Presburger constants. In linear-constant detection all expressions of the form \( a \times x + b \), where \( a \) and \( b \) are integers and \( x \) is a program variable, are interpreted in addition to non-composite expressions.

Finally, in the third dimension we vary the general nature of the constant-propagation problem. Besides the standard \textit{must-constancy} problem we consider the less frequently addressed problem of \textit{may-constancy}. Essentially, this problem asks if a variable may evaluate to a given constant \( c \) at a given program point.

\( ^{1} \)The results remain valid if we abandon the \texttt{mod} operator. Note that \texttt{mod} can be expressed by the other operators by means of the identity \( x \texttt{mod} y = x - x \times (x \texttt{div} y) \) for \( x \geq 0, y > 0 \).
in some program execution. Inspired by the work of Muth and Debray [57] we further distinguish between a single-value and a multiple-value variant, where in the latter case the values of multiple variables are checked simultaneously. While the most prominent application of must-constant propagation is compile-time simplification of expressions, both must- and may-variants are equally well suited for eliminating unnecessary branches in programs. Furthermore, the may-variant leads to insight in the complexity of (may-)aliasing of array elements.

Combination of the second and third dimension of the taxonomy gives rise to 15 different classes of constants. We succeed in almost completely characterizing the complexity of detecting these classes of constants in general (non-deterministic) flow graphs as well as in loop-free flow graphs. Only two questions remain open, both concern general flow graphs: (1) we miss an upper bound for linear may-constants and (2) the upper and lower bound for polynomial must-constants do not coincide.

**Constant Propagation via Effective Weakest Preconditions**

There are two motivations for research that classifies the complexity for sub-classes of analysis problems. On the theoretical side, we hope to increase our understanding of the tradeoff between efficiency and precision for analysis problems that can be solved only approximately. On the practical side, we hope to uncover potential for construction of more powerful analysis algorithms. Indeed, perhaps the most interesting results of our study of the constant taxonomy are the following two findings that uncover algorithmic potential (Chapter 4).

The first finding is that the detection of Presburger constants is tractable, i.e. can be done in polynomial time; the second that polynomial constants are decidable. The latter result is particularly interesting because full constants are undecidable as we have seen above. So the division operator is identified as the source of non-decidability. For showing decidability of polynomial constants we apply results from computable ring theory.

Both detection algorithms for Presburger and polynomial constants use an indirect three phase approach. In the first phase a candidate value is computed that is verified in the second and third phase by means of a symbolic weakest-precondition computation. The algorithms are obtained by instantiating a generic algorithm for the construction of approximate constant-propagation algorithms that are complete with respect to evaluation of a subset of expressions. We describe the general algorithmic idea of constant propagation via symbolic weakest-precondition computation and analyze the demands for making this general algorithmic idea effective. Assertions are represented by affine subspaces of $\mathbb{Q}^n$ for Presburger constants and by ideals in the polynomial ring $\mathbb{Z}[x_1, \ldots, x_n]$ for polynomial constants.
CHAPTER 2. INTRODUCTION

Limits for the Analysis of Parallel Programs

While the first part is concerned with analysis of sequential programs, the bulk of this thesis is concerned with analysis of parallel programs. Automatic analysis of parallel programs is known as a notoriously hard problem. A well-known obstacle is the so-called state-explosion problem: the number of (control) states of a parallel program grows exponentially with the number of parallel components. Some results that are rather surprising in view of the state-explosion problem have been the starting point for the considerations in this thesis: certain basic but important dataflow-analysis problems can still be solved completely and efficiently for programs with a fork/join kind of parallelism. Let us briefly report on these results before we describe our contribution.

Knoop, Steffen, and Vollmer [40] show that bitvector analyses, which comprise, e.g., live/dead variables, available expressions, and reaching definitions [51], can efficiently be performed on such programs. Knoop shows in [37] that a simple variant of constant detection, that of so-called strong constants, is tractable as well. These papers restrict attention to the intraprocedural problem, in which each procedure body is analyzed separately with worst case assumption on called procedures. Seidl and Steffen [71] generalize these results to the interprocedural case in which the interplay between procedures is taken into account and to a slightly more extensive class of dataflow problems called gen/kill problems. These papers extend the fixpoint computation technique common in data flow analysis to parallel programs.

Another line of research applies automata-theoretic techniques that originally have been developed for the verification of so-called PA-processes (Process-Algebra Processes) [5, 46, 7, 43], a certain class of infinite-state processes combining sequentiality and parallelism. Specifically, Esparza and Knoop [16], and Esparza and Podelski [17] demonstrate how live variables analysis can be done and indicate that other bitvector analyses can be approached in a similar fashion.

Can these results be generalized further to considerably richer classes of dataflow problems? For answering this question we investigate the complexity of exact copy-constant detection in parallel programs. Intuitively, copy-constant detection which is closely related to static-dependence analysis represents the next level of difficulty of dataflow problems beyond gen/kill problems. In the sequential setting copy-constant detection gives rise to a distributive dataflow framework on a lattice with chain height two and can thus—by the classic result of Kildall [36, 51]—completely and efficiently be solved by a fixpoint computation.

We show in Chapter 5 by means of a reduction from the halting problem for two-counter machines that copy-constant detection is undecidable in parallel programs with procedures (parallel interprocedural analysis). Moreover, we show PSPACE-completeness in case that there are no procedure calls (parallel intraprocedural analysis), and co-NP-completeness if also loops are abandoned.
(parallel acyclic analysis). The latter results rely on reductions from the intersection problem for regular and star-free regular expressions, respectively. These results render the possibility of complete and efficient dataflow algorithms for parallel programs for more extensive classes of analyses unlikely even for loop-free programs, as it is generally believed that the inclusions \( P \subseteq (\text{co-})NP \subseteq \text{PSpace} \) are proper.

Let us be a bit more specific about the setting in which these results are obtained. We consider a prototypic language of explicitly parallel programs. The threads operate on a shared memory via assignment statements of a very restricted form:

\[ x := 0 \text{ and } x := 1 \]

for two distinct constants 0 and 1, and copying assignments \( x := y \). Any sensible concurrent programming language that allows threads to access a shared memory provides such statements and therefore our hardness results are applicable to many scenarios. The language allows to form composed statements by means of sequential composition \( ; \), parallel composition \( || \), and non-deterministic branching \( \Box \). Moreover, there is a loop construct \( \text{loop } \pi \text{ end} \), that executes the loop body \( \pi \) an indefinite number of times. The non-deterministic branching and indefinite loop constructs are chosen in accordance with the abstraction of guarded to non-deterministic branching mentioned above. Parallelism is understood in an interleaving fashion; assignment statements are assumed to execute atomically.

In the intraprocedural setting we consider analysis in statements of the form described above; in the loop-free case we abandon the loop statement. In the interprocedural setting we consider programs consisting of procedures, the body of which consist of statements of the form outlined above. Of course, procedures may also (recursively) call each other. A terminological remark is in order here. Whenever we speak of interprocedural analysis, we implicitly imply that the analysis takes properly into account the call and return structure of procedures, i.e., we always assume that a dynamic instance of a procedure that is entered at a certain call site, returns to that same call sites. In the traditional parlance of the flow-analysis literature one says that only realizable paths are considered and that the analysis is context-sensitive. In the literature also so-called context-insensitive interprocedural analyses are considered. Such analyses do not properly mirror the call and return structure but pessimistically assume that a procedure called at a certain call site may return to any other call site. Clearly, this leads to sound but in general less precise analysis results. In this thesis we always imply that interprocedural problems only involve realizable paths. Thus, we reserve the term interprocedural analysis or problem for context-sensitive interprocedural analysis or problem, respectively.

The results of Chapter 5 should be contrasted with complexity and undecidability results of Taylor [77] and Ramalingam [65] who consider synchronization-

\(^2\)Two other basic statements are added only for presentational convenience and clarity: the do-nothing statement \( \text{skip} \) and write-statements \( \text{write}(e) \).
dependent dataflow analyses of parallel programs, i.e. analyses that are precise with respect to the synchronization structure of programs. Taylor and Ramalingam largely exploit the strength of rendezvous-style synchronization, while we exploit interference only here and no kind of synchronization. Our results thus point to a much more fundamental limitation in dataflow analysis of parallel programs.

In order to perform our reductions without relying on synchronization we use a subtle technique involving re-initialization of variables. In all reductions programs are constructed in such a way that certain well-behaved runs simulate some intended behavior, e.g., the execution sequences of the given two-counter machine in the undecidability proof. But we cannot avoid that the constructed programs have also certain runs that bear no correspondence to the behavior to be simulated. One would use synchronization to exclude such spurious runs but in the absence of synchronization primitives this is not possible. In order to solve this problem, we ensure by well-directed re-initialization of variables that the spurious runs do not contribute to propagation of the information that is to be determined by the analysis. Intuitively, one may interpret this as a kind of “internal synchronization”.

The prototypic framework poses only rather weak requirement such the results apply to many concurrent programming languages. One additional remark concerning the parallel composition operator is in order here. It is inherent in the definition of parallel composition that π₁ || π₂ terminates if and when both threads π₁ and π₂ terminate (like, for instance, in OCCAM [29]). This means that there is an implicit synchronization between π₁ and π₂ at the termination point. However, as explained in Section 5.6, the hardness results remain valid without this assumption. Therefore, they also apply to languages like JAVA in which spawned threads run and terminate independently of the spawning thread.

Abandoning the Atomic Execution Assumption

Another standard assumption turns out to be more critical: atomic execution of assignments. The idealization that assignments execute atomically is quite common in the literature on program verification as well as in the theoretical literature on flow analysis of parallel programs. However, in a multi-processor environment where a number of concurrently executing processes access a shared memory, this is often an unrealistic assumption. The reason is that assignments are broken into smaller instructions before execution. This is explained in more detail in Chapter 7.

Surprisingly, the reductions of Chapter 5 break down when the atomic execution assumption for assignment statements is abandoned. Without this assumption the subtle game of re-initialization of variables that is crucial for putting the reductions to work can no longer be played. This is illustrated by means of an example program in Section 7.2. Of course, this does not imply that the hardness
results are no longer valid: there could be reductions employing other techniques. But we can indeed show, that interprocedural detection of copy constants and fain-code elimination becomes decidable. Specifically, we develop EXPTIME-algorithms for these problems. Recall that these problems are undecidable under the assumption that assignments execute atomically. So, the (unrealistic) idealization from program verification “atomic execution of assignment statements” that presumably simplifies matters actually increases the difficulty of these problems from the program analysis point of view: amazingly, these problems become more tractable if we adopt a less idealized, more realistic view of execution!

The presentation of these results is spread over Chapters 6 to 9 as it is technically somewhat involved. In the following we give a high-level overview and introduction to these chapters.

Our algorithms apply the constraint-based approach to program analysis. Constraint-based program analysis provides a framework to develop analyses and argue about their correctness and completeness. Put in a nutshell, the idea is to set up constraint systems that characterize sets of program executions in parallel programs and to perform the analysis by solving these constraint systems over a lattice of abstract values. Appendix A explains this in more detail.

Constraint-based analysis of parallel programs has been pioneered by Seidl and Steffen [71]. In order to come to grips with parallel composition, new operators on run sets are used that are not needed in systems for sequential programs. The new operators are an interleaving operator \( \odot \) and prefix and postfix operators \( \text{pre} \) and \( \text{post} \). In general, it is not possible to give adequate interpretations of these new operations for arbitrary dataflow frameworks. Seidl and Steffen show, however, that for gen/kill dataflow problems this can be done. Note that the copy-constant framework does not belong to this class.

In Chapter 6 we define parallel flow graphs, furnish them with an operational semantics, and define constraint systems characterizing various sets of runs: same-level and inverse-same-level runs, reaching and terminating runs, and bridging runs. For the moment, we still assume atomic execution of base statements. While same-level and reaching runs are already found in Seidl and Steffen’s exposition, and they indicate that inverse-same-level and terminating runs can be obtained by duality, bridging runs are new. Moreover, in contrast to Seidl and Steffen we relate the constraint systems to the underlying operational semantics instead of postulating them. In our opinion this clarifies what exactly is specified by the constraint systems. It also helped to uncover and correct a subtle error in their treatment of non-reachable program points. While an understanding of the other sets of runs is not needed in the remainder of this introduction, we must explain bridging runs.

In a bridging run we are given two program points \( u \) and \( v \). A bridging run from a program point \( u \) to another program point \( v \) is a sequence of atomic actions that can bring us from a configuration in which control is at program point \( u \) to a configuration in which control is at program point \( v \). Why are
we interested in bridging runs? We call a pair of program variables \((x, y)\) a \textit{dependence} and say that a given run \textit{mediates} the dependence \((x, y)\) if the value of \(y\) after the run depends on the value of \(x\) before the run, where we judge dependences syntactically. If we are able to determine the dependences mediated by bridging runs then we can use this information to indirectly answer certain program analysis questions. In particular, this information suffices to detect copy constants and faint code.

In Chapter 7 we explain why atomic execution is not a realistic assumption on program execution and motivate and define a non-standard interpretation for the operators and constants used in the constraint systems for parallel programs. This non-standard interpretation captures non-atomic execution of base statements. The idea is to break base statements into atomic actions of smaller granularity and to use an interleaving semantics on these atomic actions. By interpreting the constraint systems from Chapter 6 with the new interpretation, we get run sets that capture non-atomic execution of base statements. These run sets are taken as the reference semantics for judging the precision of our algorithms for copy-constant detection and faint-code elimination.

Unfortunately, we cannot obtain the dependences of the interleaving \(R_1 \otimes R_2\) of two (non-atomic) run sets from the dependences of the two run sets \(R_1\) and \(R_2\): we can invent run sets that have the same dependences but behave differently when interleaved with other run sets. Therefore, we need a more informative abstract domain that allows to record more information than just dependences. This domain is the topic of Chapter 8. Here we give a rough description of the ideas underlying this domain. A more extensive explanation and motivation is provided by the introduction to Chapter 8 and the body of that chapter gives a full technical account with many examples.

The basic idea is to collect not just dependences but \textit{dependence sequences}. A dependence sequence of a run is a sequence of dependences that can successively be mediated by the run. For example, the run \(r_1 = \langle c := b, e := d \rangle\) has \((b, c), (d, e)\) as one of its dependence sequences. This dependence sequence plays a dual role: it captures, on the one hand, the potential of \(r_1\) to mediate the dependence \((b, e)\) if its environment can fill the ‘gap’ between \(c\) and \(d\) (e.g., if the environment can perform the run \(r_2 = \langle d := c \rangle\)) and, on the other hand, its potential to successively fill the ‘gaps’ \((b, c)\) and \((d, e)\) in a run of the environment (e.g., in \(r_3 = \langle b := a, d := c, f := e \rangle\)). This idea needs to be refined further in order to allow a proper propagation through all the operators: we must also collect information about transparency of runs. This leads to the notion of \textit{dependence traces}. Moreover, we need to ensure finiteness of the domain in order to ensure that fixpoint computation becomes effective. The latter problem is solved by introducing first, a subsumption order on dependence traces and, secondly, a notion of shortness of dependence traces. We then work with antichains (with respect to the subsumption order) of short dependence traces. In Chapter 8 we show that one can define on this abstract domain operations that are both sound
and precise abstractions of the corresponding operations on non-atomic run sets.

By solving the constraint system for bridging runs over the abstract domain introduced in Chapter 6, we can determine in particular the dependences mediated by bridging runs. As mentioned, this information can be used to detect copy constants and eliminate faint code. Algorithms based on this idea that solve these problems are developed in Chapter 9 and their run-time is analyzed. These algorithms prove that we can detect copy constants and eliminate faint code in parallel programs completely, if we abandon the assumption that base statements execute atomically.

The algorithms run in exponential time, which raises the question whether there are also efficient algorithms for these problems. In Chapter 10 we show by means of a reduction from the well-known SAT-problem that the answer is ‘no’, unless P=NP. Unlike the reductions in Chapter 5, this reduction relies only on active propagation along copying assignments but not on well-directed re-initialization. It applies independently of the atomicity assumption for base statements. In the conclusions, Chapter 11, we sketch possible remedies and discuss directions for future research that may still lead to algorithms of practical interest.

It follows from our reductions that copy-constant detection and faint-code elimination are NP-complete for loop-free programs. We have not yet been able to characterize the complexity for the other classes completely: the general intraprocedural problem and the interprocedural problem. Up to now we have the EXPTIME upper bound provided by the algorithms of Chapter 9 and the NP lower bound of Chapter 10. A natural idea for an NP-easiness proof would be to show that shortest witnessing runs of polynomial length are sufficient. We show in Section 10.2 that this idea does not work: we exhibit a family of programs in which the length of shortest witnessing runs is exponential in the program size. This justifies the conjecture that the general intra- and interprocedural problems do not belong to NP, i.e., cannot be solved by a non-deterministic algorithms that runs in polynomial time.
Chapter 3

A Hierarchy of Constants

Constant propagation (CP) aims at detecting expressions in programs that always yield a unique constant value at run-time. Replacing constant expressions by their value is one of the most widely used optimizations in practice (cf. [2, 26, 51]). Unfortunately, the constant propagation problem is undecidable even if the interpretation of branches is completely ignored, like in the common model of non-deterministic flow graphs where every program path is considered executable. This has been proved independently by Hecht [26] and by Reif and Lewis [67]. We discussed Reif and Lewis’ proof in the introduction. Here we briefly recall Hecht’s proof because we will encounter variants of his construction later in this chapter. It is based on the Post correspondence problem.

A Post correspondence system consists of a set of pairs \((u_1, v_1), \ldots, (u_k, v_k)\) with \(u_i, v_i \in \{0,1\}^*\). The correspondence system has a solution, if and only if there is a sequence \(i_1, \ldots, i_n\) such that \(u_{i_1} \cdot \ldots \cdot u_{i_n} = v_{i_1} \cdot \ldots \cdot v_{i_n}\). Figure 3.1 illustrates Hecht’s reduction. The variables \(x\) and \(y\) are used as decimal numbers representing strings in \(\{0,1\}^*\). For each pair of the correspondence system a distinct branch of the loop appends the strings \(u_i\) and \(v_i\) to \(x\) and \(y\), respectively. This is achieved by shifting the digits of \(x\) and \(y\) by \(|u_i|\) and \(|v_i|\) places first by multiplying them with \(10^{|u_i|}\) and \(10^{|v_i|}\), where \(|u_i|\) and \(|v_i|\) are the length of \(u_i\) and \(v_i\). Afterwards, we add \(u_i\) and \(v_i\) where we identify \(u_i\) and \(v_i\) with the decimal number they represent. It is easy to see that \(x - y\) always evaluates to a value different from 0, if the Post correspondence problem has no solution.\(^2\) In this case the expression \(1 \div ((x - y)^2 + 1)\) always evaluates to 0. But if the Post correspondence system is solvable, the expression \(x - y\) can have the value 0 such that \(1 \div ((x - y)^2 + 1)\) can evaluate to 1. Thus, \(r\) is constant (with value 0), if and only if the Post correspondence problem is not solvable. To exclude \(r\) from being constantly 1 in the case that the Post correspondence system is universally solvable, \(r\) is set to 0 by a bypassing assignment statement.

On the other hand, constant detection is certainly decidable for acyclic, i.e.,

\(^1\)This chapter is based on material from [54]

\(^2\)Note that the initialization of \(x\) and \(y\) with 1 avoids a problem with leading zeros.
loop-free, programs. But even in this setting the problem is intractable; it has been shown to be co-NP-hard [38] recently. This result is based on a polynomial-time reduction of the co-problem of 3-SAT, the satisfiability problem for clauses which are conjunctions consisting of three negated or unnegated Boolean variables (cf. [20, 60]). An instance of 3-SAT is solvable if there is a variable assignment such that every clause is satisfied.

The reduction is illustrated in Figure 3.2 for a 3-SAT instance over the Boolean variables \{b_1, \ldots, b_k\}:

\[
(b_3 \lor \overline{b_5} \lor b_6) \land \ldots \land (b_2 \lor \overline{b_3} \lor b_5).
\]

For each Boolean variable \(b_i\) two integer variables \(x_i\) and \(\overline{x_i}\) are introduced that are initialized by 0. The idea underlying the reduction is the following: each path of the program chooses a witnessing literal in each clause by setting the corresponding variable to 1. If this can be done without setting both \(x_i\) and \(\overline{x_i}\) for some \(i\) then we have found a satisfying truth assignment, and vice versa. On such a path the expression \(x_1 \overline{x_1} + \ldots + x_k \overline{x_k}\) evaluates to 0 and, consequently, both \(r_1\) and \(r_2\) are set to 0. On all other paths the value of \(x_1 \overline{x_1} + \ldots + x_k \overline{x_k}\) differs from 0 but stays in the range \{1, \ldots, k\} which implies that variable \(r_2\) is set to 1. Similarly to the undecidability reduction of Figure 3.1 the bypassing assignment \(r_1 := 1\) avoids that \(r_1\) is constantly 0 in the case that all runs induce satisfying truth assignments. Summarizing, \(r_2\) is a constant (of value 1), i.e., evaluates to 1 on every program path if and only if the underlying instance of 3-SAT has no solution.

Note that both reductions presented so far crucially depend on an operator
like integer division (or modulo) which is capable of projecting many different values onto a single one.

The purpose of this and the following chapter is to examine the borderline of intractability and undecidability more closely. To this end, we investigate the constant detection problem for non-deterministic flow graphs working on integers with respect to a three-dimensional taxonomy. The first dimension is given by the distinction between arbitrary and loop-free flow graphs.

The second dimension introduces a hierarchy of weakened versions of the constant propagation problem. In these variants only assignment statements whose right hand side belong to a given subset $S$ of expressions are interpreted exactly. Assignment statements of other form are conservatively interpreted as non-deterministic assignments. We consider expression sets $S$ that are given by restricting the set of integer operators that are allowed in expression building. We consider signatures without operators (copy constants), with operators restricted to the set $\{+, -\}$ (Presburger constants), operators restricted to $\{+, -, \ast\}$ (polynomial constants), and the standard signature, i.e., the one with operators $+, -, \ast, \text{div, mod}$ (full constants). Moreover, we consider linear expressions in one variable, i.e., expressions of the form $x := ay + b$ because the associated class of constants, linear constants, has previously been studied in the literature [70]. Ob-

Figure 3.2: Co-NP-hardness of CP for acyclic programs: reduction of co-3-SAT.
viously, the class of linear constants lies between copy constants and Presburger constants.

Finally, in the third dimension we vary the general nature of the constant detection problem. Besides the standard must-constancy problem we also consider the less frequently addressed problem of may-constancy here. Essentially, this problem asks if a variable may evaluate to a given constant \( d \) at some given program point. Inspired by work of Muth and Debray [57] we further distinguish between a single value and a multiple value variant, where in the latter case the values of multiple variables are questioned simultaneously. Muth and Debray introduced the single and multiple value variants as models for independent-attribute and relational-attribute dataflow analyses [33].

While the most prominent application of must-CP is the compile-time simplification of expressions, the must- and may-variants are equally well suited for eliminating unnecessary branches in programs. Furthermore, the may-variant has some interesting consequences for the complexity of (may-)aliasing of array elements.

In this chapter we introduce this taxonomy of constants formally, discuss the results that are known or obvious and present a number of new intractability and undecidability results that sharpen previous results. In the next chapter we show decidability of polynomial must-constants and polynomial-time decidability of Presburger must-constants.

### 3.1 A Taxonomy of Constants

#### 3.1.1 Flow Graphs

Let \( X = \{x_1, \ldots, x_n\} \) be a finite set of variables and \( \text{Expr} \) a set of expressions over \( X \); the precise nature of expressions is immaterial at the moment. A (deterministic) assignment is a pair consisting of a variable and an expression written in the form \( x := t \); the set of assignment statements is denoted by \( \text{Asg} \). A non-deterministic assignment statement consists of a variable and is written \( x := ? \); the set of non-deterministic assignment statements is denoted by \( \text{NAsg} \).

A (non-deterministic) flow graph is a structure \( G = (N, E, A, s, e) \) with finite node set \( N \), edge set \( E \subseteq N \times N \), a unique start node \( s \in N \), and a unique end node \( e \in N \). We assume that each program point \( u \in N \) lies on a path from \( s \) to \( e \). The mapping \( A : E \to \text{Asg} \cup \text{NAsg} \cup \{\text{skip}\} \) associates each edge with a deterministic or non-deterministic assignment statement or with the statement \( \text{skip} \). Edges represent the branching structure and the statements of a program, while nodes represent program points. The set of successors of program point \( u \in N \) is denoted by \( \text{Succ}[u] = \{v \mid (u, v) \in E\} \).

A path reaching a given program point \( u \in N \) is a non-empty sequence of edges \( p = \langle e_1, \ldots, e_k \rangle \) with \( e_i = (u_i, v_i) \in E \) such that \( u_1 = s \), \( v_k = u \), and
3.1. A TAXONOMY OF CONSTANTS

$v_i = u_{i+1}$ for $1 \leq i < k$. In addition $p = \varepsilon$, the empty sequence, is a path reaching the start node s. We write $R[u]$ for the set of paths reaching $u$.

Let $\text{Val}$ be a set of values. A mapping $\sigma : X \rightarrow \text{Val}$ that assigns a value to each variable is called a state; we write $\Sigma = \{\sigma \mid \sigma : X \rightarrow \text{Val}\}$ for the set of states. For $x \in X$, $d \in \text{Val}$ and $\sigma \in \Sigma$, we write $\sigma[x \mapsto d]$ for the state that maps $x$ to $d$ and coincides for the other variables with $\sigma$. We assume a fixed interpretation for the operators used in terms and we assume that the value of term $t$ in state $\sigma$, which we denote by $\sigma^t$, is defined in the standard way.

In order to accommodate non-deterministic assignments we interpret statements by relations on $\Sigma$ rather than functions. The relation associated with assignment statement $x := t$ is $[x := t] \overset{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma' = \sigma[x \mapsto t']\}$; the relation associated with non-deterministic assignment $x := ?$ is $[x :=?] \overset{\text{def}}{=} \{(\sigma, \sigma') \mid \exists d \in \text{Val}: \sigma' = \sigma[x \mapsto d]\}$; and the relation associated with skip is the identity: $[\text{skip}] \overset{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma = \sigma'\}$. This local interpretation of statements is straightforwardly extended to paths $p = (e_1, \ldots, e_k) \in E^*$: $[p] = [A(e_1)] ; \ldots ; [A(e_k)]$, where $;\ldots;;$ denotes relational composition. We obtain the set of states $S[u]$, which are possible at a program point $u \in N$ as follows: $S[u] \overset{\text{def}}{=} \{\sigma \mid \exists \sigma_0 \in \Sigma, p \in R[u]: (\sigma_0, \sigma) \in \[p]\}$. The state $\sigma_0$ represents the unknown initial state—the state in which the program is started—which models the input to the program.

3.1.2 May- and Must-Constants

In this section we define when a variable $x$ is a constant at a program point $u$ in a given flow graph. We distinguish between must-constants and the less frequently considered class of may-constants. May-constants come in two variants: as single and multiple may-constants. We provide formal definitions as well as some typical application scenarios. For simplicity, we restrict attention to constancy of variables in our formal framework. In practice also constancy of expressions is of interest. Our definitions can straightforwardly be extended to this more general case and in discussing applications we assume that this has been done. All our results apply also to this more general setting as constancy of expressions is easily reduced to constancy of variables: if we are interested in constancy of an expression $e$ at a program point $u$ we can add an assignment $v := e$ to a new variable $v$ at $u$ and question for constancy of $v$.

**Must-Constants**

A variable $x \in X$ is a **must-constant** at a program point $u \in N$ if

$$\exists d \in \text{Val} \forall \sigma \in S[u]: \sigma(x) = d.$$  

The problem of **must-constant detection** is to determine for a given variable $x$ and program point $u$, whether $x$ is a must-constant, and, if so, what the value of the constant is.
Must-constancy information can be used in various ways. The most important application is the compile-time simplification of expressions. Furthermore, information on must-constancy can be exploited in order to eliminate conditional branches. For instance, if there is a condition \( e \neq d \) situated at an edge leaving node \( n \) and \( e \) is determined a must-constant of value \( d \) at node \( n \), then this branch is unexecutable (cf. Figure 3.3(a)) and may be removed. Since (must-)constant detection and the elimination of unexecutable branches mutually benefit from each other, approaches for conditional constant propagation were developed taking this effect into account [79, 9].

![Figure 3.3: Constancy information used for branch elimination.](image)

**May-Constants**

Complementary to must-constancy, a variable \( x \in X \) is a *may-constant* of value \( d \in \mathbb{Z} \) at a program point \( u \in N \) if

\[
\exists \sigma \in S[u] : \sigma(x) = d.
\]

Note that opposed to the must-constancy definition here the value of the constant is given as an additional input parameter. There is a natural multiple value extension of the notion of may-constancy. Given variables \( x_1, \ldots, x_k \) and values \( d_1, \ldots, d_k \in \mathbb{Z} \) the corresponding multiple value may-constancy problem is defined by:

\[
\exists \sigma \in S[u] : \sigma(x_1) = d_1 \land \ldots \land \sigma(x_k) = d_k.
\]

While may-constancy information cannot be used for expression simplification, it has also some valuable applications. Most obvious is a complementary branch elimination transformation. If an expression \( e \) is not a may-constant of value \( d \) at node \( n \) then any branch that is guarded by the condition \( e = d \) is unexecutable (cf. Figure 3.3(b)).

May-constancy information is also valuable for reasoning about aliasing of array elements. This can be used, for instance, for parallelization of code or for
improving the precision of other analyses by excluding a worst-case treatment of assignments to elements in an array. Figure 3.4 gives such an example in the context of constant propagation. Here the assignment to $x$ can be simplified towards $x := 6$, only if the assignment to $a[i]$ does not influence $a[0]$. This, however, can be guaranteed if $i$ is not a may-constant of value 0 at the corresponding program node.

$$a[0] := 5$$

$$a[i] := \ldots$$

$$x := a[0] + 1$$

Figure 3.4: Using array alias information from may-constant detection in the context of must-constant propagation.

### 3.1.3 Weakened Constant Detection Problems

We can weaken the demands for a constant detection algorithm as follows: we select a certain subset of expressions $S \subseteq \text{Expr}$ that are interpreted precisely and assume conservatively that assignments whose right hand side does not belong to $S$ assign an arbitrary value to their respective target variable. This can be made formal as follows.

For a given flow graph $G = (N, E, A, s, e)$ and subset of expressions $S \subseteq \text{Expr}$, let $G_S = (N, E, A_S, s, e)$ be the flow graph with the same underlying edge graph but with the following weakened edge annotation:

$$A_S(e) = \begin{cases} 
  x := ?, & \text{if } A(e) = (x := t) \text{ and } t \notin S \\
  A(e), & \text{otherwise}.
\end{cases}$$

A variable $x \in X$ is called an $S$-must-constant ($S$-may-constant) at program point $u \in N$ in flow graph $G$ if it is a must-constant (may-constant) at $u$ in the weakened flow graph $G_S$. The detection problem for $S$-must-constants ($S$-may-constants) is the problem of deciding for a given set of variables $X$, flow graph $G$, variable $x$, and program point $u$ whether $x$ is an $S$-must-constant ($S$-may-constant) at $u$ in $G$. Clearly, if $x$ is an $S$-must-constant at $u$ it is also a must-constant at $u$. Similarly, if $x$ is not an $S$-may-constant at $u$ it is not a may-constant at $u$. In both cases the reverse implication does not hold in general. Thus, an analysis
that solves a weakened constant-detection problem yields sound information for must-constancy and non-may-constancy in the original flow graph.

We should emphasize two points about the above framework that make the construction of $S$-constant-detection algorithms more challenging. Firstly, in contrast to the setup in [54], we allow assignment statements, the right hand side of which do not belong to $S$. They are interpreted as non-deterministic assignments. Forbidding them is adequate for studying lower complexity bounds for analysis questions, which is the main concern of [54]. It is less adequate when we are interested in algorithms because in practice we want to detect $S$-constants in the context of other code.

Secondly, a variable can be an $S$-constant although its value statically depends on an expression that is not in $S$. As a simple example consider the flow graph in Fig. 3.5 and assume that the expressions $0$ and $y - y$ belong to $S$ but $e$ does not.

![Figure 3.5: Static dependences and $S$-constancy: variable $x$ is an $S$-constant at program point 3 although it statically depends on the uninterpreted expression $e$.](image)

Because $y - y$ equals 0 for any value $y \in \mathbb{Z}$, an $S$-must-constant detection algorithm must identify $x$ as a must-constant (of value 0) at program point 3, although the value of $x$ at program point 3 statically depends on the uninterpreted expression $e$. Besides cancellation through subtraction such effects arise through multiplication with terms evaluating to zero. Hence, $S$-constant detection algorithms must handle arithmetic properties of the expressions in $S$. Of course, in real programs cancellation through arithmetic properties may not be as obvious as in this example.

There are at least two other natural definitions for a notion of $S$-constant propagation:

1. We can study constant propagation in flow graphs whose edge annotation is restricted to assignments from $S \cup \{\text{skip}\}$; this is the setup in [54].

2. We can treat the effect of assignments whose right hand side does not belong to $S$ more pessimistically: if the value of $x$ at $u$ statically depends on an uninterpreted assignment, we may define that
3.1. A TAXONOMY OF CONSTANTS

- $x$ is not a must-constant at $u$ and that
- $x$ is a may-constant at $u$ for any value $d$.

This definition also leads to a conservative approximation of must- and non-may constancy, but is weaker than our definition as demonstrated by the above example.

From all the potential definitions our definition requires most from an $S$-constant-propagation algorithm. Firstly, it must handle more inputs than with Definition 1. Secondly, an $S$-constant-propagation algorithm in the sense of 2 can easily be obtained from an algorithm in our sense. We only need to combine it in a straightforward way with a static dependence analysis. The latter can be performed by a cheap bitvector analysis [26, 51]. On the other hand, Definition 1 poses in principle the strongest requirements for hardness considerations. Fortunately, all our reductions use only statements from $S \cup \{\texttt{skip}\}$. Therefore, all our results apply to all three definitions.

3.1.4 Classes of Integer Constants

To study weakened versions of constant-detection problems is particularly interesting for programs computing on the integers, i.e., if $\texttt{Expr}$ is the set of integer expressions formed from integer constants and variables with the standard operators $+, -, \times, \div, \mod$: we have seen above that the general constant-detection problem is undecidable in this case.

We introduce now weakened classes of integer constants. Except for linear constants these classes are induced by considering only a fragment of the standard signature. While the first two classes are well-known in the field of (must-) constant propagation and the class of Presburger constants is closely related to the class of constants considered in [35], we are not aware of any work devoted to the fragment of polynomial constants.

Copy Constants. $S$-constants with respect to the set $S = X \cup \mathbb{Z}$, i.e., the set of non-composite expressions, are known as copy constants [18]. This is due to the fact that constants can only be produced by assignments $x := c$ and be propagated by assignments of the form $x := y$.

Linear Constants. $S$-constants with respect to the set $S = \{a \times x + b \mid a, b \in \mathbb{Z}, x \in X\} \cup X \cup \mathbb{Z}$ are known as linear constants [70].

Presburger Constants. A Presburger constant is an $S$-constant for the set $S$ of integer expressions that can be built from the operators $+$ and $-$. We decided for this term because in Presburger arithmetics integer operations are also restricted to addition and subtraction. Note, however, that the complexity issues
in deciding Presburger formulas and Presburger constants are of a completely different nature, since in the context of constant detection the problem is mainly induced by paths in flow graphs and not by a given logical formula. We call $S$-constants with respect to the set $S = \{c_0 + \sum_{i=1}^{k} c_i \cdot x_i \mid c_j \in \mathbb{Z}, x_i \in X\}$ affine constants. As far as expressiveness is concerned Presburger expressions and affine expressions coincide because multiplication with constants can be simulated by iterated addition. Affine expressions can, however, be more succinct. Nevertheless, all our results on Presburger constants equally apply to affine constants and from now on we do not distinguish these two classes of constants.

**Polynomial Constants.** If all expressions built from the operators $+, -, \ast$ are interpreted, the resulting constants are called polynomial constants as this signature allows just to write multi-variate polynomials. Formally, polynomial constants are $S$-constants with respect to the set $S = \mathbb{Z}[x_1, \ldots, x_n]$, the set of multi-variate polynomials in the variables $x_1, \ldots, x_n$ with coefficients in $\mathbb{Z}$.

### 3.2 Known Results

Table 3.1 summarizes the complexity results that are known or obvious. Problems that have a polynomial-time algorithm are emphasized in a light shade of grey, those that are decidable though intractable in a dark shade of grey, and the undecidable fields are filled black. White fields represent problems where the complexity and decidability is unknown or at least, to the best of our knowledge, undocumented. In the following we briefly comment on these results.

For an unrestricted signature we already presented Hecht’s undecidability proof for must-constants and the co-NP-hardness result for the acyclic counterpart. It is also well-known that the must-constant detection problem is distributive [26], if all right-hand side expressions are either constant or represent a one-to-one function in $\mathbb{Z} \rightarrow \mathbb{Z}$ depending on a single variable (see the remark on page 206 in [72] for a similar observation). Hence the class of linear constants defines a distributive dataflow problem, which guarantees that the standard maximum fixed-point iteration strategy over $\mathbb{Z} \cup \{-, +\}$ computes the exact solution in polynomial time.\(^3\)

The may-constancy problem for copy constants has recently been examined by Muth and Debray [57]. It is easy to see that the single value case can be dealt with in polynomial-time: the number of constant values that a variable may possess at a program point (via copy assignments) is bound to the number of assignments to constants in the program. Hence one can essentially keep track

\(^3\)Sagiv, Reps and Horwitz [70] give an alternative procedure for detecting linear constants by solving a graph reachability problem on the exploded supergraph of a program. They additionally show that with this method linear constant detection can be solved precisely even for interprocedural control flow.
3.2. KNOWN RESULTS

<table>
<thead>
<tr>
<th></th>
<th>Must-Constants</th>
<th>May-Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>single value</td>
<td>multiple value</td>
</tr>
<tr>
<td>acyclic control flow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy Constants</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>Linear Constants</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>Presburger Constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+,-,* Constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full Constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unrestricted control flow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy Constants</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>Linear Constants</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>Presburger Constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+,-,* Constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full Constants</td>
<td>undecidable</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Complexity classification of a taxonomy of CP: the known results.

of any possible constant value at a program point by collecting the set of possible values of variables. Formally, this can be achieved by computing the union-over-all-path solution in a union-distributive dataflow framework over the lattice \( \{ \sigma \mid \sigma : Var \rightarrow 2^{Z_G} \} \), where \( Z_G \) denotes the set of constant right-hand sides in the flow graph \( G \) under consideration and the order on functions is the pointwise lift of subset inclusion on \( 2^{Z_G} \).

The multiple value problem has been shown NP-complete in the acyclic case and PSPACE-complete in the presence of unrestricted control flow by Muth and Debray [57]. For proving NP-hardness and PSPACE-hardness they use reductions from 3-SAT and the acceptance problem for polynomial-space-bounded Turing machines, respectively. It is worth mentioning that the number of variables questioned simultaneously for constancy in these reductions is not bounded by a fixed constant. Finally, since Muth and Debray do not consider any kind of arithmetics, all other fields in the may-constancy column remain open.

In the following we aim at successively filling the white parts in Table 3.1. To this end, we start with providing new undecidability results and then prove a number of new intractability results. Positive results for the classes of Presburger and polynomial must-constants are presented in Chapter 4.
3.3 New Undecidability Results

Hecht's construction sketched in Fig. 3.1 can easily be adapted for proving undecidability of Presburger may-constants. The only modification is to replace the two assignment to $r$ in Figure 3.1 by a single assignment $r := x - y$. As argued before, $x$ may equal $y$ immediately after leaving the loop, if and only if the instance of the Post correspondence problem has a solution. Hence, in this case $x - y$ may evaluate to 0.

**Theorem 3.1** Deciding single valued may-constancy at a program point is undecidable for the class of Presburger constants.

This construction can be further modified to obtain a stronger undecidability result for the class of multiple value may-constants. Here we have:

**Theorem 3.2** Deciding multiple valued may-constancy at a program point is undecidable for the class of linear constants. This even holds if only two values are questioned.

The idea is to substitute the difference $x - y$ in the assignment to $r$ by a loop which simultaneously decrements $x$ and $y$. It is easy to see that $x = 0 \land y = 0$ may hold at the end of the resulting program if and only if $x$ may equal $y$ at the end of the main loop.

Complexity of Array Aliasing.

The previous two undecidability results have immediate implications for array aliasing, which complements similar results known in the field of pointer induced aliasing [42]. As a consequence of Theorem 3.1 we have:

**Corollary 3.3** Deciding whether $A[i]$ may alias $A[c]$ for a one-dimensional array $A$, integer variable $i$ and integer constant $c$ is undecidable, even if $i$ is computed only using the operators $+$ and $-$.

Theorem 3.2 provides a negative result for array accesses with linear index calculations.

**Corollary 3.4** Let $c_1, c_2$ be integer constants and $i, j$ integer variables. Determining whether $A[i, j]$ may alias $A[c_1, c_2]$ for a two-dimensional array $A$ is an undecidable problem even if $i, j$ are computed only with linear assignments of the form $x := a y + b$. 
Clearly, \( x \) may equal \( y \) at the end of the loop in Hecht’s construction if and only if the given Post correspondence system has a solution. Thus, the problem to decide whether an array access \( A[x] \) may alias another access \( A[y] \) just after the loop is also undecidable. This gives us the following result for one-dimensional arrays.

**Theorem 3.5** Let \( i, j \) be integer variables. Determining whether \( A[i] \) may alias \( A[j] \) for a one-dimensional array \( A \) is an undecidable problem even if \( i, j \) are computed only with linear assignments of the form \( x := ay + b \).

It should be noted that traditional work on array dependences like the omega test [62, 64] is restricted to scenarios where array elements are addressed by affine functions depending on some index variables of possibly nested for-loops. In this setting the aliasing problem can be stated as an integer linear programming problem which can be solved effectively. In contrast, our results address the more fundamental issue of aliasing in the presence of arbitrary loops.

### 3.4 New Intractability Results

After having marked off the range of undecidability we prove in this section some intractability results.

We start by strengthening the result on the co-NP-hardness of must-constant detection for acyclic control flow. Here the construction of Figure 3.2 can be modified such that the usage of integer division is no longer necessary. Basically, the trick is to use multiplication by \( 0 \) as the projective operation, i.e. as the operation with the power to map many different values onto a single one. In the construction of Figure 3.2 this requires the following modifications (cf. Fig. 3.6).

All variables are now initialized by \( 1 \) and the part reflecting the clauses sets the corresponding variables to \( 0 \). Finally, the assignments to \( r_1 \) and \( r_2 \) are substituted by a single assignment \( r := (x_1 + \overline{x}_1) \ldots (x_k + \overline{x}_k) \) that is bypassed by another assignment \( r := 0 \). It is easy to see that the instance of 3-SAT has no solution if and only if on every path both \( x_i \) and \( \overline{x}_i \) are set to \( 0 \) for some \( i \in \{1, \ldots, k\} \). This, however, guarantees that at least one factor of the right-hand side expression defining \( r \) is \( 0 \) which then ensures that \( r \) is a must-constant of value \( 0 \). Finally, the branch performing the assignment \( r := 0 \) assures that \( r \) cannot be a must-constant of any other value. Thus, we have:

**Theorem 3.6** Deciding polynomial must-constants in acyclic programs is co-NP-hard.

On the other hand, it is not hard to see that the problem of must-constant propagation is in co-NP for acyclic control flow. To this end, one has to prove that the co-problem, i.e., checking non-constancy at a program point, is in NP,
which is easy to see: a non-deterministic Turing machine can guess two paths
through the program witnessing two different values. Since each path is of linear
length in the program size and the integer operations can be performed in linear
time with respect to the sum of the lengths of the decimal representation of their
inputs, this can be done in polynomial time. Hence we have:

**Theorem 3.7** Must-constant propagation is in co-NP when restricted to acyclic
control flow.

Next we are going to show that the problem addressed by Theorem 3.6 gets
presumably harder without the restriction to acyclic control flow.

**Theorem 3.8** Detecting polynomial must-constants in arbitrary flow graphs is
PSPACE-hard.

Theorem 3.8 is proved by means of a polynomial time reduction from the
language-universality problem of non-deterministic finite automata (NDFA) (cf.
remark to Problem A11 in [20]). This is the question whether an NDFA $\mathcal{A}$ over
an alphabet $X$ accepts the universal language, i.e., whether $L(\mathcal{A}) = X^*$. Without
loss of generality, let us consider an NDFA $\mathcal{A} = (X, S, \delta, s_1, F)$, where $X = \{0,1\}$
is the underlying alphabet, $S = \{1, \ldots, k\}$ the set of states, $\delta \subseteq S \times X \times S$ the
transition relation, $s_1$ the start state, and $F \subseteq S$ the set of accepting states.
The polynomial time reduction to a constant propagation problem is depicted in Figure 3.7.

For every state \( i \in \{1, \ldots, k\} \) a variable \( s_i \) is introduced. The idea of the construction is to guess an arbitrary input word letter by letter. While this is done, it is ensured by appropriate assignments that each variable \( s_i \) holds 0 if and only if the automaton can be in state \( i \) after reading the word guessed so far. This implies that \( \prod_{i \in F} s_i \) is 0 for all words if and only if \( A \) accepts the universal language.

Initially, only the start state variable \( s_1 \) is set to 0 as 1 is the only state which is reachable under the empty word. The central part of the program is a loop which guesses a next alphabet symbol. If we decide, for instance, for 0, then, for each \( i \), an auxiliary state variable \( t_i \) is set to 0 by the assignment \( t_i := \prod_{(j,0,i) \in \delta} s_j \), if and only if one of its 0-predecessors is recognized reachable by the word guessed so far. After all the variables \( t_i \) have been set in this way their values are copied to the variables \( s_i \). When the loop is exited which can happen after an arbitrary word has been guessed, it is checked whether the guessed word is accepted. Like before, the direct assignment \( f := 0 \) has the purpose to ensure that constant values different from 0 are impossible. Therefore, \( f \) is a must-constant (of value 0) at the end of the program, if and only if the underlying automaton accepts the universal language \( \{0, 1\}^* \).

The final reduction in this section addresses the complexity of linear may-constants. Here we have:

\[^4\text{The auxiliary state variables } t_i \text{ are introduced in order to avoid overwriting state variables which are still used in consecutive assignments.}\]
Theorem 3.9 Deciding linear may-constants is NP-hard.

Again we employ a polynomial time reduction from 3-SAT which however differs from the ones seen before. The main idea here is to code a set of satisfied clauses by a number interpreted as a bit-string. For example, in an instance with three clauses the number 100 would indicate that clause two is satisfied, while clauses zero and one are not. To avoid problems with carry-over effects, we employ a \((k + 1)\)-adic number representation where \(k\) is the number of variables in the 3-SAT instance. With this coding we can use linear assignments to set the single "bits" corresponding to satisfied clauses.

![Diagram](image.png)

Figure 3.8: NP-hardness of linear may-CP.

To illustrate our reduction let us assume an instance of 3-SAT with Boolean variables \(\{b_1, \ldots, b_k\}\) and clauses \(c_0, \ldots, c_{n-1}\), where the literal \(b_1\) is contained in \(c_3\) and \(c_5\), and the negated literal \(-b_1\) is contained in \(c_2\) only. Then this is coded in a program as depicted in Figure 3.8. We have a non-deterministic choice part for each Boolean variable \(b_i\). The left branch sets the bits for the clauses that contain \(b_i\) and the right branch those for the clauses that contain \(\overline{b_i}\). Every assignment can be bypassed by an empty edge in case that the clause is also made true by another literal. It is now easy to see that \(r\) is a may-constant of value \(1 \ldots 1\) \((n \text{ times})\) if and only if the underlying instance of 3-SAT is satisfiable.

On the other hand, it is easy to see that detecting may-constancy is in NP for acyclic control flow, since a non-deterministic Turing machine can guess a witnessing path for a given constant in polynomial time. Thus, we have:
### Table 3.2: Complexity classification of a taxonomy of CP: preliminary summary.

<table>
<thead>
<tr>
<th>Control Flow Type</th>
<th>Must-Constants</th>
<th>May-Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>single value</td>
<td>multiple value</td>
</tr>
<tr>
<td>Acyclic</td>
<td>Co-NP compl.</td>
<td></td>
</tr>
<tr>
<td>unrestricted</td>
<td>PSPACE-compl.</td>
<td>NP-complete</td>
</tr>
</tbody>
</table>

**Theorem 3.10** May-constant propagation is in NP when restricted to acyclic control flow.

### 3.5 Summary

The decidability and complexity results of this chapter are summarized in Table 3.2. Note that hardness results propagate from a class of constants to more comprehensive classes of constants, i.e., downwards in the table, and vice versa for easiness results. Moreover, hardness results for acyclic control flow propagate to unrestricted control flow which explains the NP-hardness entry for linear constants and unrestricted control flow.

The table shows that we have already gone a good deal on the way towards classifying the complexity of the problems in our taxonomy of constant propagation. In the next chapters we complement the negative results of this chapter by positive results. Specifically, we attack Presburger and polynomial must-constant propagation.
Chapter 4

Deciding Constants by Effective Weakest Preconditions

One goal of classifying the complexity of weakened versions of program-analysis problems is to uncover potential for more precise analysis algorithms. As witnessed by the white space in Table 3.2, three questions remained open in the complexity classification of the previous chapter: there is no result for Presburger must-constants and there are no upper bounds for polynomial must-constants and Presburger may-constants. In this chapter we provide answers for two of these questions that uncover algorithmic potential. We show that Presburger must-constants can be detected in polynomial time and that polynomial must-constants are decidable by developing corresponding algorithms. These classes are interesting from a practical point of view because the operators $+,-,\times$ are very frequently used, e.g., for computing memory addresses of array components. As we consider must-constant propagation throughout this chapter, we omit the qualifying prefix `must' in the following.

The two algorithms share the same basic algorithmic idea. The main ingredient is effective computation of the weakest precondition of a certain assertion. In this computation, assertions are represented by appropriate mathematical structures. In order to emphasize similarity of the algorithms and to enable application to other scenarios, we develop a generic framework for development of $S$-constant detection algorithms in Section 4.3. Afterwards, we show how to apply it to detection of Presburger and polynomial constants. In the algorithm for Presburger constants, which is discussed in Section 4.4, assertions are represented by affine subspaces of $\mathbb{Q}^n$, where $n$ is the number of variables in the underlying flow graph and well-known results from linear algebra are exploited. In the algorithm for polynomial constants presented in Section 4.7, assertions are represented by the set of zeros of ideals of $\mathbb{Z}[x_1, \ldots, x_n]$, the ring of multi-variate polynomials in the variables $x_1, \ldots, x_n$ with coefficients in $\mathbb{Z}$. Here we rely on results from com-

---

1 This chapter is based on material from [56] and [54]

35
putable ring theory in order to compute with ideals. We recall these less known results in Section 4.5 and describe some additional observations in Section 4.6.

In order to allow the reader to develop some intuition for the algorithms before following the technical generic description in Section 4.3, we provide a more illustrative and informal description of the Presburger constant-propagation algorithm beforehand (Section 4.2). Before that we illustrate the power of the algorithms by discussing some examples of Presburger and polynomial constants (Section 4.1).

4.1 Presburger and Polynomial Constants

Presburger constants are already beyond the scope of standard algorithms. Consider, for instance, the two example flow graphs in Figure 4.1.

Figure 4.1: Presburger constants beyond the scope of standard algorithms.

Part (a) extends the classic example that the standard CP algorithm, so-called simple constant propagation, is non-distributive (cf. [26]). In this flow graph, $z$ is a constant of value 14 at the end of the program. However, none of its operands is constant, although both are defined outside of any conditional branch. Simple constant propagation works by a forward propagation of variable assignments of the form $\delta : X \to \text{Val} \cup \{\bot, \top\}$ where $X$ is the set of program variables and $\text{Val}$ is the value domain. It takes the meet of variable assignments at join points. Already at the join point of the first diamond this algorithm computes a variable assignment with $\delta(a) = \delta(b) = \bot$ because the variables are assigned different values in the two branches and there is no way to recover from this loss of precision.

Part (b) shows a small loop that simultaneously decrements $x$ and increments $y$. Obviously, $z$ is a (Presburger) constant of value 0 at the end of the program.
4.2. PRESBURGER-CONSTANT DETECTION

However, this example is also outside the scope of any standard algorithm and even outside the scope of Knoop and Steffen’s EXPTIME algorithm for finite constants [75] because no finite unfolding of the loop suffices to identify \( z \) as a constant. We should mention that Karr’s algorithm [35], which is briefly discussed in the conclusions of this chapter, is able to identify \( z \) as a constant.

![Diagram](image)

Figure 4.2: A polynomial constant not detected by standard algorithms.

In Fig. 4.2, variable \( x \) is a (polynomial) constant of value 0 at node 4. For similar reasons as above, no standard algorithms can handle this example. Because constancy depends on the multiplications in the terms \( xy - 6 \) and \( x^2 + xy \) neither our Presburger constant-detection algorithm nor Karr’s algorithm can handle this example, in contrast to our algorithm for polynomial constants.

4.2 Presburger-Constant Detection at a Glance

For Presburger-Constant Detection we employ techniques known from linear algebra. We use a backward analysis propagating sets of linear equations describing affine vector spaces (over \( \mathbb{Q} \)).

*The Dataflow Framework.* Given a set of program variables \( X = \{x_1, \ldots, x_n\} \) a linear equation is of the following form: \( \sum_i a_i x_i = b \), where \( a_i, b \in \mathbb{Q}, i = 1, \ldots, n \). Since at most \( n \) of these linear equations are linearly independent, an affine vector space can always be described by means of a linear equation system \( Ax = b \) where \( A \) is a \( k \times n \)-matrix over \( \mathbb{Q} \), \( 1 \leq k \leq n \), and \( b \in \mathbb{Q}^k \). The affine vector sub-spaces of \( \mathbb{Q}^n \) are partially ordered by set inclusion. This results in a (complete) lattice where the length of chains is bounded by \( n \) as any affine space strictly contained in another affine space has a smaller dimension.
The Meet Operation. The meet of two affine vector spaces represented by the equations \( A_1 x = b_1 \) and \( A_2 x = b_2 \) can be computed by normalizing the equation

\[
\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}
\]

which can be done efficiently using Gauss-elimination [60].

Local Transfer Functions. The local transfer functions of affine assignments perform a backward substitution on the linear equations. For instance, the equation \( 3x + y = 10 \) is backward-substituted along an assignment \( x := 2u - 3v + 5 \) towards \( 3 (2u - 3v + 5) + y = 10 \) which then can be “normalized” towards \( y + 6u - 9v = 5 \). Clearly, this can be done in polynomial time. After this normalization, the resulting equation system is again simplified using Gauss-elimination.

A linear equation that depends on \( x \) like \( 3x + y = 10 \) cannot be generally valid after a non-deterministic assignment \( x := ? \). Such equations are thus transformed along \( x := ? \) to unsatisfiable equations like \( 0x = 1 \). Equations that are independent of \( x \) are propagated unchanged. Non-affine assignments are treated in the same way.

The Overall Procedure. Our backward dataflow analysis can be regarded as a demand-driven analysis which works separately for each variable \( x \) and program point \( u \). Conceptually, it is organized in three phases:

**Phase 1:** Guess an arbitrary cycle-free path from the startnode to \( u \), for instance using depth-first search, and compute the value \( c \) of \( x \) on this path.

**Phase 2:** Solve the backward dataflow analysis where initially the program point \( u \) is annotated by the affine vector space described by the linear equation \( x = c \) and all other program points by the universal affine space, i.e., the one given by \( \sum_i 0x_i = 0 \).

**Phase 3:** The guess generated in phase 1 is proved, if and only if the start node is still associated with the universal affine vector space.\(^2\)

The completeness of the algorithm is a consequence of the distributivity of the analysis. Obviously, the guessed equation \( x = c \) is true iff the backward substitution along every path originating at the start node yields a universally valid constraint at the start node. Since this defines the meet-over-all-paths solution of our dataflow framework the algorithmic solution is guaranteed to coincide if the transfer functions are distributive, which is immediate from the definition.

\(^2\)In practice, one may already terminate with the result of non-constancy of \( x \) whenever a linear equation system encountered during the analysis is unsolvable.
4.2. PRESBURGER-CONSTANT DETECTION

The algorithm can also be understood from a program verification point of view. By Phase 1, \( c \) is the only candidate value for \( x \) being constant at \( u \). Phase 2 effectively computes the weakest (liberal) precondition of the assertion \( x = c \) at program point \( n \). Clearly, \( x \) is a constant at \( u \) if and only if the weakest liberal precondition of \( x = c \) is universally valid. This point of view is elaborated in the remainder of this chapter.

As mentioned, the length of chains in the analysis is bounded by the number of variables \( n \). Any change at a node can trigger a re-evaluation at its predecessor nodes. Therefore, we have at most \( O(en) \) Gauss-elimination steps, where \( e \) denotes the number of edges in the flow graph. Each Gauss-elimination step is of order \( O(n^3) \) [60]. Thus, the complexity for the complete dataflow analysis for a single occurrence of a program variable is \( O(en^4) \). For an exhaustive analysis that computes constancy information for any left-hand side occurrence of a variable the estimation becomes \( O(pe n^4) \), where \( p \) denotes the number of program points in the flow graph. Summarizing, we have:

**Theorem 4.1** Presburger (must-)constants can be detected in polynomial time.

We now illustrate our algorithm by means of the example of Figure 4.1.

![Figure 4.1](image-url)

The emphasized annotation of Figure 4.3 contains the linear equations resulting from the initial guess \( z = 14 \) (in Figure 4.3(a)) and \( z = 0 \) (in Figure 4.3(b)), respectively. It should be noted that for the sake of presentation we did not display the equations for every program point. The particular power of this technique results from the normalization performed on the linear equations which
provides a handle to cope with arithmetic properties like commutativity and associativity to a certain extent. For instance, the equation $a + b = 5$ in Figure 4.3(a) is the uniform result of two different intermediate equations.

Let us briefly discuss the modifications for polynomial constant propagation. We use more expressive equations of the form $p(x_1, \ldots, x_n) = 0$, where $p(x_1, \ldots, x_n)$ is a multi-variate polynomial with coefficients in $\mathbb{Z}$. A collection of such equations represents the set of zeros of an ideal in $\mathbb{Z}[x_1, \ldots, x_n]$. Exploiting results from computable ring theory we can effectively compute with such equations. In particular, we use Gröbner bases as canonic representation of ideals and the Buchberger algorithm for simplification. While polynomial assignments are handled analogously to affine assignments by backward substitution, the treatment of non-deterministic assignments needs a refinement. The reason is that an equations that depends on $x$ can still be generally valid for all values of $x$ for certain values of other variables. The equation $x y = 0$, for instance, is valid after $x :=?$ if $y = 0$ is valid before.

After this informal presentation of the algorithms we are now ready for the more formal generic description.

4.3 A Generic Algorithm

We assume the formal framework of Section 3.1. Suppose we are given a variable $x \in X$ and a program point $w \in N$. In this chapter we describe a generic algorithm for deciding whether $x$ is an $S$-constant at $w$ or not. While standard constant propagation works by forward propagation of variable assignments, we use a three phase algorithm that employs a backwards propagation of assertions, as we have seen in Section 4.2. For the moment we can think of assertions as predicates on states as in program verification.

**Phase 1:** In the first phase we follow an arbitrary cycle-free path from $s$ to $w$, for instance using depth-first search, and compute the value $c$, referred to as the *candidate value*, that $x$ holds after this path is executed. This implies that, if $x$ is a constant at $w$, it must be a constant of value $c$.

**Phase 2:** In the second phase we compute the weakest precondition for the assertion $x = c$ at program point $w$ in $G_s$ by means of a backwards dataflow analysis.

**Phase 3:** Finally, we check whether the computed weakest precondition for $x = c$ at $w$ is *true*, i.e., is valid for all states.

It is obvious that this algorithm is correct. The problem is that Phase 2 and 3 are in general not effective. However, as only assignments of a restricted form appear in $G_s$, the algorithm becomes effective for certain sets $S$, if assertions


are represented appropriately. In the remainder of this section we analyze the
requirements for adequate representations. For this purpose, we first characterize
weakest preconditions in flow graphs.

Semantically, an assertion is a subset of states \( \phi \subseteq \Sigma \). Given an assertion \( \phi \)
and a statement \( s \), the weakest precondition of \( s \) for \( \phi \), \( \text{wp}(s)(\phi) \), is the largest
assertion \( \phi' \) such that execution of \( s \) from all states in \( \phi' \) is guaranteed to terminate
only in states in \( \phi \).\(^3\) The following identities for the weakest precondition of
assignment and skip statements are well-known:

\[
\begin{align*}
\text{wp}(x := e)(\phi) & \overset{\text{def}}{=} \phi[e/x] \overset{\text{def}}{=} \{ \sigma \mid \sigma[x \mapsto e^\sigma] \in \phi \} \\
\text{wp}(x := ?)(\phi) & \overset{\text{def}}{=} \forall x(\phi) \overset{\text{def}}{=} \{ \sigma \mid \forall d \in \mathbb{Z} : \sigma[x \mapsto d] \in \phi \} \\
\text{wp}(\text{skip})(\phi) & \overset{\text{def}}{=} \phi
\end{align*}
\]

These identities characterize weakest preconditions of basic statements. Let us
now consider the following more general situation in a given flow graph \( G = (N, E, A, s, e) \): we are given an assertion \( \phi \subseteq \Sigma \) as well as a program point
\( w \in N \) and we are interested in the weakest precondition that guarantees validity
of \( \phi \) whenever execution reaches \( w \). The latter can be characterized as follows.

Let \( W_0[w] = \phi \) and \( W_0[u] = \Sigma \) and consider the following equation system
consisting of one equation for each program point \( u \in N \):

\[
W[u] = W_0[u] \cap \bigcap_{v \in Succ[u]} \text{wp}(A(u, v))(W[v]). \quad (4.1)
\]

By the Knaster-Tarski fixpoint theorem, this equation system has a largest solu-
tion (w.r.t. subset inclusion) because \( \text{wp}(s) \) is well-known to be monotonic. By
abuse of notation, we denote the largest solution by the same letter \( W[u] \). For
each program point \( u \in N \), \( W[u] \) is the weakest assertion such that execution
starting from \( u \) with any state in \( W[u] \) guarantees that \( \phi \) holds whenever execution
reaches \( w \). In particular, \( W[s] \) is the weakest precondition for validity of \( \phi \)
at \( w \). The intuition underlying equation (4.1) is the following: firstly, \( W_0[u] \) must
be implied by \( W[u] \) and, secondly, for all successors \( v \), we must guarantee that
their associated condition \( W[u] \) is valid after execution of the statement \( A(u, v) \)
associated with the edge \((u, v)\); hence \( \text{wp}(A(u, v))(W[v]) \) must be valid at \( u \) too.

For two reasons, the above equation system cannot be solved directly in gen-
eral: firstly, because assertions may be infinite sets of states they cannot be repre-
sented explicitly; secondly, there are infinitely long descending chains of assertions
such that we cannot guarantee that standard fixpoint iteration terminates.

In order to construct an algorithm that detects \( S \)-constants we represent assertions
by the members of a lattice \((\mathbb{D}, \sqsubseteq)\). For Presburger constants \( \mathbb{D} \) is the set of

\(^3\)In the sense of Dijkstra [15] this is the weakest liberal precondition as it does not guarantee termination. For simplicity we omit the qualifying prefix “liberal” in this chapter.
affine spaces of $\mathbb{Q}^n$ and for polynomial constants the set of ideals in $\mathbb{Z}[x_1, \ldots, x_n]$. We then simulate the iterative fixpoint computation for $\mathbf{W}$ on the members of lattice $\mathbb{D}$. In order to ensure termination, we require that $\mathbb{D}$ has no infinite ascending chains. In order to ensure that the computed result represents $\mathbf{W}$ precisely, we make sure (1) that the start value $W_0$ is represented precisely and (2) that the operations on $\mathbb{D}$ mirror the operations on assertions precisely. These requirements are detailed below. Note that it is a non-trivial fact that we can find such a lattice $\mathbb{D}$ for a certain set $S$ of expressions: if, for instance, $S$ is the set of all integer expressions, such a lattice cannot exist, because this would imply decidability of must-constancy.

Let us assume that $\gamma : \mathbb{D} \rightarrow 2^2$ captures how the lattice element represent assertions. First of all, we require

(a) $\mathbb{D}$ has no infinite decreasing chains, i.e., there is no infinite chain $d_1 \sqsupset d_2 \sqsupset d_3 \sqsupset \ldots$

This guarantees that maximal fixpoints of monotonic functions can effectively be computed by standard fixpoint iteration. Secondly, we suppose

(b) $\gamma$ is universally conjunctive, i.e., $\gamma(\sqcap X) = \bigcap \{\gamma(d) \mid d \in X\}$ for all $X \subseteq \mathbb{D}$.

The most important reason for making this assumption is that it ensures that we can validly compute on representations without loosing precision: if we precisely mirror the equations characterizing weakest preconditions on representations, the largest solution of the resulting equation system on representations characterizes the representation of the weakest precondition by the following well-known lemma. It appears in the literature (for the dual situation of least fixpoints) under the name Transfer Lemma [4] or $\mu$-Fusion Rule [44].

**Lemma 4.2** Suppose $L, L'$ are complete lattices, $f : L \rightarrow L$ and $g : L' \rightarrow L'$ are monotonic functions and $\gamma : L \rightarrow L'$ (cf. Fig. 4.4).

If $\gamma$ is universally conjunctive and $\gamma \circ f = g \circ \gamma$ then $\gamma(\nu f) = \nu g$, where $\nu f$ and $\nu g$ are the largest fixpoints of $f$ and $g$, respectively.

We must mirror the elements comprising the equation system characterizing weakest preconditions on representations precisely. Firstly, we must represent the start value, $W_0$. Universal conjunctivity of $\gamma$ implies that $\gamma(\top) = \Sigma$, i.e., the top value of $\mathbb{D}$ is a precise representation of $\Sigma$. In addition, we require:
4.4. DETECTION OF PRESBURGER CONSTANTS

(c) Assertion \( x = c \) can be represented precisely: for each \( x \in X, c \in \text{Val} \) we can effectively determine \( d_{x=c} \in \mathbb{D} \) with \( \gamma(d_{x=c}) = \{ \sigma \in \Sigma \mid \sigma(x) = c \} \).

Secondly, we need effective representations for the operators appearing in equations. Requirement (b) implies that the meet operation of \( \mathbb{D} \) precisely abstracts intersection of assertions. In order to enable effective computation of intersections, we require in addition:

(d) for given \( d, d' \in \mathbb{D} \), we can effectively compute \( d \cap d' \).

By induction this implies that we can compute finite meets \( d_1 \cap \ldots \cap d_k \) effectively.

The only remaining operations on assertions are the weakest precondition transformers of basic statements. We must represent \( \text{wp}(x := t) \) for expressions \( t \in S \), which is the substitution operator \( (\cdot)[t/x] \) on assertions. As the \( S \)-constant detection algorithm computes the weakest precondition in weakened flow graph \( G_S \), assignments \( x := t \) with \( t \notin S \) do not occur.

(e) There is a computable substitution operation \( (\cdot)[t/x] : \mathbb{D} \to \mathbb{D} \) for each \( x \in X, t \in S \), which satisfies \( \gamma(d[t/x]) = \gamma(d)[t/x] \) for all \( d \in \mathbb{D} \).

Obviously, \( \text{wp}(\text{skip}) \), the identity, is precisely represented by the identity on \( R \). Thus, it remains to represent \( \text{wp}(x := ?) \):

(f) There is a computable projection operation \( \text{proj}_i : \mathbb{D} \to \mathbb{D} \) for each variable \( x_i \in X \) such that \( \gamma(\text{proj}_i(d)) = \forall x_i(\gamma(d)) \) for all \( d \in \mathbb{D} \).

Finally, we need the following in order to make Phase 3 of the algorithm effective.

(g) Assertion true is decidable, i.e., there is a decision procedure that decides for a given \( d \in \mathbb{D} \), whether \( \gamma(d) = \Sigma \) or not.

If, for a given set \( S \subseteq \text{Expr} \), we can find a lattice satisfying requirements (a)–(g), we can effectively execute the three phase algorithm from the beginning of this section by representing assertions by elements from this lattice. This results in a detection algorithm for \( S \)-constants.

In this chapter we are interested in detection of Presburger and polynomial constants. Thus, from now on, let \( \text{Val} = \mathbb{Z} \).

4.4 Detection of Presburger Constants

Before we turn attention to detection of polynomial constants let us explain that the detection algorithm for Presburger constants that has informally been presented in Section 4.2 is an instance of the generic algorithm described in Section 4.3. Let \( S = \{ c_0 + \sum_{i=1}^{n} c_i x_i \mid c_0, \ldots, c_n \in \mathbb{Z} \} \). In the algorithm of Section 4.2 assertions are represented by affine vector spaces in \( \mathbb{Q}^n \). In addition
we need the empty set for representing the assertion false = ∅. Thus, \( \mathcal{D} = \{ z + U \mid z \in \mathbb{Q}^n, \ U \text{ is a subspace of } \mathbb{Q}^n \} \cup \{ \emptyset \} \). For the remainder of this section, we adopt the convention to consider the empty set an affine space. We write 0 for a matrix or vector with zero entries and rely on the context to resolve the ambiguity inherent in this convention.

The order on \( \mathcal{D} \) is set union: \( \emptyset = \emptyset \). From linear algebra we know that the intersection of arbitrary affine spaces is again an affine space. Thus, \( (\mathcal{D}, \subseteq) \) is a complete lattice with intersection as its meet operation. Note, however, that the join operation of this lattice, \( \cup \), is different from set union \( \cup \) because the union of affine spaces is in general not an affine space. The join of a family \( \mathcal{A} \subseteq \mathcal{D} \) is the smallest affine space that contains all members of \( \mathcal{A} \): \( \bigcup \mathcal{A} = \bigcap\{ B \in \mathcal{D} \mid \forall A \in \mathcal{A} : A \subseteq B \} \).

The representation mapping \( \gamma : \mathcal{D} \to 2^\Sigma \) is defined by

\[
\gamma(d) := \{ \sigma \in 2^\Sigma \mid (\sigma(x_1), \ldots, \sigma(x_n)) \in d \}.
\]

As we are using affine subspaces of \( \mathbb{Q}^n \) to represent assertions for integer variables, the representation mapping \( \gamma \) does two things. Firstly, it transfers the tuple representation to a state representation which is merely an isomorphic transformation. Secondly, it selects the integer tuples from the given affine space \( d \subseteq \mathbb{Q}^n \).

From linear algebra we know that all affine spaces \( z + U \in \mathcal{D} \) can be represented by a matrix \( A \in \mathbb{Q}^{k \times n} \) with \( k \leq n \) and a (column) vector \( b \in \mathbb{Q}^k \), such that \( z + U = \{ x \in \mathbb{Q}^n \mid Ax = b \} \). The empty set can also be represented by a matrix and a vector, e.g., by \( A = (0, \ldots, 0) \) and \( b = (1) \). In the concrete algorithm the elements of \( \mathcal{D} \) are represented in this way by a matrix \( A \) and a vector \( b \) but this further representation step is suppressed in this section. We show, however, that all the needed operations on affine spaces can efficiently be performed on their representation by a matrix and a vector. Conceptually, it is simpler to consider the affine spaces themselves as representations because they are ordered. On pairs \( (A, b) \) we have only the pre-order induced by their interpretation as affine spaces:

\[
(A, b) \leq (A', b') \iff \{ x \mid Ax = b \} \subseteq \{ x \mid A'x = b' \}.
\]

Thus, in order to cover this further representation step also, we would need a more general description of the generic algorithm that permits pre-orders as representations. While it is not hard to develop this more general framework it would obscure the presentation.

Let us now show that the requirements of the generic algorithm are satisfied:

(a) For dimension reasons a properly decreasing chain of affine spaces can have at most length \( n + 1 \).

(b) That the representation mapping \( \gamma \) is universally conjunctive is obvious from the definition.
For $x_i \in X$ and $c \in \mathbb{Z}$, define $d_{x_i = c} \overset{\text{def}}{=} \{(c_1, \ldots, c_n) \in \mathbb{Q}^n \mid c_i = c \}$. Obviously, this set can be represented by the matrix $A = (a_{ij}) \in \mathbb{Q}^{1 \times n}$ defined by $a_{ij} = 1$ and $a_{ij} = 0$ if $j \neq i$ and the vector $b = (c)$: $d_{x_i = c} = \{x \in \mathbb{Q}^n \mid Ax = b \}$. This also shows that $d_{x_i = c}$ is indeed an affine space.

If we are given two affine spaces $d_1 = \{x \in \mathbb{Q}^n \mid A_1 x = b_1 \} \in \mathbb{D}$ and $d_2 = \{x \in \mathbb{Q}^n \mid A_2 x = b_2 \} \in \mathbb{D}$ we can effectively determine a representation of $d_1 \cap d_2 = d_1 \cap d_2$ by normalizing the following equation via Gauss elimination:

$$
\begin{pmatrix}
A_1 \\
A_2
\end{pmatrix} x = \begin{pmatrix}
b_1 \\
b_2
\end{pmatrix}.
$$

Suppose we are given $x_s \in X$ and $e = c_0 + \sum_{i=1}^{n} c_i x_i \in S$. For $x = (x_i) \in \mathbb{Q}^n$, let us write $x[e/x_s]$ for the vector $y = (y_i) \in \mathbb{Q}^n$ with $y_s = c_0 + \sum_{i=1}^{n} c_i x_i$ and $y_i = x_i$ for $i \neq s$. We define the substitution operator $(\cdot)[e/x_s] : \mathbb{D} \rightarrow \mathbb{D}$ by $d[e/x_s] = \{x \in \mathbb{Q}^n \mid x[e/x_s] \in d \}$. This definition directly reflects the definition of substitution on assertions. Therefore, the following lemma is obvious.

**Lemma 4.3 (Adequacy)** $\gamma(d)[e/x_s] = \gamma(d[e/x_s])$.

The following lemma shows that and how the substitution operator can (efficiently) be computed on representations of affine spaces via matrices and vectors. It also implies that $d[e/x_s]$ is indeed an affine space and thus ensures well-definedness of $(\cdot)[e/x_s]$. The lemma formalizes the backwards substitution and subsequent normalization on linear equations in the informal explanation of the local transfer functions of affine assignments in Section 4.2.

Suppose we are given $A = (a_{ij}) \in \mathbb{Q}^{k \times n}$ and $b = (b_i) \in \mathbb{Q}^k$ such that $d = \{x \in \mathbb{Q}^n \mid Ax = b \}$.

**Lemma 4.4 (Computation)** Let $A' = (a'_{ij}) \in \mathbb{Q}^{k \times n}$ with $a'_{is} := a_{is} c_s$ and $a'_{ij} := a_{ij} + a_{is} c_j \text{ for } j \neq s$, and $b' = (b'_i) \in \mathbb{Q}^k$ with $b'_i := b_i - a_{is} c_0$.

Then: $d[e/x_s] = \{x \in \mathbb{Q}^n \mid A' x = b' \}$.

**Proof.** Let $x \in \mathbb{Q}^n$ and $y = x[e/x_s]$. By the definitions, $x \in d[e/x_s]$ if and only if $A y = b$. By the definition of matrix multiplication this is the case if and only if for all $i$, $1 \leq i \leq k$,

$$
\sum_{j=1}^{n} a_{ij} y_j = b_i.
$$

(4.2)
As the sum on the left hand side can be rewritten as follows

\[
\sum_{j=1}^n a_{ij}y_j = \sum_{j=1}^n a_{ij}x_j + a_{is}(c_0 + \sum_{j=1}^n c_jx_j) = \sum_{j=1}^n a'_{ij}x_j + a_{is}c_0,
\]

Equation (4.2) holds if and only if \(\sum_{j=1}^n a'_{ij}x_j = b_i - a_{is}c_0 = b'_i\). Consequently, \(x \in d[e/x_s]\) if and only if \(A'x = \theta\). \(\square\)

(f) Suppose we are given \(x_s \in X\). We define the projection operator \(proj_{x_s} : \mathbb{D} \to \mathbb{D}\) on representations of affine spaces as follows: if \(d = \{x \in \mathbb{Q}^n \mid Ax = b\} \in \mathbb{D}\) then

\[
proj_{x_s}(d) = \begin{cases} 
  d & \text{if } a_{is} = 0 \text{ for all } i \in \{1, \ldots, k\} \\
  \emptyset & \text{otherwise}.
\end{cases}
\]

This definition is motivated by the following intuition: a vector \(x = (x_i) \in \mathbb{Z}^n\) (or \(x \in \mathbb{Q}^n\), this doesn't make any difference) satisfies all the linear equations described by \(Ax = b\) for arbitrary variation of \(x_s\) (in \(\mathbb{Z}\) or \(\mathbb{Q}\)) if and only if all equations are independent of \(x_s\). A formalization of this intuition yields:

**Lemma 4.5 (Adequacy)** \(\forall x_s(\gamma(d)) = \gamma(proj_{x_s}(d))\).

We leave the formal proof, which is similar to the proof of Lemma 4.7 below, to the reader.

It is also not hard to show that the above definition is independent of the representation by a matrix \(A\) and vector \(b\). The crucial lemma, the proof of which is also left to the reader, is this:

**Lemma 4.6 (Well-definedness)** Let \(A = (a_{ij}) \in \mathbb{Q}^{k \times n}\), \(b \in \mathbb{Q}^k\), \(A' = (a'_{ij}) \in \mathbb{Q}^{k' \times n}\), and \(b' \in \mathbb{Q}^{k'}\). Suppose \(\{x \mid Ax = b\} = \{x \mid A'x = b'\} \neq \emptyset\). Then: \(a_{is} = 0\) for all \(i = 1, \ldots, k\) if and only if \(a'_{is} = 0\) for all \(i = 1, \ldots, k'\).

It is immediate from its definition that and how the projection operator can efficiently be computed on the representation of an affine space via a matrix \(A\) and a vector \(b\). We only need to check whether the \(s'\)th row of \(A\) is constantly 0; if this is the case, \(d\) is left unchanged by the projection such that \(proj_{x_s}(d)\) is again represented by \(A\) and \(b\); otherwise the projection of \(d\) is empty and we can use, e.g., \(A = (0, \ldots, 0)\) and \(b = (1)\) for representing \(proj_{x_s}(d)\) because \(\emptyset = \{x \in \mathbb{Q}^n \mid (0, \ldots, 0)x = 1\}\).
(g) In order to check whether an affine space \( d = \{ x \in \mathbb{Q}^n \mid Ax = b \} \in \mathbb{D} \) given by matrix \( A \) and column vector \( b \) represents \( \Sigma \) we need only check whether all entries of \( A \) and \( b \) are zero as witnessed by the following lemma. Obviously this condition can efficiently be decided from \( A \) and \( b \).

\textbf{Lemma 4.7 (Test for true)} \( \gamma(\{ x \in \mathbb{Q}^n \mid Ax = b \}) = \Sigma \) if and only if \( A = 0 \) and \( b = 0 \).

\textbf{Proof.} By definition of \( \gamma \), we have \( \gamma(\{ x \mid Ax = b \}) = \Sigma \) if and only if \( Ad = b \) for all \( d \in \mathbb{Z}^n \). We show that the latter condition holds if and only if \( A = 0 \) and \( b = 0 \).

If, on the one hand, \( A = 0 \) and \( b = 0 \), then we clearly have \( Ad = 0 = b \) for all \( d \in \mathbb{Z}^n \). If, on the other hand, \( Ad = b \) holds for all \( d \in \mathbb{Z}^n \), we have, first of all, \( b = A0 = 0 \). Moreover, all entries of \( A \) must be zero: If \( A \) has a non-zero entry, say \( a_{i,j} \), then the \( j \)’th component of its application to the vector \( d = (d_k) \) with \( d_i = 1 \) and \( d_k = 0 \) for \( k \neq i \) would be \( a_{i,j} \neq 0 = b_j \). \( \square \)

4.5 A Primer on Computable Ideal Theory

They key idea for the detection of polynomial constants is to represent assertions by the zeros of ideals in the polynomial ring \( \mathbb{Z}[x_1, \ldots, x_n] \) and to apply techniques from computable ideal theory. While a full introduction to this area is beyond the scope of this thesis, in this section we recall the facts needed and make some additional observations in Section 4.6. Accessible introductions can be found in standard textbooks on computer algebra. The case of polynomial rings over fields is covered, e.g., by [14, 21, 81], while [50] treats the more general case of polynomial rings over rings, that is of relevance here, as \( \mathbb{Z} \) is an integral domain but not a field.

Recall that \( \mathbb{Z} \) together with addition and multiplication forms a commutative ring, i.e., a structure \((\mathbb{R}, +, \cdot)\) with a non-empty set \( \mathbb{R} \) and two inner operations + and \( \cdot \) such that \((\mathbb{R}, +)\) is an Abelian group, \( \cdot \) is associative and commutative, and the distributive law \( a \cdot (b + c) = a \cdot b + a \cdot c \) is valid for all \( a, b, c \in \mathbb{R} \). On the set of polynomials, \( \mathbb{Z}[x_1, \ldots, x_n] \), we can define addition and multiplication operations in the standard way; this makes \( \mathbb{Z}[x_1, \ldots, x_n] \) a commutative ring as well.

A non-empty subset \( I \subseteq R \) of a ring \( R \) is called an \textit{ideal} if \( a + b \in I \) and \( r \cdot a \in I \) for all \( a, b \in I \), \( r \in R \). The ideal \textit{generated} by a subset \( B \subseteq R \) is

\[ (B) = \{ r_1 \cdot b_1 + \ldots + r_k \cdot b_k \mid r_1, \ldots, r_k \in R, b_1, \ldots, b_k \in B \}, \]

and \( B \) is called a \textit{basis} or \textit{generating system} of \( I \) if \( I = (B) \). An ideal is called \textit{finitely generated} if it has a finite basis \( B = \{ b_1, \ldots, b_m \} \). Hilbert’s famous basis
Theorem tells us that $\mathbb{Z}[x_1, \ldots, x_n]$ is Noetherian, since $\mathbb{Z}$ is Noetherian, i.e., that there are no infinitely long strictly increasing chains $I_1 \subset I_2 \subset I_3 \subset \ldots$ of ideals in $\mathbb{Z}[x_1, \ldots, x_n]$. This implies that every ideal of $\mathbb{Z}[x_1, \ldots, x_n]$ is finitely generated.

It is crucial for our algorithm that we can compute effectively with ideals. While Hilbert's basis theorem ensures that we can represent every ideal of $\mathbb{Z}[x_1, \ldots, x_n]$ by a finite basis, it does not give effective procedures for basic questions like membership tests or equality tests of ideals represented in this way. Indeed, Hilbert's proof of the basis theorem was famous (and controversial) at its time for its non-constructive nature.

Fortunately, the theory of Gröbner bases and the Buchberger algorithm provide a solution for some of these problems. While a complete presentation of this theory is way beyond the scope of this thesis—the interested reader is pointed to the books mentioned above—a few sentences are in order here. A Gröbner basis is a basis for an ideal that has particularly nice properties. From any given finite basis of an ideal the Buchberger algorithm effectively computes a Gröbner basis. There is a natural notion of reduction of a polynomial with respect to a set of polynomials. Reduction of a polynomial $p$ with respect to a Gröbner basis always terminates and yields a unique result. This result is the zero polynomial if and only if $p$ belongs to the ideal represented by the Gröbner basis. Hence reduction with respect to a Gröbner basis yields an effective membership test, that in turn can be used to check equality and inclusion of ideals.

In the terminology of [50], $\mathbb{Z}[x_1, \ldots, x_n]$ is a strongly computable ring. This implies that the following operations are computable for ideals $I, I' \subseteq \mathbb{Z}[x_1, \ldots, x_n]$ given by finite bases $B, B'$ and polynomials $p \in \mathbb{Z}[x_1, \ldots, x_n]$, cf. [50]:

**Ideal membership:** Given an ideal $I$ and a polynomial $p$. Is $p \in I$?

**Ideal inclusion:** Given two ideals $I, I'$. Is $I \subseteq I'$?

**Ideal equality:** Given two ideals $I, I'$. Is $I = I'$?

**Sum of ideals:** Given two ideals $I, I'$. Compute a basis for $I + I' \overset{\text{def}}{=} \{ p + p' \mid p \in I, p' \in I' \}$. As a matter of fact, $I + I' = (B \cup B')$.

**Intersection of ideals:** Given two ideals $I, I'$. Compute a basis for $I \cap I'$.

It is straightforward (and well-known) that $I + I'$ and $I \cap I'$ are again ideals if $I$ and $I'$ are. We can use the above operations as basic operations in our algorithms.

### 4.6 More About $\mathbb{Z}[x_1, \ldots, x_n]$

#### 4.6.1 $\mathbb{Z}[x_1, \ldots, x_n]$ as a Complete Lattice

Interestingly, the ideals in $\mathbb{Z}[x_1, \ldots, x_n]$ form also a complete lattice under subset inclusion $\subseteq$. Suppose we are given a set $\mathcal{I}$ of ideals in $\mathbb{Z}[x_1, \ldots, x_n]$. Then the
4.6. MORE ABOUT $\mathbb{Z}[X_1, \ldots, X_N]$

The largest ideal contained in all ideals in $\mathcal{I}$ obviously is $\bigcap \mathcal{I}$, and the smallest ideal that contains all ideals in $\mathcal{I}$ is $\sum \mathcal{I} := \{ r_1 \cdot a_1 + \ldots + r_k \cdot a_k \mid r_1, \ldots, r_k \in \mathbb{Z}[x_1, \ldots, x_n], a_1, \ldots, a_k \in \bigcup \mathcal{I} \}$. The least element of the lattice is the zero ideal $\{0\}$ that consists only of the zero polynomial and the largest element is $\mathbb{Z}[x_1, \ldots, x_n]$. While this lattice does not have finite height it is Noetherian by Hilbert’s basis theorem such that we can effectively compute least fixpoints of monotonic functions on ideals of $\mathbb{Z}[x_1, \ldots, x_n]$ by standard fixpoint iteration.

4.6.2 Zeros

As mentioned, we represent assertions by the zeros of ideals in our algorithm. A state $\sigma$ is called a zero of polynomial $p$ if $p^\sigma = 0$; we denote the set of zeros of polynomial $p$ by $\mathcal{Z}(p)$. More generally, for a subset $B \subseteq \mathbb{Z}[x_1, \ldots, x_n]$, $\mathcal{Z}(B) = \{ \sigma \mid \forall p \in B : p^\sigma = 0 \}$. For later use some facts concerning zeros are collected in the following lemma, in particular of the relationship of ideal operations with operations on their zeros.

**Lemma 4.8** Suppose $B, B'$ are sets of polynomials, $q$ is a polynomial, $I, I'$ are ideals, and $\mathcal{I}$ is a set of ideals in $\mathbb{Z}[x_1, \ldots, x_n]$.

1. If $B \subseteq B'$ then $\mathcal{Z}(B) \supseteq \mathcal{Z}(B')$.
2. $\mathcal{Z}(B) = \mathcal{Z}(\langle B \rangle) = \bigcap_{p \in B} \mathcal{Z}(p)$. In particular, $\mathcal{Z}(q) = \mathcal{Z}(\langle q \rangle)$.
3. $\mathcal{Z}(\sum \mathcal{I}) = \bigcap \{ \mathcal{Z}(I) \mid I \in \mathcal{I} \}$. In particular, $\mathcal{Z}(I + I') = \mathcal{Z}(I) \cap \mathcal{Z}(I')$.
4. $\mathcal{Z}(\bigcap \mathcal{I}) = \bigcup \{ \mathcal{Z}(I) \mid I \in \mathcal{I} \}$, if $\mathcal{I}$ is finite. In particular, $\mathcal{Z}(I \cap I') = \mathcal{Z}(I) \cup \mathcal{Z}(I')$.
5. $\mathcal{Z}(\{0\}) = \Sigma$ and $\mathcal{Z}(\mathbb{Z}[x_1, \ldots, x_n]) = \emptyset$.
6. $\mathcal{Z}(I) = \Sigma$ if and only if $I = \{0\} = \{0\}$.

**Proof.** We only prove property 4; the proof of the other properties is simpler and is left to the reader. So suppose $\mathcal{I} = \{ I_1, \ldots, I_k \} \subseteq \mathbb{Z}[x_1, \ldots, x_n]$ is a finite set of ideals.

\begin{itemize}
    \item \textbf{`$\supseteq$':} Suppose $\sigma \in \bigcup \{ \mathcal{Z}(I) \mid I \in \mathcal{I} \}$. Then there is $j \in \{1, \ldots, k\}$ with $\sigma \in \mathcal{Z}(I_j)$. Then, by 1., we have $\sigma \in \mathcal{Z}(I_j) \subseteq \mathcal{Z}(\bigcap \mathcal{I})$ because $I_j \supseteq \bigcap \mathcal{I}$.
    \item \textbf{`$\subseteq$':} We use contraposition. So suppose $\sigma \notin \bigcup \{ \mathcal{Z}(I) \mid I \in \mathcal{I} \}$. Then we can choose for each $j = 1, \ldots, k$ a polynomial $p_j \in I_j$ with $p_j^\sigma \neq 0$. For the product of these polynomials we have $\prod_{j=1}^k p_j \in I_j \cap \mathcal{I}$ and $(\prod_{j=1}^k p_j)^\sigma = \prod_{j=1}^k p_j^\sigma \neq 0$. Hence, $\sigma \notin \mathcal{Z}(\bigcap \mathcal{I})$. \hfill \qed
\end{itemize}

Note that the assumption that $\mathcal{I}$ is finite is essential in property 4: if we choose, for instance, $\mathcal{I} = \{(x^i) \mid i > 1\}$ we have $\mathcal{Z}(\bigcap \mathcal{I}) = \mathcal{Z}(\{0\}) = \mathbb{Z}$ but $\bigcup \mathcal{Z}(I) \mid I \in \mathcal{I} \} = \{0\}$ because $\mathcal{Z}(\langle x^i \rangle) = \mathcal{Z}(x^i) = \{0\}$ for all $i > 0$. 

4.6.3 Substitution

Suppose we are given a polynomial \( p \in \mathbb{Z}[x_1, \ldots, x_n] \) and a variable \( x \in X \). We can define a substitution operation on ideals \( I \) as follows: \( I[p/x] = \{ q[p/x] \mid q \in I \} \), where the substitution of polynomial \( p \) for \( x \) in \( q \), \( q[p/x] \), is defined as usual. By definition, \( I[p/x] \) is the smallest ideal that contains all polynomials \( q[p/x] \) with \( q \in I \). From a basis for \( I \), a basis for \( I[p/x] \) is obtained in the expected way: if \( I = (B) \), then \( I[p/x] = \{ b[p/x] \mid b \in B \} \). Thus, we can easily obtain a finite basis for \( I[p/x] \) from a finite basis for \( I \): if \( I = (b_1, \ldots, b_k) \) then \( I[p/x] = (b_1[p/x], \ldots, b_k[p/x]) \). Hence we can add substitution to our list of computable operations.

The substitution operation on ideals defined in the previous paragraph mirrors precisely semantic substitution in assertions which has been defined in connection with \( \wp(x := e) \).

**Lemma 4.9** \( \mathcal{Z}(I)[p/x] = \mathcal{Z}(I[p/x]) \).

We leave the proof of this equation that involves the substitution lemma known from logic to the reader.

4.6.4 Projection

In this section we define projection operators \( \text{proj}_i \), \( i = 1, \ldots, n \), such that for each ideal \( I \), \( \mathcal{Z}(\text{proj}_i(I)) = \forall x_i(\mathcal{Z}(I)) \). Semantic universal quantification over assertions has been defined in connection with \( \wp(x := ?) \).

A polynomial \( p \in \mathbb{Z}[x_1, \ldots, x_n] \) can uniquely be written as a polynomial in \( x_i \) with coefficients in \( \mathbb{Z}[x_1, \ldots, x_{i-1}, x_{i+1}, x_n] \), i.e., in the form \( p = c_kx_i^k + \ldots + c_0x_i^0 \), where \( c_0, \ldots, c_k \in \mathbb{Z}[x_1, \ldots, x_{i-1}, x_{i+1}, x_n] \), and \( c_k \neq 0 \) if \( k > 0 \). We call \( c_0, \ldots, c_k \) the coefficients of \( p \) with respect to \( x_i \) and let \( \mathcal{C}_i(p) = \{ c_0, \ldots, c_k \} \).

**Lemma 4.10** \( \forall x_i(\mathcal{Z}(p)) = \mathcal{Z}(\mathcal{C}_i(p)) \).

**Proof.** Let \( p = c_kx_i^k + \ldots + c_0x_i^0 \) with \( \mathcal{C}_i(p) = \{ c_0, \ldots, c_k \} \).

\( \supseteq \): Let \( \sigma \in \mathcal{Z}(\mathcal{C}_i(p)) \). We have \( c_j^{\sigma[x_j \rightarrow d]} = c_j^\sigma = 0 \) for all \( d \in \mathbb{Z} \), \( j = 0, \ldots, k \), because \( c_j \) is independent of \( x_i \). Hence, \( p^{\sigma[x_i \rightarrow d]} = c_k^{\sigma[x_i \rightarrow d]}d^k + \ldots + c_0^{\sigma[x_i \rightarrow d]}d^0 = 0d^k + \ldots + 0d^0 = 0 \) for all \( d \in \mathbb{Z} \), i.e. \( \sigma \in \forall x_i(\mathcal{Z}(p)) \).

\( \subseteq \): Let \( \sigma \in \forall x_i(\mathcal{Z}(p)) \). Again, we have \( c_j^{\sigma[x_j \rightarrow d]} = c_j^\sigma \) for all \( d \in \mathbb{Z} \), \( j = 0, \ldots, k \), because \( c_k \) is independent of \( x_i \). Therefore, \( c_k^{\sigma[x_i \rightarrow d]}d^k + \ldots + c_0^{\sigma[x_i \rightarrow d]}d^0 = p^{\sigma[x_i \rightarrow d]} = 0 \) for all \( d \in \mathbb{Z} \) because of \( \sigma \in \forall x_i(\mathcal{Z}(p)) \). This means that the polynomial \( c_k^{\sigma}x_i^k + \ldots + c_0^{\sigma}x_i^0 \) vanishes for all values of \( x_i \). Hence, it has more than \( k \) zeros which implies that it is the zero polynomial. Consequently, \( c_j^\sigma = 0 \) for all \( j = 0, \ldots, k \), i.e. \( \sigma \in \mathcal{Z}(\mathcal{C}_i(p)) \). \( \square \)

Suppose \( I \subseteq \mathbb{Z}[x_1, \ldots, x_n] \) is an ideal with basis \( B \).
4.7 Detection of Polynomial Constants

We represent assertions by ideals of the polynomial ring \( \mathbb{Z}[x_1, \ldots, x_n] \) in the detection algorithm for polynomial constants. Thus, let \( \mathcal{D} \) be the set of ideals of \( \mathbb{Z}[x_1, \ldots, x_n] \) and \( \subseteq \) be \( \supseteq \). The representation mapping is \( \gamma(I) = \mathcal{Z}(I) \). Note that the order is reverse inclusion of ideals. This is because larger ideals have smaller sets of zeros. Thus, the meet operation is the sum operation of ideals and the top element is the ideal \( \{0\} = (0) \).

In a practical algorithm, ideals are represented by finite bases. For transparency, we suppress this further representation step but ensure that only operations that can effectively be computed on bases are used.

The lattice \( (\mathcal{D}, \supseteq) \) satisfies requirements (a)-(g) of Section 4.3:

(a) \( \mathbb{Z}[x_1, \ldots, x_n] \) is Noetherian.

(b) By the identity \( \mathcal{Z}(\sum \mathcal{I}) = \bigcap \{ \mathcal{Z}(I) \mid I \in \mathcal{I} \} \), \( \mathcal{Z} \) is universally conjunctive.

(c) Suppose \( x \in X \) and \( c \in \mathbb{Z} \). Certainly, a state is a zero of the ideal generated by the polynomial \( x - c \) if and only if it maps \( x \) to \( c \). Hence, we choose \( d_{x=c} \) as the ideal \( (x - c) \) generated by \( x - c \).

(d) In Section 4.5 we have seen that the sum of two ideals can effectively be computed on bases.

(e) By Section 4.6.3, \( (\cdot)[p/x] \) is an adequate, computable substitution operation.

(f) Again by Section 4.6.4, \( \text{proj}_i \) is an adequate, computable projection operation.

**Lemma 4.11** \( \forall x_i(\mathcal{Z}(I)) = \mathcal{Z}(\bigcup_{f \in B} \mathcal{C}_i(f)) \).

**Proof.** \( \forall x_i(\mathcal{Z}(I)) = \forall x_i(\mathcal{Z}(B)) = \forall x_i(\bigcap_{p \in B} \mathcal{Z}(p)) = \bigcap_{p \in B} \forall x_i(\mathcal{Z}(p)) = \bigcap_{p \in B} (\mathcal{Z}(\mathcal{C}_i(p))) = \mathcal{Z}(\bigcup_{p \in B} \mathcal{C}_i(p)). \)

In view of this formula, it is natural to define \( \text{proj}_i(I) = (\bigcup_{p \in B} \mathcal{C}_i(p)) \) where \( B \) is a basis of \( I \). It is not hard but tedious to show that this definition is independent of the basis; we leave this proof to the reader. Obviously, \( \text{proj}_i \) is effective: if \( I \) is given by a finite basis \( \{b_1, \ldots, b_k\} \) then \( \text{proj}_i(I) \) is given by the finite basis \( \bigcup_{j=1}^k \mathcal{C}_i(b_j) \).

**Corollary 4.12** \( \forall x_i(\mathcal{Z}(I)) = \mathcal{Z}(\text{proj}_i(I)) \).

**Proof.** \( \forall x_i(\mathcal{Z}(I)) = \mathcal{Z}(\bigcup_{p \in B} \mathcal{C}_i(p)) = \mathcal{Z}(\bigcap_{p \in B} \mathcal{C}_i(p))) = \mathcal{Z}(\text{proj}_i(I)) \).
(g) We know that \( \mathcal{Z}(I) = \Sigma \) if and only if \( I = \{0\} \). Moreover, the only basis of the ideal \( \{0\} \) is \( \{0\} \) itself. Hence, in order to decide whether an ideal \( I \) given by a basis \( B \) represents \( \Sigma \), we only need to check whether \( B = \{0\} \).

We can thus apply the generic algorithm from Section 4.3 for the detection of polynomial constants. In order to make this more specific, we put the pieces together, and describe the resulting algorithm in more detail.

Suppose we are given a variable \( x \in X \) and a program point \( w \in N \) in a flow graph \( G = (N, E, A, s, e) \). Then the following algorithm decides whether \( x \) is a polynomial constant at \( w \) or not:

**Phase 1:** Determine a candidate value \( c \in \mathbb{Z} \) for \( x \) at \( w \) by executing an arbitrary (cycle-free) path from \( s \) to \( w \).

**Phase 2:** Associate with each edge \( (u, v) \in E \) a transfer function \( f_{(u,v)} : \mathbb{D} \to \mathbb{D} \) that represents \( \wp(A_S(u, v)) \):

\[
 f_{(u,v)}(I) = \begin{cases} 
 I & \text{if } A(u, v) = \text{skip} \\
 I/p/x & \text{if } A(u, v) = (x := p) \text{ with } p \in \mathbb{Z}[x_1, \ldots, x_n] \\
 \text{proj}_x(I) & \text{if } A(u, v) = (x := t) \text{ with } t \notin \mathbb{Z}[x_1, \ldots, x_n] \\
 \text{proj}_x(I) & \text{if } A(u, v) = (x :=?) 
\end{cases}
\]

Set \( A_0[w] = (x - c) \) and \( A_0[u] = (0) \) for all \( u \in N \setminus \{w\} \) and compute the largest solution (w.r.t. \( \subseteq \supseteq \)) of the equation system

\[
 A[u] = A_0[u] + \sum_{v \in \text{Succ}[u]} f_{(u,v)}(A[v]) \quad \text{for each } u \in N.
\]

We can do this as follows. Starting from \( A_0[u] \) we iteratively compute, simultaneously for all program points \( u \in N \), the following sequences of ideals

\[
 A_{i+1}[u] = A_i[u] + \sum_{v \in \text{Succ}[u]} f_{(u,v)}(A_i[v]).
\]

We stop upon stabilization, i.e., when we encounter an index \( i_a \) such that \( A_{i_a+1}[u] = A_{i_a}[u] \) for all \( u \in N \). Obviously, \( A_0[u] \subseteq A_1[u] \subseteq A_2[u] \subseteq \ldots \), such that computation must terminate eventually because \( \mathbb{Z}[x_1, \ldots, x_n] \) is Noetherian. In this computation we represent ideals by finite bases and perform Gröbner-basis computations in order to check whether \( A_{i+1}[u] = A_i[u] \).

**Phase 3:** Check if the ideal computed for the start node, \( A_i[s] \), is \( (0) \). If so, \( x \) is a polynomial constant of value \( v \) at \( w \); otherwise, \( x \) is not a polynomial constant at \( w \).

\[^4\text{As } A_{i+1}[u] \supseteq A_i[u] \text{ by construction, it suffices to check } A_{i+1}[u] \subseteq A_i[u].\]
4.8. **Conclusion**

In this chapter we have shown that Presburger constants can be detected in polynomial time and that polynomial constants are decidable. These classes are interesting from a practical point of view because the sets of operators $+, -$ and $+, -, \ast$, respectively, are very frequently used, e.g., for computing memory addresses of array components.

The polynomial-constant detection algorithm can easily be extended to handle conditions of the form $p \neq 0$ with $p \in \mathbb{Z}[x_1, \ldots, x_n]$. The weakest precondition is $\text{wp}(p \neq 0)(\phi) = (p \neq 0 \Rightarrow \phi) = (p = 0 \lor \phi)$ and if $\phi$ is represented by an ideal $I$, the assertion $p = 0 \lor \phi$ is represented by the ideal $I \cap \langle p \rangle$ according to Lemma 4.8. This observation can be used to handle such conditions in our algorithm. We can extend this easily to an arbitrary mixture of disjunctions and conjunctions of conditions of the form $p \neq 0$. Of course, we cannot handle the dual form of conditions, $p = 0$: with both types of conditions we can obviously simulate two-counter machines. In contrast, the Presburger constant detection algorithm cannot easily be extended to conditions as affine spaces are not closed under union.

The detection algorithms of this chapter use an indirect three phase approach; the main work is done in the second phase. In the first phase a candidate value is computed that is verified in the second and third phase by means of a symbolic weakest precondition computation. We have analyzed the demands for making this general algorithmic idea effective which results in a generic framework for the construction of $S$-constant-propagation algorithms. Assertions are represented by affine subspaces of $\mathbb{Q}^n$ for Presburger constants and by ideals in the polynomial ring $\mathbb{Z}[x_1, \ldots, x_n]$ for polynomial constants.

Standard constant propagation relies on forward propagation while we use backwards propagation of assertions. Interestingly, Presburger constants can also be detected by forward propagation of affine spaces. Karr [35] describes such an
algorithm but does not address completeness issues. In forward propagation of assertions we effectively compute strongest postconditions rather than weakest preconditions and this computation involves union of assertions rather than intersection. Because affine spaces are not closed under union, Karr defines a (complicated) union operator of affine spaces that over-approximates their actual union by an affine space. One is tempted to consider forward propagation of ideals of $\mathbb{Z}[x_1, \ldots, x_n]$. At first glance, this idea looks promising, because ideals are closed under intersection and intersection of ideals corresponds to union of their sets of zeros, such that we can even precisely represent the union of assertions. There is, however, another problem: $\mathbb{Z}[x_1, \ldots, x_n]$ is not ‘co-Noetherian’, i.e., there are infinitely long strictly decreasing chains of ideals, e.g., $(x) \supset (x^2) \supset (x^3) \supset \ldots$. Therefore, strongest postcondition computations with ideals cannot be guaranteed to terminate in general.

Our approach to compute weakest preconditions symbolically with effective representations is closely related to abstract interpretation [12, 13]. Requirement (b) of the generic algorithm, universal conjunctivity of the representation mapping $\gamma : \mathbb{D} \to 2^\Sigma$, implies that $\gamma$ has a lower adjoint, i.e., that there is a monotonic mapping $\alpha : 2^\Sigma \to \mathbb{D}$ such that $(\alpha, \gamma)$ is a Galois connection [47]. In the standard abstract interpretation framework, we are interested in computation of least fixpoints and the lower adjoint, $\alpha$, is the abstraction mapping. Here, we are in the dual situation: we are interested in computation of greatest fixpoints. Thus, the role of the abstraction is played by the upper adjoint, $\gamma : \mathbb{D} \to 2^\Sigma$. Funnily, this means that in a technical sense the members of $\mathbb{D}$ provide more concrete information than the members of $2^\Sigma$ and that we compute on the concrete side of the abstract interpretation. Thus, we interpret the lattice $\mathbb{D}$ as an exact partial representation rather than an abstract interpretation. The representation via $\mathbb{D}$ is partial because it does not represent all assertions exactly; this is indispensable due to countability reasons because we cannot represent all assertions effectively. It is an exact representation because it allows us to infer the weakest preconditions arising in the S-constant algorithms precisely, which is achieved by ensuring that the initial value of the fixpoint computation is represented exactly and that the occurring operations on representations mirror the corresponding operations on assertions precisely.

By the very nature of Galois connections, the representation mapping $\gamma$ and its lower adjoint $\alpha$ satisfy the two inequalities $\alpha \circ \gamma \subseteq \text{Id}_\mathbb{D}$ and $\text{Id}_{2^\Sigma} \subseteq \gamma \circ \alpha$, where $\text{Id}_\mathbb{D}$ and $\text{Id}_{2^\Sigma}$ are the identities on $\mathbb{D}$ and $2^\Sigma$, respectively. Interestingly, none of these inequalities degenerates to an equality when we represent assertions by ideals of $\mathbb{Z}[x_1, \ldots, x_n]$ as in our algorithm for detection of polynomial constants. On the one hand, $\gamma \circ \alpha \neq \text{Id}_{2^\Sigma}$ because the representation is necessarily partial. On the other hand, $\alpha \circ \gamma \neq \text{Id}_\mathbb{D}$ because the representation of assertions is not unique. For example, if $p \in \mathbb{Z}[x_1, \ldots, x_n]$ does not have a zero in the integers, we have $\mathbb{Z}(p) = \emptyset$ such that $\mathbb{Z}(p) = \mathbb{Z}(1) = \mathbb{Z}(\mathbb{Z}[x_1, \ldots, x_n])$. But by undecidability of Hilbert’s tenth problem, we cannot decide whether we are faced with such a
4.8. **CONCLUSION**

<table>
<thead>
<tr>
<th>Control Flow</th>
<th>Must-Constants</th>
<th>May-Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>single value</td>
<td>multiple value</td>
</tr>
<tr>
<td>acyclic control flow</td>
<td></td>
<td>Co-NP compl.</td>
</tr>
<tr>
<td>Copy Constants</td>
<td>green</td>
<td>red</td>
</tr>
<tr>
<td>Linear Constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presburger Constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+,-,* Constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full Constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unrestricted control flow</td>
<td></td>
<td>NP complete</td>
</tr>
<tr>
<td>Copy Constants</td>
<td>green</td>
<td>red</td>
</tr>
<tr>
<td>Linear Constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presburger Constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+,-,* Constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full Constants</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Complexity classification of a taxonomy of CP: summary of results.

polynomial $p$ and thus cannot effectively identify $(p)$ and $\Gamma$. This forces us to work with a non-unique representation. While we cannot decide whether the set of zeros of an ideal $I$ given by a basis $B$ is empty, we can decide whether it equals $\Sigma$ because this only holds for $I = (0)$. Fortunately, this is the only question that needs to be answered for the weakest precondition.

As a consequence of non-uniqueness, the weakest precondition computation on ideals does not necessarily stop once it has found a collection of ideals that represents the largest fixpoint on assertions but may proceed to larger ideals that represent the same assertions. Fortunately, we can still prove termination by arguing on ideals directly.

The decidability and complexity results of this and the previous chapter are summarized in Table 4.1. We almost completely succeeded in filling the white fields of Table 3.1. As apparent, only two questions are unsolved so far. Firstly, there is a gap between the lower bound (PSPACE-hardness) and the upper bound (decidability) for polynomial must-constants. Secondly, we miss an upper bound for linear may-constants. To attack these problems opens up opportunities for future research. An observation which is immediately obvious from the table is that the detection of may-constants is significantly harder than detecting their must-counterparts.
Chapter 5

Limits of Parallel Flow Analysis\(^1\)

Automatic analysis of parallel programs is known as a notoriously hard problem. A well-known obstacle is the so-called state-explosion problem: the number of (control) states of a parallel program grows exponentially with the number of parallel components. Therefore, most practical flow analysis algorithms of concurrent programs conservatively approximate the effects arising from interference of different threads in order to achieve efficiency. An excellent survey on practical research towards analysis of concurrent programs with many references is provided by Rinard [69]. In contrast to this research, we are interested in analyses of parallel programs that are exact (or precise) except for the common abstraction of guarded branching to non-deterministic branching that is well-known from analysis of sequential programs.

Surprisingly, certain basic but important dataflow analysis problems can still be solved precisely and efficiently for programs with a fork/join kind of parallelism. Results of this kind have been achieved in recent years by extending fixpoint computation techniques common in classic dataflow analysis to parallel programs [40, 37, 71] and by automata-theoretic techniques [16, 17]. The most general result shown by Seidl and Steffen [71] is that all gen/kill problems\(^2\) can be solved interprocedurally in fork/join parallel programs efficiently and precisely. This comprises the important class of bitvector analyses, e.g., live/dead variable analysis, available expression analysis, and reaching definitions analysis [51].

In this chapter, we consider the question whether these results can be generalized to richer classes of dataflow problems. For this purpose we investigate the complexity of copy-constant detection [18]. Copy-constant detection may be seen as a canonic representative of the next level of difficulty beyond gen/kill problems. In the sequential setting it gives rise to a distributive dataflow framework on a lattices with a small chain height and can thus—by the classic result of Kildall [36, 51]—completely and efficiently be solved by a fixpoint computation.

---

\(^1\)This chapter is based on [55]

\(^2\)Gen/kill problems are characterized by the fact that all transfer functions are of the form \((\lambda x : (x \land a) \lor b)\), where \(a, b\) are constants from the underlying lattice of dataflow facts.
Specifically, we show by means of a reduction from the halting problem for two-counter machines that copy-constant detection is undecidable in parallel programs with procedures (parallel interprocedural analysis). We refine this result by proving copy-constant detection to be PSPACE-complete in case that there are no procedure calls (parallel intraprocedural analysis), and co-NP-complete if also loops are abandoned (parallel acyclic analysis). The latter results are shown by means of reductions from the intersection problem for regular and star-free regular expressions, respectively. These findings render the possibility of complete and efficient dataflow algorithms for parallel programs for more extensive classes of analyses unlikely even for loop-free programs, as it is generally believed that the inclusions $P \subseteq \text{co-NP} \subseteq \text{PSPACE}$ are proper.

The prototypic framework in which these results are obtained poses only rather weak requirements such that the results apply to many concurrent programming languages. In particular, the results are independent of synchronization operations which distinguishes them from previous intractability and undecidability results for synchronization-sensitive flow analysis in parallel languages [77, 65]. They should also be compared to undecidability of LTL model-checking for parallel languages as proved by Bouajjani and Habermehl [6]. While Bouajjani and Habermehl also consider a parallel language without explicit synchronization operations, they use the LTL formula to synchronize the runs of two parallel threads that simulate a two-counter machine. Thus, our results point to a more fundamental limitation for flow analysis of parallel programs as they exploit no synchronization properties.

One remark concerning the parallel composition operator is in order here. It is inherent in the definition of parallel composition that $\pi_1 \parallel \pi_2$ terminates if and when both threads $\pi_1$ and $\pi_2$ terminate (like, for instance, in OCCAM [29]). This means that there is an implicit synchronization between $\pi_1$ and $\pi_2$ at the termination point. However, as explained in Section 5.6, the hardness results remain valid without this assumption. Therefore, they also apply to languages like JAVA in which spawned threads run and terminate independently of the spawning thread.

In order to perform our reductions without relying on synchronization we use a subtle technique involving re-initialization of variables. In all reductions programs are constructed in such a way that certain well-behaved runs simulate some intended behavior, e.g., the execution sequences of the given two-counter machine in the undecidability proof. But we cannot avoid that the constructed programs have also certain runs that bear no correspondence to the behavior to be simulated. Let us call such runs spurious runs for the moment. One would typically use synchronization to exclude spurious runs but in the absence of synchronization primitives this is not possible. In order to solve this problem, we ensure by well-directed re-initialization of variables that the spurious runs do not contribute to the information that is to be determined by the analysis. In order to verify this crucial property in the reductions, we present formal program proofs in the
style of Owicki and Gries [59, 19, 3] for the programs constructed in the reductions. Intuitively, one may interpret the well-directed re-initialization of variables as a kind of “internal synchronization”. However, in contrast to synchronization well-directed re-initialization does not prohibit execution of spurious runs from happening; it only ensures that spurious runs do not influence the analysis result.

In this chapter we assume that each basic statement executes as an atomic step, a standard assumption in verification and analysis of concurrent programs. Although we use only basic statements of a very simple form, we will see in the remaining chapters that this assumption is not as innocent as it may seem: interprocedural copy-constant detection becomes indeed decidable in parallel programs if this assumption is abandoned as we will see in Chapter 9.

### 5.1 A Motivating Example

Before we turn to the technical results, let us discuss a small example that illustrates the subtlety of copy-constant detection in parallel programs and the crucial re-initialization technique in our reductions. Consider the program

$$a := 1; [(b := a; b := 0; c := 0; \text{write}(c)) \parallel c := b].$$

In Fig. 5.1 (a) a control flow graph-like representation of the program is shown and in (b) the def/use relationships between the basic statements. There is a def/use relationship from a statement $S$ to a statement $T$ if there is a program execution in which $S$ updates a variable that is later referenced by $T$ without another update of this variable in between.
Although there is a path in the graph of def/use edges from the initialization \( a := 1 \) to the instruction \( \text{write}(c) \), \( c \) is a (copy) constant of value 0 at the write instruction. In order to see this, consider the following. In any execution, \( c := 0 \) must be executed either after or before \( c := b \) in the parallel thread. If it is executed after \( c := b \) then \( c \) certainly holds 0 at the write statement because 0 is assigned to \( c \) in the last executed assignment, \( c := 0 \). On the other hand, if \( c := 0 \) is executed before \( c := b \) then also the initialization of \( b, b := 0 \), must have been executed before \( c := b \) such that \( c := b \) also loads the value 0 to \( c \). This reasoning exploits the causality inherent in sequential composition.

From this example we learn two things. Firstly, a thread can prohibit a parallel thread from propagating an ‘interesting’ value via a copying assignment \( c := b \) by re-initializing first \( b \) and then \( c \) with an ‘uninteresting’ value. This is exploited in the reductions. Secondly, transitive relationships in the graph of def/use edges do not necessarily correspond to indirect dependences that can be realized in executions, in contrast to the (inprocedural) sequential case. Following transitive relationships in the def/use graph is thus an incomplete (albeit sound) approach for dependency analysis in parallel program. Thus, while we can efficiently determine the def/use relationships in a parallel program—this is a bitvector problem—this information is not as useful as in a sequential program.

## 5.2 Parallel Programs

We consider a prototypic language with shared memory, atomic assignments and fork/join parallelism. A *procedural parallel program* comprises a finite set \( \text{Proc} \) of *procedure names* containing a distinguished name \( \text{Main} \). Each procedure name \( P \) is associated with a statement \( \pi_P \), the corresponding *procedure body*, constructed according to the following grammar, in which \( Q \) ranges over \( \text{Proc} \setminus \{ \text{Main} \} \) and \( x \) over some given finite set of variables:

\[
\begin{align*}
  e & ::= \ c \mid x \\
  \pi & ::= \ x := e \mid \text{write}(e) \mid \text{skip} \mid Q \mid \pi_1;\pi_2 \\
      & \quad \pi_1 \parallel \pi_2 \mid \pi_1 \cap \pi_2 \mid \text{loop } \pi \text{ end}.
\end{align*}
\]

We use the syntax \( \text{procedure } P; \pi_P \text{ end} \) to indicate the association of procedure bodies to procedure names. Note that procedures do not have parameters.

The specific nature of constants and the domain in which they are interpreted is immaterial; we only need that 0 and 1 are two constants representing different values, which—by abuse of notation—are denoted by 0 and 1, too. In other words we only need Boolean variables. The atomic statements of the language are assignment statements \( x := e \) that assign the current value of \( e \) to variable \( x \), the do-nothing statement \( \text{skip} \), and write statements. We use write statements in order to indicate where in the program we are interested in constancy of which
variable; this is the only purpose of write instructions here. A statement of the form \( Q \) denotes a call of procedure \( Q \). The operator ; denotes sequential composition and \( \parallel \) parallel composition. The operator \( \sqcap \) represents non-deterministic branching and \textbf{loop} \( \pi \) \textbf{end} stands for a loop that iterates \( \pi \) an indefinite number of times. Such constructs are chosen in accordance with the abstraction of guarded branching to non-deterministic branching discussed in the introduction. We apply the non-deterministic choice operator also to finite sets of statements; \( \sqcap \{\pi_1, \ldots, \pi_n\} \) denotes \( \pi_1 \sqcap \cdots \sqcap \pi_n \). The ambiguity inherent in this notation is harmless because \( \sqcap \) is commutative, associative, and idempotent semantically.

Note that there are no synchronization operations in the language. The synchronization of start and termination inherent in fork- and join-parallelism is also not essential for our results; see Section 5.6.

Parallelism is understood in an interleaving fashion; assignments and write statements are assumed to be atomic. A run of a program is a maximal sequence of atomic statements that may be executed in this order in an execution of the program. The program \( (x := 1; x := y) \parallel y := x \), for example, has the three runs \( \langle x := 1, x := y, y := x \rangle \), \( \langle x := 1, y := x, x := y \rangle \), and \( \langle y := x, x := 1, x := y \rangle \). We denote the set of runs of program \( \pi \) by \( \text{Runs}(\pi) \).

Note that the prototypic language has only assignments of a very simple form: \( x := k \) where \( k \) is either a constant or a variable. These are just the assignments that are interpreted in copy-constant detection. Consequently, for the prototypic language, constants and copy constants coincide. Hardness results for constant detection in programs of this prototypic language can immediately be interpreted as hardness results for copy-constant detection in more general parallel languages. This justifies to identify for the purpose of this chapter the copy-constant detection problem in parallel programs with the detection problem of constants in programs of the prototypic language.

### 5.3 Interprocedural Copy-Constant Detection

The goal of this section is to prove the following theorem.

#### Theorem 5.1

Parallel interprocedural copy-constant detection is undecidable.

It is well-known that the termination problem for two-counter machines is undecidable [49]. In the remainder of this section, we reduce this problem to an interprocedural copy-constant detection problem thereby proving Theorem 5.1.

### 5.3.1 Two-Counter Machines

A two-counter machine has two counter variables \( c_0 \) and \( c_1 \) that can be incremented, decremented, and tested against zero. It is common to use a combined
chapter5

62

CHAPTER 5. LIMITS OF PARALLEL FLOW ANALYSIS

decrement- and test-instruction in order to avoid complications with decrementing a zero counter. The basic idea of our reduction is to represent the values of the counters by the stack height of two threads of procedures running in parallel. Incrementing a counter is represented by calling another procedure in the corresponding thread, decrementing by returning from the current procedure, and the test against zero by using different procedures at the first and the other stack levels that represent the possible moves for zero and non-zero counters, respectively. It simplifies the argumentation if computation steps involving the two counters alternate. This can always be enforced by adding skip-instructions that do nothing except of transferring control.

Formally, we use the following model. A two-counter machine \( M \) comprises a finite set of (control) states \( S \). \( S \) is partitioned into two sets \( P = \{ p_1, \ldots, p_n \} \) and \( Q = \{ q_1, \ldots, q_m \} \); moves involving counter \( c_0 \) start from \( P \) and moves involving counter \( c_1 \) from \( Q \). Execution commences at a distinguished start state which, without loss of generality, is \( p_1 \). There is also a distinguished final state, without loss of generality \( p_n \), at which execution terminates. Each state \( s \in S \) except of the final state \( p_n \) is associated with an instruction \( I(s) \) taken from the following selection:

- \( c_i := c_i + 1; \text{goto } s' \) (increment),
- \( \text{if } c_i = 0 \text{ then goto } s' \text{ else } c_i := c_i - 1; \text{goto } s'' \) (test-decrement), or
- \( \text{goto } s' \) (skip),

where \( i = 0 \) and \( s', s'' \in Q \) if \( s \in P \), and \( i = 1 \) and \( s', s'' \in P \) if \( s \in Q \). Note that this condition captures that moves alternate.

Execution of a two-counter machine \( M \) is represented by a transition relation \( \rightarrow_M \) on configurations \( \langle s, x_0, x_1 \rangle \) that consist of a current state \( s \in S \) and current values \( x_0 \geq 0 \) and \( x_1 \geq 0 \) of the counters. Configurations with \( s = p_n \) are called final configurations. We have \( \langle s, x_0, x_1 \rangle \rightarrow_M \langle s', x_0', x_1' \rangle \) if and only if one of the following conditions is valid for \( i = 0, 1 \):

- \( I(s) = c_i := c_i + 1; \text{goto } s', x_i' = x_i + 1, \text{ and } x_{1-i}' = x_{1-i} \).
- \( I(s) = \text{if } c_i = 0 \text{ then goto } s' \text{ else } c_i := c_i - 1; \text{goto } s'', x_i = 0, x_i' = x_i, \text{ and } x_{1-i}' = x_{1-i} \).
- \( I(s) = \text{if } c_i = 0 \text{ then goto } s'' \text{ else } c_i := c_i - 1; \text{goto } s', x_i \neq 0, x_i' = x_i - 1, \text{ and } x_{1-i}' = x_{1-i} \).
- \( I(s) = \text{goto } s', x_i' = x_i, \text{ and } x_{1-i}' = x_{1-i} \).

Thus, each non-final configuration has a unique successor configuration. We denote the reflexive transitive closure of \( \rightarrow_M \) by \( \rightarrow_M^* \) and omit the subscript \( M \) if it is clear from the context.
5.3. INTERPROCEDURAL COPY-CONSTANT DETECTION

Execution of a two-counter machine commences at the start state with the counters initialized by zero, i.e. in the configuration \( \langle p_1, 0, 0 \rangle \). The two-counter machine terminates if it ever reaches the final state, i.e. if \( \langle p_1, 0, 0 \rangle \rightarrow^* \langle p_n, x_0, x_1 \rangle \) for some \( x_0, x_1 \). As far as the halting behavior is concerned, we can assume without loss of generality that both counters are zero upon termination. This can be ensured by adding two loops at the final state that iteratively decrement the counters until they become zero. Obviously, this modification preserves the termination behavior of the two-counter machine. Note that for the modified machine the conditions \( \langle p_1, 0, 0 \rangle \rightarrow^* \langle p_n, x_0, x_1 \rangle \) and \( \langle p_1, 0, 0 \rangle \rightarrow^* \langle p_n, 0, 0 \rangle \) are equivalent. We assume in the following that such loops have been added to the given machine.

5.3.2 Constructing a Program

From a two-counter machine we construct a parallel program, \( \pi_M \). For each state \( p_k \in P \) the program uses a variable \( x_k \) and for each state \( q_l \in Q \) a variable \( y_l \). Intuitively, \( x_k \) holds the value 1 in an execution of the program iff this execution corresponds to a run of the two-counter machine reaching state \( p_k \), and similarly for the \( y_l \).

The main procedure of \( \pi_M \) reads as follows:

\[
\begin{align*}
\text{procedure } & \text{Main;} \\
& x_1 := 1; \text{Init;} \\
& (P_0 \parallel Q_0); \\
& (x_n := 0 \triangleright \text{skip}); \text{write}(x_n) \\
\text{end}
\end{align*}
\]

The threads \( P_0 \) and \( Q_0 \) are constructed such that \( M \) terminates if and only if \( x_n \) is not a constant at the write instruction. Note that this implies Theorem 5.1.

The initialization \( x_1 := 1 \) is the only occurrence of the constant 1 in the program; all other variables are initialized by 0. Moreover, all other assignment statements only copy values or re-initialize variables by 0. Thus, \( x_n \) can hold only the values 0 or 1 at the write statement. Clearly, \( x_n \) may hold 0 due to the statement \( (x_n := 0 \triangleright \text{skip}) \) immediately before the write statement. Thus \( x_n \) can only be a constant of value 0, and, obviously, this is the case if and only if \( x_n \) cannot hold value 1 at the write instruction. Thus, we can reformulate the goal of the construction as follows:

\[
M \text{ terminates if and only if } x_n \text{ may hold 1 at the write statement.} \tag{5.1}
\]

The initialization of all variables except \( x_1 \) by 0 reflects that \( p_1 \) is the initial state. For each of the two counters the program uses two procedures, \( P_0 \) and \( P_{\neq 0} \) for counter \( c_0 \) and \( Q_0 \) and \( Q_{\neq 0} \) for counter \( c_1 \). They are defined in Fig. 5.2 and 5.3. We describe \( P_0 \) and \( P_{\neq 0} \) in detail in the following, \( Q_0 \) and \( Q_{\neq 0} \) are completely analogous.


\[\text{procedure } P_0;\] 
\[\text{loop}\]
\[\{p := x_k; \text{KillAll}_P; y_l := p; P \neq 0 |\]
\[I(p_k) = c_0 := c_0 + 1; \text{goto } q_i\} \cap\]
\[\{p := x_k; \text{KillAll}_P; y_l := p |\]
\[I(p_k) = \text{if } c_0 = 0 \text{ then goto } q_i \text{ else } \ldots \} \cap\]
\[\{p := x_k; \text{KillAll}_P; y_l := p | I(p_k) = \text{goto } q_i\}\]
\[\text{end}\]
\[\text{end}\]

\[\text{procedure } P_{\neq 0};\]
\[\text{loop}\]
\[\{p := x_k; \text{KillAll}_P; y_l := p; P \neq 0 |\]
\[I(p_k) = c_0 := c_0 + 1; \text{goto } q_i\} \cap\]
\[\{p := x_k; \text{KillAll}_P; y_l := p | I(p_k) = \text{goto } q_i\}\]
\[\text{end};\]
\[\{p := x_k; \text{KillAll}_P; y_l := p |\]
\[I(p_k) = \text{if } c_0 = 0 \text{ then } \ldots \text{else } \ldots \text{goto } q_i\}\]
\[\text{end}\]

\[\text{procedure KillAll}_P;\]
\[y_1 := 0; \ldots; y_m := 0; q := 0; x_1 := 0; \ldots; x_n := 0\]
\[\text{end}\]

Figure 5.2: Definition of \(P_0\) and \(P_{\neq 0}\).

Intuitively, \(P_0\) and \(P_{\neq 0}\) mirror transitions of \(M\) induced by counter \(c_0\) being \(= 0\) and \(\neq 0\), respectively, hence their name. Each procedure non-deterministically guesses the next transition. Such a transition involves two things: firstly, a state change and, secondly, an effect on the counter value. The state change from some \(p_k\) to some \(q_i\) is represented by copying \(x_k\) to \(y_l\) via an auxiliary variable \(p\) and re-initializing \(x_k\) by zero as part of \(\text{KillAll}_P\). The effect on the counter value is represented by how we proceed:

- For transitions that do not change the counter we jump back to the beginning of the procedure such that other transitions with the same counter value can be simulated subsequently. This applies to skip-transitions and test-decrement transitions for a zero counter, i.e. test-decrement transitions simulated in \(P_0\).

- For incrementing transitions we call another instance of \(P_{\neq 0}\) that simulates the transitions induced by the incremented counter. A return from this new instance of \(P_{\neq 0}\) means that the counter is decremented, i.e. has the old
procedure $Q_0$;
  loop
    $\{ q := y_k ; \text{KillAll}_{Q} ; x_1 := q ; Q_{\neq 0} \}$
    $I(q_k) = c_1 := c_1 + 1 ; \text{goto } p_l \} \cap$
    $\{ q := y_k ; \text{KillAll}_{Q} ; x_1 := q \}$
    $I(q_k) = \text{if } c_1 = 0 \text{ then goto } p_l \text{ else...} \}$
    $\{ q := y_k ; \text{KillAll}_{Q} ; x_1 := q \} \ L(l) = \text{goto } p_l \}$
  end
end

procedure $Q_{\neq 0}$;
  loop
    $\{ q := y_k ; \text{KillAll}_{Q} ; x_1 := q ; Q_{\neq 0} \}$
    $I(q_k) = c_1 := c_1 + 1 ; \text{goto } p_l \} \cap$
    $\{ q := y_k ; \text{KillAll}_{Q} ; x_1 := q \} \ L(l) = \text{goto } p_l \}$
  end;
  $\{ q := y_k ; \text{KillAll}_{Q} ; x_1 := q \}$
  $I(q_k) = \text{if } c_1 = 0 \text{ then...else...goto } p_l \}$
end

procedure KillAll$_Q$;
  $x_1 := 0 ; \ldots ; x_n := 0 ; p := 0 ; y_1 := 0 ; \ldots ; y_m := 0$
end

Figure 5.3: Definition of $Q_0$ and $Q_{\neq 0}$.

value. We therefore jump back to the beginning of the procedure after the return from $P_{\neq 0}$.

* For test-decrement transitions simulated in $P_{\neq 0}$, we leave the current procedure.

This behavior is described in a structured way by means of loops and sequential and non-deterministic composition and is consistent with the representation of the counter value by the number of instances of $P_{\neq 0}$ on the stack.

The problem with achieving (5.1) is that a procedure may try to ‘cheat’: it may execute the code representing a transition from $p_i$ to $q_j$ although $x_i$ does not hold the value 1. If this is a decrementing or incrementing transition the coincidence between counter values and stack heights may then be destroyed and the value 1 may subsequently be propagated erroneously. Cheating may thus invalidate the ‘if’ direction.

This problem is solved as follows. We ensure by appropriate re-initialization that all variables are set to 0 if a procedure tries to cheat. Thus, such executions
cannot contribute to the propagation of the value 1. But re-initializing a set of variables safely is not trivial in a concurrent environment. We have only atomic assignments to single variables available; a variable just set to 0 may well be set to another value by instructions executed by instances of the procedures $Q_0$ and $Q_{\neq 0}$ running in parallel while we are initializing the other variables. Here our assumption that moves involving the counters alternate comes into play. Due to this assumption all copying assignments in $Q_0$ and $Q_{\neq 0}$ are of the form $q := y_i$ or $x_j := q$ ($q$ is the analog of the auxiliary variable $p$). Thus, we can safely assign 0 to the $y_i$ in $P_0$ and $P_{\neq 0}$ as they are not the target of a copy instruction in $Q_0$ or $Q_{\neq 0}$. After we have done so, we can safely assign 0 to $q$; a copy instruction $q := y_i$ executed by the parallel thread cannot destroy the value 0 as all $y_i$ contain 0 already. After that we can safely assign 0 to the $x_i$ by a similar argument. This explains the definition of $\text{KillAll}_P$.

5.3.3 Correctness of the Reduction

From the intuition underlying the definition of $\pi_M$, the ‘only if’ direction of (5.1) is rather obvious: If $M$ terminates, i.e., if it has transitions leading from $\langle p_1, 0, 0 \rangle$ to $\langle p_n, 0, 0 \rangle$, we can simulate these transitions by a propagating run of $\pi_M$. By explaining the definition of $\text{KillAll}_P$, we justified the ‘if’ direction as well. A formal proof can be given along the lines of the classic Owicki/Gries method for proving partial correctness of parallel programs [59, 19, 3]. Although this method is usually presented for programs without procedures it is sound also for procedural programs. In the Owicki/Gries method, programs are annotated with assertions that represent properties valid for any execution reaching the program point at which the assertion is written down. This annotation is subject to certain rules that guarantee soundness of the method.

Specifically, we prove that just before the write instruction in $\pi_M$ the following assertion is valid:

$$x_n = 1 \Rightarrow \langle p_1, 0, 0 \rangle \rightarrow^* \langle p_n, 0, 0 \rangle.$$ 

Validity of this assertion implies the ‘if’ direction of (5.1). The details of this proof are deferred to Section 5.8 in order to increase readability of this chapter.

Our proof should be compared to undecidability of reachability in presence of synchronization as proved by Ramalingam [66], and undecidability of LTL model-checking for parallel languages (even without synchronization) as proved by Bouajjani and Habermehl [6]. Both proofs employ two sequential threads running in parallel. Ramalingam uses the two recursion stacks of the threads to simulate context-free grammar derivations of two words whose equality is enforced by the synchronization facilities of the programming language. Bouajjani and Habermehl use the two recursion stacks to simulate two counters (as we do) whose joint operation then is synchronized through the LTL formula. Thus, both proofs rely on some kind of “external synchronization” of the two threads – which is not
available in our scenario. Instead, our undecidability proof works with “internal synchronization” which is provided implicitly by killing of the circulating value 1 as soon as one thread deviates from the intended synchronous behavior.

5.4 Intraprocedural Copy-Constant Detection

The undecidability result just presented means that we cannot expect to detect all copy constants in parallel programs. Therefore, we must lower our expectation. In dataflow analysis one often investigates also intraprocedural problems. These can be viewed as problems for programs without procedure calls. Here, we find:

**Theorem 5.2** Intraprocedural copy-constant detection is PSPACE-complete for parallel programs.

We can construct a non-deterministic algorithm that determines non-constancy by guessing two runs witnessing different values for the variable in question at the program point of interest. This algorithm can be implemented in polynomial space: In a fork/join parallel program without procedures, the number of threads potentially running in parallel is bounded by the size of the program. Therefore, every run of the program can be simulated by a Turing machine using just a polynomial amount of space. Moreover, as no arithmetic is involved, only values present in the program have to be represented during the computation of the runs. We conclude that the intraprocedural copy-constant detection problem is in NPSPACE=PSPACE.

It remains to show that PSPACE is also a lower bound on the complexity of copy-constant detection, i.e. PSPACE-hardness. This is done by a reduction from the **Regular Expression Intersection** problem. This problem is chosen in favor of the better known intersection problem for finite automata as we are heading for structured programs and not for flow graphs.

An instance of **Regular Expression Intersection** is given by a sequence \( r_1, \ldots, r_n \) of regular expressions over some finite alphabet \( A \). The problem is to decide whether \( L(r_1) \cap \ldots \cap L(r_n) \) is non-empty.

**Lemma 5.3** The **Regular Expression Intersection** problem is PSPACE-complete.

PSPACE-hardness of the **Regular Expression Intersection** problem follows by a reduction from the acceptance problem for linear space bounded Turing machines along the same lines as in the corresponding proof for finite automata [41]. The problem remains PSPACE-complete if we consider expressions without \( \emptyset \).

Suppose now that \( A = \{a_1, \ldots, a_k\} \), and we are given \( n \) regular expressions \( r_1, \ldots, r_n \). In our reduction we construct a parallel program that starts \( n + 1 \)
threads $\pi_0, \ldots, \pi_n$ after some initialization of the variables used in the program:

\begin{verbatim}
procedure Main;
  KillXY_0; \ldots; KillXY_n; x_{n,a_1} \leftarrow 1;
  [\pi_0 \parallel \pi_1 \parallel \cdots \parallel \pi_n];
  (x_{0,a_1} \leftarrow 0 \cap skip); write(x_{0,a_1})
end
\end{verbatim}

The threads refer to variables $x_{i,a}$ and $y_i$ ($i \in \{0, \ldots, n\}$, $a \in A$). Thread $\pi_0$ is defined as follows.

\begin{verbatim}
\pi_0 = loop
  \cap \{y_0 := x_{n,a}; KillAll_0; x_{0,b} := y_0 \mid a, b \in A\}
end
\end{verbatim}

The statement KillAll_0 that is defined below ensures that all variables except $y_0$ are re-initialized by 0 irrespective of the behavior of the other threads as shown below.

For $i = 1, \ldots, n$, the thread $\pi_i$ is induced by the regular expression $r_i$. It is given by $\pi_i = \pi_i(r_i)$, where $\pi_i(r)$ is defined by induction on $r$ as follows.

\begin{verbatim}
\pi_i(\varepsilon) = \text{skip}
\pi_i(a) = y_i := x_{i-1,a}; KillAll_i; x_{i,a} := y_i
\pi_i(r_1 \cdot r_2) = \pi_i(r_1); \pi_i(r_2)
\pi_i(r_1 + r_2) = \pi_i(r_1) \cap \pi_i(r_2)
\pi_i(r^*) = \text{loop } \pi_i(r) \text{ end}
\end{verbatim}

The statement KillAll_i re-initializes all variables except $y_i$. This statement as well as statements KillX_j and KillXY_j on which its definition is based, are defined as follows.

\begin{verbatim}
KillX_j = x_{j,a_1} := 0; \ldots; x_{j,a_k} := 0
KillXY_j = y_j := 0; KillX_j
KillAll_i = KillX_i; KillXY_{i+1}; \ldots; KillXY_n;
                   KillXY_0; \ldots; KillXY_{i-1}
\end{verbatim}

Again it is not obvious that thread $\pi_i$ can safely re-initialize the variables because the other threads may arbitrarily interleave. But by exploiting that only copy instructions of the form $y_j := x_{j-1,a}$ and $x_{j,a} := y_j$ with $j \neq i$ are present in the other threads this can be done by performing the re-initializations in the order specified above.\(^3\) Two crucial properties are exploited for this. First, whenever

\(^3\)Here and in the following, addition and subtraction in subscripts of variables and processes is understood modulo $n + 1$. 


5.4. INTRAPROCEDURAL COPY-CONSTANT DETECTION

\( a := b \) is a copying assignments in a parallel thread, variable \( b \) is re-initialized before \( a \). Therefore, execution of \( a := b \) after the re-initialization of \( b \) just copies the initialization value 0 from \( b \) to \( a \) but cannot destroy the initialization of \( a \). Secondly, in all constant assignments \( a := k \) in parallel threads \( k \) equals 0 such that no other values can be generated.

Altogether, the threads are constructed in such a way that the following is valid.

\[
L(r_1) \cap \ldots \cap L(r_n) \neq \emptyset \text{ if and only if } x_{0,a_1} \text{ is not a constant (of value 0) at the write statement.} \tag{5.2}
\]

Again, the latter is the case if and only if there is a run that propagates the value 1 by which \( x_{n,a_1} \) is initialized to the write-instruction. In the following, we describe the intuition underlying the construction and at the same time prove (5.2).

The threads can be considered to form a ring of processes in which process \( \pi_i \) has processes \( \pi_{i-1} \) as left neighbor and \( \pi_{i+1} \) as right neighbor. Each thread \( \pi_i \) \((i = 1, \ldots, n)\) guesses a word in \( L(r_i) \); thread \( \pi_0 \) guesses some word in \( A^* \). The special form of the threads ensures that they can propagate the initialization value 1 for \( x_{n,a_1} \) if and only if all of them agree on the guessed word and interleave the corresponding runs in a disciplined fashion. Obviously, the latter is possible iff \( L(r_1) \cap \ldots \cap L(r_n) \neq \emptyset \).

Let \( w = c_1 \ldots c_l \) be a word in \( L(r_1) \cap \ldots \cap L(r_n) \) and let \( c_0 = a_1 \), the first letter in alphabet \( A \). In the run induced by \( w \) that successfully propagates the value 1, the threads circulate the value 1 around the ring of processes in the variables \( x_{i,c_j} \) for each letter \( c_i \) of \( w \). We call this the propagation game in the following. At the beginning of the \( j \)-th round, \( j = 1, \ldots, l \), process \( \pi_0 \) 'proposes' the letter \( c_j \) by copying the value 1 from the variable \( x_{n,c_{j-1}} \) to \( x_{0,c_j} \) in which it was left by the previous round or by the initialization, respectively. For technical reasons this copying is done via the 'local' variable\(^4\) \( y_0 \). Afterwards the processes \( \pi_i \) \((i = 1, \ldots, n)\) successively copy the value from \( x_{i-1,c_j} \) to \( x_{i,c_j} \) via their 'local' variables \( y_i \). From \( x_{n,c_j} \) it is copied by \( \pi_0 \) in the next round to \( x_{0,c_{j+1}} \) and so on. After the last round \( (j = l) \) \( \pi_0 \) finally copies the value 1 from \( x_{n,c_l} \) to \( x_{0,a_1} \) and all processes terminate. Writing—by a little abuse of notation—\( \pi_i(a) \) for the single run of \( \pi_i(a) \) and \( \pi_0(a,b) \) for the single run of \( y_0 := x_{n,a} ; \text{KillAll}_0 ; x_{0,b} := y_0 \), we can summarize above discussion by saying that

\[
\begin{align*}
\pi_0(a_1, c_1) \cdot \pi_1(c_1) \cdot \ldots \cdot \pi_n(c_1) \\
\pi_0(c_1, c_2) \cdot \pi_1(c_2) \cdot \ldots \cdot \pi_n(c_2) \\
\vdots \\
\pi_0(c_{i-1}, c_i) \cdot \pi_1(c_i) \cdot \ldots \cdot \pi_n(c_i) \\
\pi_0(c_i, a_1)
\end{align*}
\]

\(^4\)Variable \( y_i \) is not local to \( \pi_i \) in a strict sense. But the other threads do not use it as target or source of a copying assignment; they only re-initialize it.
is a run of $\pi_0 \parallel \ldots \parallel \pi_n$ that witnesses that $x_{0,a_1}$ may hold the value 1 finally, and is thus not a constant at the write statement. This implies the ‘only if’ direction of (5.2).

Next we show that the construction of the threads ensures that runs not following the propagation game cannot propagate value 1 to the write instruction. In particular, if $L(r_1) \cap \ldots \cap L(r_n) = \emptyset$, no propagating run exists, which implies the ‘if’ direction of (5.2).

Note first that all runs of $\pi_i$ are composed of pieces of the form $\pi_i(a)$ and all runs of $\pi_0$ of pieces of the form $\pi_0(a, b)$ which is easily shown by induction. A run can now deviate from the propagation game in two ways. First, it can follow the rules but terminate in the middle of a round:

$$
\pi_0(a_1, c_1) \cdot \pi_1(c_1) \cdot \ldots \cdot \pi_i(c_1) \cdot \ldots \cdot \pi_n(c_1) . \\
\pi_0(a_2, c_2) \cdot \pi_1(c_2) \cdot \ldots \cdot \pi_i(c_2) \cdot \ldots \cdot \pi_n(c_2) . \\
\vdots \\
\pi_0(c_m-1, c_m) \cdot \pi_1(c_m) \cdot \ldots \cdot \pi_i(c_m)
$$

Such a run does not propagate the value 1 to the write instruction as $\text{KillAll}_i$ in $\pi_i(c_m)$ re-initializes $x_{0,a_1}$.

Secondly, a run might cease following the rules of the propagation game after some initial (possibly empty) part. Consider then the first code piece $\pi_i(a)$ or $\pi_0(a, b)$ that is started in negligence of the propagation game rules. It is not hard to see that the first statement in this code piece, $y_i := x_{i-1,a}$ or $y_0 := x_{n,a}$, respectively, then sets the local variable $y_i$ to zero. The reason is that the propagation game ensures that variable $x_{i-1,a}$ or $x_{n,a}$ holds 0 unless the next statement to be executed according to the rules of the propagation game comes from $\pi_i(a)$ or some $\pi_0(a, b)$, respectively. The subsequent statement $\text{KillAll}_i$ or $\text{KillAll}_0$ then irrevocably re-initializes all the other variables irrespective of the behavior of the other threads as we have shown above. Thus, such a run also cannot propagate the value 1 to the write instruction.

An Owicki/Gries style proof that confirms the crucial ‘if’ direction of (5.2) can be found in Section 5.9.

### 5.5 Copy-Constant Detection in Loop-Free Programs

We may lower our expectation even more, and ban not only procedures but also loops from the programs. But even then, copy-constant detection remains intractable, unless P=NP.

**Theorem 5.4** The parallel intraprocedural copy-constant detection problem in loop-free programs is co-NP-complete.
That the problem is in \( \text{co-NP} \) is easy to see. If a variable \( x \) is not a constant at a certain program point \( p \), we can guess two runs of the program that witness different values for \( x \) at \( p \). Each of these runs can involve each statement in the program at most once as the program is loop-free. Hence its length is linear in the size of the given program. As no arithmetic is involved in copy-constant detection, only values present in the input program have to be represented such that the time necessary for guessing the runs is polynomial in the size of the input program.

Co-NP-hardness can be proved by specializing the construction from Section 5.4 to star-free regular expressions. The intersection problem for such expressions is \( \text{NP} \)-complete.

An alternative reduction from the well-known SAT problem is presented in Chapter 10. In contrast to the construction of the current chapter, the reduction there relies only on propagation along copying assignments but not on “quasi-synchronization” through well-directed re-initialization of variables. However, this technique does not seem to generalize to the general intraprocedural and the interprocedural case.

## 5.6 Beyond Fork/Join Parallelism

A weak form of synchronization is inherent in the fork/join parallelism assumed in this chapter, as start and termination of threads is synchronized. The hardness results of this chapter, however, are not restricted to such settings but can also be shown without assuming synchronous start and termination. Therefore, they also apply to languages like JAVA.

The \( \text{PSPACE} \)-hardness proof in Section 5.4, for instance, can be modified as follows. Let \( c, d \) be two new distinct letters and \( B = A \cup \{c, d\} \). Now \( \pi_i \) is defined as \( \pi_i(c \cdot r_i \cdot d) \) and the initialization and the final write instruction is moved to thread \( \pi_0 \). More specifically, \( \pi_0 \) is redefined as follows:

\[
\begin{align*}
\pi_0 & = \text{KillAll}_0; x_{0,c} := 1; \\
\text{loop} & \quad \cap \{ y_0 := x_{n,a}; \text{KillAll}_0; x_{0,b} := y_0 \mid a, b \in B \} \\
\text{end} & \quad (x_{n,d} := 0 \cap \text{skip}; \text{write}(x_{n,d})
\end{align*}
\]

(Of course the statements \( \text{KillX}_i \) have to re-initialize also the new variables \( x_{i,c} \) and \( x_{i,d} \).) Essentially this modification amounts to requiring that the propagation game is played with a first round for letter \( c \)—this ensures a quasi-synchronous start of the threads—and a final round for letter \( d \)—this ensures a quasi-synchronous termination. Thus,

\[
L(r_1) \cap \ldots \cap L(r_n) \neq \emptyset \text{ if and only if } x_{n,d} \text{ is not a constant (of value 0) at the write statement.}
\]
Similar modifications work for the reductions in Section 5.3 and 5.5.

5.7 Owicki/Gries-style Program Proofs

Reasoning about parallel programs is known as a notoriously error-prone activity. The actions of different threads can interleave in many different ways and far too easily certain interleaveings are overlooked that invalidate an informal argument for subtle reasons. In order to safeguard against error in our reasoning, we perform formal program proofs in the style of Owicki and Gries’ classic method [59, 19, 3] that confirm the critical parts of the reasoning in the reductions. In the remainder of this section we briefly recall the Owicki/Gries method and in the following two sections we present the proofs for the critical directions in the undecidability proof of Section 5.3 and the PSPACE-hardness proof of Section 5.4. These sections may safely be skipped on first reading.

The Owicki/Gries method relies on proof outlines which are programs annotated with assertions. Assertions are formulas that represent properties valid for any execution that reaches the program point where the assertion is written down. As usual we write assertions in braces. The annotation is subject to the rules well-known from sequential program proofs. For example if an assignment statement $x := e$ is preceded by an assertion $\{\phi\}$ and followed by an assertion $\{\psi\}$, then $\phi$ must imply $\psi[e/x]$, where $\psi[e/x]$ denotes the assertion obtained by substituting $e$ for $x$ in $\psi$. We assume that the reader is familiar with this style of program proofs (for details see e.g. [59, 19, 3]).

The rule for parallel programs looks as follows [3, Rule 19]:

\[
\text{The standard proof outlines } \{p_i\} S_i^* \{q_i\}, \quad i \in \{1, \ldots, n\}, \text{ are interference free} \\
\{\wedge_{i=1}^n p_i\} \parallel S_1 \parallel \cdots \parallel S_n \parallel \{\wedge_{i=1}^n q_i\}
\]

In this rule $S_i^*$ stands for an annotated version of parallel component $S_i$ and the requirement that the proof outlines for the component programs are ‘standard’ means in our context that every atomic statement is surrounded by assertions.

The crucial additional premise for parallel programs is interference freedom. The following must be true in an interference-free proof outline for a parallel program: Suppose $\{\phi\}$ is an assertion in one parallel component and $S$ is an atomic statement in another parallel component that is preceded by the assertion $\text{pre}(S)$. Then $\{\phi \land \text{pre}(S)\} S \{\phi\}$ must be valid in the usual sense of partial correctness. Intuitively, interference freedom guarantees that validity of an assertion is not destroyed by a thread running in parallel.
5.8 Correctness of the Reduction in Section 5.3

Let us now formally prove the ‘if’ direction of (5.1). We assume all notations and definitions of Section 5.3. As mentioned, we prove that just before the write instruction in $\pi_M$ the following assertion is valid in the sense of partial correctness, i.e., that any execution reaching this program point satisfies the property:

$$x_n = 1 \Rightarrow \langle p_0, 0, 0 \rangle \rightarrow^* \langle p_n, 0, 0 \rangle.$$  \hspace{1cm} (5.3)

Validity of this assertion corresponds directly to the ‘if’ direction of (5.1).

5.8.1 Enriching the Program

Before we discuss proof outlines, we enrich the program $\pi_M$ by two variables $c_0$ and $c_1$ that reflect the values of the counters. Initialization statements $c_0 := 0$ and $c_1 := 0$ are added to the Init procedure. Furthermore, $c_0$ and $c_1$ are incremented and decremented at appropriate places in $P_0$, $P_{\neq 0}$, $Q_0$, and $Q_{\neq 0}$. (For the purpose of performing the proof we allow more general expressions in assignment statements.) Specifically, the code pieces of the form

$$p := x_k \ll P ; y_l := p ; P_{\neq 0}$$

that represent incrementing transitions in $P_0$ and $P_{\neq 0}$ are replaced by

$$p := x_k \ll P ; y_l := p + 1 ; P_{\neq 0}$$

and the code pieces after the loop in $P_{\neq 0}$ that represent decrementing transitions are replaced by

$$p := x_k \ll P ; y_l := p - 1 ; y_l := p.$$  \hspace{1cm} (5.3)

Analogous modifications are made in $Q_0$ and $Q_{\neq 0}$ for counter $c_1$. It is obvious that Assertion (5.3) holds in the modified program if and only if it holds in the original program as $c_0$ and $c_1$ are only used in assignments to themselves. ($c_0$ and $c_1$ are auxiliary variables in the formal sense of the term used in connection with the Owicki/Gries method. It is well-known that the Owicki/Gries method is incomplete without auxiliary variables [19].)

5.8.2 The Proof Outlines

The assertions in the proof ensure that certain configurations are reachable in $M$ if a certain variable in $\pi_M$ holds value 1. We introduce an abbreviation for the formula expressing this fact:

$$OK(x, s, c_0, c_1) \iff x = 1 \Rightarrow \langle p_1, 0, 0 \rangle \rightarrow^* \langle s, c_0, c_1 \rangle$$
Here \( x \) is a variable of the constructed program, \( s \) is a state of the two-counter machine and \( c_0, c_1 \) are expressions involving the auxiliary variables from above. Note that Assertion (5.3) is simply \( \text{OK}(x_n, p_n, 0, 0) \).

The proof outline for the body of procedure \textit{Main} looks as follows. For clarity, we use a comma to denote conjunction in assertions.

\begin{verbatim}
[ 1]  {true}
[ 2]  x_1 := 1;
[ 3]  {x_1 = 1}
[ 4]  \textbf{Init}
[ 5]  \{x_1 = 1, c_0 = 0, c_1 = 0, \land_{i=2}^n x_i = 0, \land_{i=1}^m y_i = 0\}
[ 6]  \{c_0 = 0, c_1 = 0, \land_{i=1}^n \text{OK}(x_i, p_i, c_0, c_1), \land_{i=1}^m \text{OK}(y_i, q_i, c_0, c_1)\}
[ 7]  (P_0 \parallel Q_0);
[ 8]  \{c_0 = 0, c_1 = 0, \land_{i=1}^n \text{OK}(x_i, p_i, c_0, c_1), \land_{i=1}^m \text{OK}(y_i, q_i, c_0, c_1)\}
[ 9]  (x_n := 0 \cap \textbf{skip})
[10]  \{\text{OK}(x_n, p_n, 0, 0)\}
[11] \textbf{write}(x_n)
\end{verbatim}

The obvious proof outline for \textit{Init} is omitted. It is easy to see that line [5] implies the assertion in line [6] as \( \text{OK}(x, s, 0, 0) \) trivially holds if \( x \) holds 0 or if \( s \) is \( p_1 \). Also statement [9] is partially correct with respect to the surrounding assertions; \( x_n := 0 \) establishes Assertion [10] for trivial reasons; and validity for \textbf{skip} follows from the fact that the Assertion [8] implies the Assertion [10] which is obvious.

It remains to show that the statement in line [7], \( P_0 \parallel Q_0 \), is partially correct with respect to the surrounding assertions. For this purpose we show—by interference free proof outlines—that \( P_0 \) and \( Q_0 \) satisfy the following specifications and apply the parallel rule of the Owicki/Gries method:

\[
\begin{align*}
\{c_0 = 0, \land_{i=1}^n \text{OK}(x_i, p_i, c_0, c_1)\} \quad & \quad \{c_1 = 0, \land_{i=1}^m \text{OK}(y_i, q_i, c_0, c_1)\} \\
P_0 \quad & \quad Q_0 \\
\{c_0 = 0, \land_{i=1}^n \text{OK}(x_i, p_i, c_0, c_1)\} \quad & \quad \{c_1 = 0, \land_{i=1}^m \text{OK}(y_i, p_i, c_0, c_1)\}
\end{align*}
\]

Simultaneously, we prove similar specifications for \( P_{\neq 0} \) and \( Q_{\neq 0} \) that are parameterized by a constant \( k > 0 \):

\[
\begin{align*}
\{c_0 = k, \land_{i=1}^n \text{OK}(x_i, p_i, c_0, c_1)\} \quad & \quad \{c_1 = k, \land_{i=1}^m \text{OK}(y_i, q_i, c_0, c_1)\} \\
P_{\neq 0} \quad & \quad Q_{\neq 0} \\
\{c_0 = k - 1, \land_{i=1}^n \text{OK}(x_i, p_i, c_0, c_1)\} \quad & \quad \{c_1 = k - 1, \land_{i=1}^m \text{OK}(y_i, q_i, c_0, c_1)\}
\end{align*}
\]

As we are concerned with partial correctness, it suffices to show that the body of the procedures satisfy these specification, under the assumption that recursive calls do.

In the following we present the proof outlines for \( P_0 \) and \( P_{\neq 0} \) in detail; the proofs for \( Q_0 \) and \( Q_{\neq 0} \) are completely analogous. Afterwards we show interference freedom, a proof that reflects crucial properties of our construction.
5.8. CORRECTNESS OF THE REDUCTION IN SECTION 5.3

The first goal is to show that the precondition of each procedure is an invariant of the loop in the body of that procedure. This amounts to proving that each path through the loop preserves the precondition. Let \( k = 0 \) for the proof in \( P_0 \) and \( k > 0 \) for the proof in \( P_{\neq 0} \).

This is the proof for the paths induced by skip-transitions in both procedures or test-decrement transitions in \( P_0 \):

\[
\begin{align*}
11. & \quad \{ c_0 = k, \ \land_{i=1}^{n} \text{OK}(x_i, p_i, c_0, c_1) \} \\
12. & \quad p := x_k ; \\
13. & \quad \{ c_0 = k, \ \text{OK}(p, p_k, c_0, c_1) \} \\
14. & \quad \text{KillAll}_P \\
15. & \quad \{ c_0 = k, \ \text{OK}(p, p_k, c_0, c_1), \ \land_{i=1}^{m} y_i = 0, \ q = 0, \ \land_{i=1}^{n} x_i = 0 \} \\
16. & \quad y_i := p \\
17. & \quad \{ c_0 = k, \ \land_{i=1}^{n} \text{OK}(x_i, p_i, c_0, c_1) \}
\end{align*}
\]

Instruction [16] leaves all variables \( x_i \) untouched. Hence, it establishes its postcondition [17], because all \( x_i \) are ensured to be zero in [15] and \( \text{OK}(x_i, p_i, c_0, c_1) \) holds trivially if \( x_i = 0 \). It may be surprising that the conjunct \( \text{OK}(p, p_k, c_0, c_1) \) is not needed in this proof because, intuitively, it captures a crucial property of the construction. The reason is that the proofs of \( P_0 \) and \( P_{\neq 0} \) establish only a property about the \( x_i \). The conjunct \( \text{OK}(p, p_k, c_0, c_1) \) is, however, important to ensure interference freedom of [16] with the proof outlines for \( Q_0 \) and \( Q_{\neq 0} \) that concern the variables \( y_i \).

The specification of \( \text{KillAll}_P \), viz. \{[13]\} \text{KillAll}_P \{[15]\}, is again parameterized by a constant \( k \geq 0 \) and is also used in the proof outlines that follow. It is straightforward to construct a proof outline witnessing this specification: the variables that have already been re-initialized are collected in an increasingly larger conjunction.

The proof outline for the paths through the loop bodies induced by incrementing transitions is similar but has to reflect the change of the counter. It also applies the assumption about recursive calls of \( P_{\neq 0} \) (for \( k_{\text{new}} \overset{\text{def}}{=} k + 1 \)):

\[
\begin{align*}
18. & \quad \{ c_0 = k, \ \land_{i=1}^{n} \text{OK}(x_i, p_i, c_0, c_1) \} \\
19. & \quad p := x_k ; \\
20. & \quad \{ c_0 = k, \ \text{OK}(p, p_k, c_0, c_1) \} \\
21. & \quad \text{KillAll}_P \\
22. & \quad \{ c_0 = k, \ \text{OK}(p, p_k, c_0, c_1), \ \land_{i=1}^{m} y_i = 0, \ q = 0, \ \land_{i=1}^{n} x_i = 0 \} \\
23. & \quad c_0 := c_0 + 1 \\
24. & \quad \{ c_0 = k + 1, \ \text{OK}(p, p_k, c_0 - 1, c_1), \ \land_{i=1}^{m} y_i = 0, \ q = 0, \ \land_{i=1}^{n} x_i = 0 \} \\
25. & \quad y_i := p \\
26. & \quad \{ c_0 = k + 1, \ \land_{i=1}^{n} \text{OK}(x_i, p_i, c_0, c_1) \} \\
27. & \quad P_{\neq 0} \\
28. & \quad \{ c_0 = k, \ \land_{i=1}^{n} \text{OK}(x_i, p_i, c_0, c_1) \}
\end{align*}
\]
This completes the proof that the preconditions of $P_0$ and $P_{\neq 0}$ are loop invariants and also finishes the proof outline for $P_0$, as its pre- and postcondition coincide and its body just consists of the loop.

It remains to show that the paths from the loop exit to the procedure exit in $P_{\neq 0}$ induced by decrementing transitions establish the postcondition from the loop invariant, i.e. the precondition of $P_{\neq 0}$:

\[
\begin{align*}
[29] & \{ c_0 = k, \ \bigwedge_{i=1}^{n} \text{OK}(x_i, p_i, c_0, c_1) \} \\
[30] & p := x_k; \\
[31] & \{ c_0 = k, \ \text{OK}(p, p_k, c_0, c_1) \} \\
[32] & \text{KillAll}_P \\
[33] & \{ c_0 = k, \ \text{OK}(p, p_k, c_0, c_1), \ \bigwedge_{i=1}^{m} y_i = 0, \ q = 0, \ \bigwedge_{i=1}^{n} x_i = 0 \} \\
[34] & c_0 := c_0 - 1; \\
[35] & \{ c_0 = k - 1, \ \text{OK}(p, p_k, c_0 + 1, c_1), \ \bigwedge_{i=1}^{m} y_i = 0, \ q = 0, \ \bigwedge_{i=1}^{n} x_i = 0 \} \\
[36] & y_i := p \\
[37] & \{ c_0 = k - 1, \ \bigwedge_{i=1}^{n} \text{OK}(x_i, p_i, c_0, c_1) \}
\end{align*}
\]

5.8.3 Interference Freedom

Let us now check interference freedom. We look at each type of assignment found in $Q_0$ and $Q_{\neq 0}$. It is clear that an assignment to a variable $z$ cannot invalidate conjuncts in assertions that do not mention $z$. Therefore, we only need to consider conjuncts in assertions mentioning the variable to which the statement in question assigns.

- $x_i := 0, y_i := 0, p := 0$: these re-initializing assignment statements cannot invalidate any assertion in the proof outline because all conjuncts that mention the left-hand-side variable trivially hold if the variable is zero. This holds in particular for conjuncts of the form $\text{OK}(x, s, c_0, c_1)$.

- $c_1 := c_1 + 1$ and $c_1 := c_1 - 1$: all conjuncts of the form $\text{OK}(p, p_k, c_0, c_1)$ or $\text{OK}(x_i, p_i, c_0, c_1)$ could potentially be invalidated by these statements. All incrementations and decrements of $c_1$ are however—in analogy to [22] and [33]—guarded by a precondition that ensures that $p$ as well as all variables $x_i$ hold zero, which make $\text{OK}(p, p_k, c_0, c_1)$ or $\text{OK}(x_i, p_i, c_0, c_1)$ true for trivial reasons.

Note that this argument exploits that the variables are re-initialized in order to avoid ‘cheating’.

- $q := y_k$: such a statement could potentially invalidate a conjunct of the form $q = 0$. However, the conjunct $q = 0$ appears in assertions only together with the conjunct $\bigwedge_{i=1}^{m} y_i = 0$. In particular this holds in the (omitted) proof outline for $\text{KillAll}_P$ because the variables $y_i$ are re-initialized before $q$. Therefore, $q := y_k$ cannot destroy validity of the assertion.
5.9. Correctness of the Reduction in Section 5.4

Note that it is essential for this argument that the re-initializations in KillAll$_I$ are done in the correct order as discussed in Section 5.3.2.

- $x_l := q$: such a statement could potentially invalidate conjuncts of the form $x_l = 0$ or OK($x_l, p_l, c_0, c_1$).

All assertions that contain $x_l = 0$ also contain a conjunct $q = 0$. Thus, we can argue as for instructions of the form $q := y_k$.

For conjuncts of the form OK($x_l, p_l, c_0, c_1$) the argument is more subtle. Similarly to [15], [24], and [35], $x_l := q$ is preceded by an assertion that ensures in particular that OK($q, q_k, c_0, c_1 + \iota$) holds, where $\iota \in \{-1, 0, 1\}$. By the construction of $\pi_M$, $\iota = -1, 1, 0$ iff there is a transition from $q_k$ to $p_l$ that increments, decrements, or leaves the counter $c_1$ unchanged, respectively. Now suppose that $x_l$ is assigned the value 1 by $x_l := q$, otherwise OK($x_l, p_l, c_0, c_1$) holds trivially. Then clearly $q = 1$ which implies $\langle p_1, 0, 0 \rangle \rightarrow^* \langle q_k, c_0, c_1 + x \rangle$ by OK($q, q_k, c_0, c_1 + x$). By the transition from $q_k$ to $p_l$, this transition sequence can now be extended to a sequence $\langle p_1, 0, 0 \rangle \rightarrow^* \langle p_l, c_0, c_1 \rangle$. Hence, OK($x_l, p_l, c_0, c_1$) holds.

It is interesting to observe that the crucial properties of the construction are reflected in the interference freedom proof rather than the local proofs. Note, however, that the interference freedom proof exploits the preconditions of the interleaving statements that are established by the local proofs.

5.9 Correctness of the Reduction in Section 5.4

In this section we provide a formal proof of the ‘if’ direction of (5.2). As in Section 5.8 we present an Owicki/Gries-style program proof. Specifically, we show that the assertion

$$x_{0,a_1} = 1 \Rightarrow L(r_1) \cap \ldots \cap L(r_n) \neq \emptyset$$

is valid in the sense of partial correctness just before the write instruction in Main. This suffices to establish the ‘if’ direction of (5.2): if the initialization $x_{n,a_1} := 1$ belongs to the optimal slice, then there is a run that propagates the value 1 from the initialization to the write statement; together with validity of (5.4) at this program point, this implies that $L(r_1) \cap \ldots \cap L(r_n) \neq \emptyset$.

5.9.1 Enriching the Program

In order to perform the proof of (5.4), the threads are enriched by auxiliary variables $w_i$, $i = 0, \ldots, n$, that take values in $A^*$ and record the words guessed
by the threads \( \pi_i \). For this purpose the definition of \( \pi_0 \) and \( \pi_i(a) \) is modified as follows:

\[
\begin{align*}
\pi_0 &= \text{loop} \\
&\quad \cap \{ y_0 := x_{n,a} ; \text{Kil} ; \text{All}_0 ; \} \\
&\quad w_0 := w_0 \cdot b ; x_{0,b} := y_0 \mid a, b \in A \} \\
\text{end} \\
\pi_i(a) &= y_i := x_{i-1,a} ; \text{Kil} ; \text{All}_i ; \text{Kil} ; \text{All}_i ; w_i := w_i \cdot a ; x_{i,a} := y_i .
\end{align*}
\]

The other clauses for \( \pi_i \) are left unchanged. The auxiliary variables \( w_i \) are initialized with the empty word \( \varepsilon \) in the \textit{Main} procedure:

\[
\begin{align*}
\text{procedure Main}; \\
\text{Kil} ; \text{Kil} ; \ldots ; \text{Kil} ; \text{Kil} ; x_{n,a_1} := 1 ; \\
w_0 := \varepsilon ; \ldots ; w_n := \varepsilon ; \\
[\pi_0 \parallel \pi_1 \parallel \ldots \parallel \pi_n] ; \\
(x_{0,a_1} := 0 \cap \text{skip}) ; \\
\text{write}(x_{0,a_1}) \\
\text{end}
\end{align*}
\]

It is obvious that the addition of the variables \( w_i \) does not affect validity of Assertion (5.4).

### 5.9.2 An Auxiliary Predicate

A crucial property of the constructed program is the following: the fact that a certain variable holds the value 1 at a certain point in the program means that the propagation game has been played correctly up to this point in the execution and is in a certain stage. In the formal proof we try to capture the essence of this by an assertion on the words \( w_i \) guessed by the parallel threads so far. To allow a concise statement of the corresponding assertions in the proof of thread \( \pi_i \), we introduce a predicate \( \text{OK}(x, i, c) \) as an abbreviation, where \( x \) is a variable, \( i \in \{1, \ldots, n + 1\} \) is a thread number (\( n + 1 \) stands for thread \( \pi_0 \)) and \( c \in A \) is a letter.

Intuitively, \( \text{OK}(x, i, c) \) expresses the following: if variable \( x \) holds value 1 then all threads \( j < i \) have guessed the same word—as a reference we use word \( w_0 \)—and all threads \( j \geq i \) have guessed the word obtained from \( w_0 \) by removing the last letter; moreover, \( c \) is this last letter. Formally, we define:

\[
\text{OK}(x, i, c) \iff x = 1 \Rightarrow (\bigwedge_{0 \leq j < i} w_0 = w_j \wedge \bigwedge_{i \leq j < n+1} w_0 = w_j \cdot c).
\]

Note that the \( \text{OK} \)-predicate refers to all the variables \( w_i \) but does not list them explicitly in the argument list.
5.9. CORRECTNESS OF THE REDUCTION IN SECTION 5.4

In the following we discuss first the specification of thread \( \pi_0 \) and then a generic specification for the threads \( \pi_i, \ i = 1, \ldots, n \), and give corresponding proof outlines. Afterwards we present the proof outline for the Main procedure and discuss interference freedom. Only validity of non-trivial local proof obligations is discussed in detail.

5.9.3 Proof Outline for \( \pi_0 \)

The specification for \( \pi_0 \) reads as follows:

\[
\{ \wedge_{c \in A} \text{OK}(x_{n,c}, n + 1, c) \} \ \ \ {\pi}_0 \ \ \ {\{ \wedge_{c \in A} \text{OK}(x_{n,c}, n + 1, c) \}}
\]

Note that pre- and postcondition coincide. The specification is shown to be valid by proving that the precondition is an invariant of the loop that constitutes \( \pi_0 \):

1. \( \{ \wedge_{c \in A} \text{OK}(x_{n,c}, n + 1, c) \} \)
2. \( y_0 := x_{n,a} \)
3. \( \{ \text{OK}(y_0, n + 1, a) \} \)
4. \( \text{KillAll}_0 ; \)
5. \( \{ \text{OK}(y_0, n + 1, a), \ \wedge_{j=0}^{n} \wedge_{c \in A} x_{j,c} = 0, \ \wedge_{j=1}^{n} y_j = 0 \} \)
6. \( w_0 := w_0 \cdot b ; \)
7. \( \{ \text{OK}(y_0, 0, b), \ \wedge_{j=0}^{n} \wedge_{c \in A} x_{j,c} = 0, \ \wedge_{j=1}^{n} y_j = 0 \} \)
8. \( x_{0,b} := y_0 \)
9. \( \{ \wedge_{c \in A} \text{OK}(x_{n,c}, n + 1, c) \} \)

In the step from Assertion [5] to [7], only the \text{OK}-predicates are of interest. To see the validity of this step note that \( \text{OK}(y_0, n + 1, a) \) simplifies to

\[
y_0 = 1 \Rightarrow \wedge_{0 \leq j < n + 1} w_0 = w_j
\]

and \( \text{OK}(y_0, 0, b) \) to

\[
y_0 = 1 \Rightarrow \wedge_{0 \leq j < n + 1} w_0 = w_j \cdot b.
\]

The step from Assertion [7] to [9] exploits that \( \text{OK}(x, i, c) \) holds trivially if \( x = 0 \). Interestingly, the conjunct \( \text{OK}(y_0, 0, b) \) is not needed for proving the postcondition [9]. But it is crucial for showing interference freedom of \( x_{0,b} := y_0 \) with the assertion \( \text{OK}(x_{0,b}, 1, b) \) that occurs in the proof outline for \( \pi_1 \). To be complete, we should also state a proof outline for KillAll\(_0\). But this proof outline is straightforward: we simply collect the variables that have already been set to 0 in an increasingly larger conjunction.
5.9.4 Proof Outline for \( \pi_i(r) \)

The specification of thread \( \pi_i \), for \( i = 0, \ldots, n \), reads as follows.

\[
\{ w_i = \varepsilon, \ \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \} \\
\pi_i \\
\{ w_i \in L(r), \ \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \} \quad (5.5)
\]

Thread \( \pi_i = \pi_i(r_i) \) is defined by induction on the structure of the regular expression \( r_i \). In order to show validity of (5.5) we show a generalized specification for \( \pi_i(r) \) also by induction on \( r \):

\[
\{ w_i \in L, \ \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \} \\
\pi_i(r) \\
\{ w_i \in L \cdot L(r), \ \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \}
\]

for any language \( L \subseteq A^* \) and regular expression \( r \). Specification (5.5) then follows by instantiating \( L \) by \( \{ \varepsilon \} \) and \( r \) by \( r_i \).

Now we discuss the proof outlines in the structural induction on \( r \). The proof outline for \( \pi_i(a) \) is similar to the one of \( \pi_0 \). We therefore omit a detailed justification of the local steps.

\[
\begin{align*}
[10] \quad & \{ w_i \in L, \ \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \} \\
[11] \quad & y_i := x_{i-1,a} ; \\
[12] \quad & \{ w_i \in L, \ \text{OK}(y_i, i, a) \} \\
[13] \quad & \text{ KillAll}_i; \\
[14] \quad & \{ w_i \in L, \ \text{OK}(y_i, i, a), \ \land_{j=0}^n \land_{c \in A} x_{j,c} = 0, \ \land_{j \neq i} y_j = 0 \} \\
[15] \quad & w_i := w_i \cdot a ; \\
[16] \quad & \{ w_i \in L \cdot L(a), \ \text{OK}(y_i, i + 1, a), \ \land_{j=0}^n \land_{c \in A} x_{j,c} = 0, \ \land_{j \neq i} y_j = 0 \} \\
[17] \quad & x_{i,a} := y_i \\
[18] \quad & \{ w_i \in L \cdot L(a), \ \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \}
\end{align*}
\]

Again we should carry out a proof for \( \text{KillAll}_i \) for the sake of completeness, but this proof is just as straightforward as the proof for \( \text{KillAll}_0 \) mentioned above.

The proof outline for \( \pi_i(r_1 \cdot r_2) \) is very simple, given that we can apply the induction hypothesis for \( \pi_i(r_1) \) and \( \pi_i(r_2) \):

\[
\begin{align*}
[19] \quad & \{ w_i \in L, \ \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \} \\
[20] \quad & \pi_i(r_1) \\
[21] \quad & \{ w_i \in L \cdot L(r_1), \ \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \} \\
[22] \quad & \pi_i(r_2) \\
[23] \quad & \{ w_i \in L \cdot L(r_1) \cdot L(r_2), \ \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \} \\
[24] \quad & \{ w_i \in L \cdot L(r_1 \cdot r_2), \ \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \}
\end{align*}
\]
5.9. CORRECTNESS OF THE REDUCTION IN SECTION 5.4

In the proof for \( \pi_i(r_1 + r_2) \), we have to show that every component in the non-deterministic choice comprising \( \pi_i(r_1 + r_2) \) satisfies the specification. Using the induction hypothesis this is again quite easy. Suppose \( l \in \{1, 2\} \). Then

\[
\begin{align*}
25 & \{ w_i \in L, \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \} \\
26 & \pi_i(r_l) \\
27 & \{ w_i \in L \cdot L(r_l) \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \} \\
28 & \{ w_i \in L \cdot L(r_1 + r_2) \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \}
\end{align*}
\]

Assertion [27] implies [28], as \( L(r_l) \subseteq L(r_1 + r_2) \).

For \( \pi_i(r^*) \) we have to show validity of

\[
\begin{align*}
\{ w_i \in L, \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \} \\
\text{loop } \pi_i(r) \text{ end} \\
\{ w_i \in L \cdot L(r)^*, \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \}
\end{align*}
\]

We prove that the postcondition is a loop invariant. First of all, it follows from the precondition because \( \varepsilon \in L(r)^* \). Secondly, it is preserved by the loop body, which follows easily from the induction hypothesis and the inclusion \( L(r)^* \cdot L(r) \subseteq L(r)^* \):

\[
\begin{align*}
29 & \{ w_i \in L \cdot L(r)^*, \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \} \\
30 & \pi_i(r) \\
31 & \{ w_i \in L \cdot L(r)^* \cdot L(r), \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \} \\
32 & \{ w_i \in L \cdot L(r)^*, \land_{c \in A} \text{OK}(x_{i-1,c}, i, c) \}
\end{align*}
\]

5.9.5 Proof Outline for Main

Now we are ready to give the proof for the Main procedure that relies on the specifications for the \( \pi_i \) proved above. Note that this proof yields that (5.4) is indeed valid just before the write instruction.

\[
\begin{align*}
33 & \{ \text{true} \} \\
34 & \text{KillXY}_0 : \ldots ; \text{KillXY}_n ; x_{n,a_1} := 1 ; \\
35 & w_0 := \varepsilon ; \ldots ; w_n := \varepsilon ; \\
36 & \{ x_{n,a_1} = 1, \land_{j \in [0,n)} x_{j,c} = 0, \land_{j=0}^n y_j = 0, \land_{j=0}^n w_j = \varepsilon \} \\
37 & [\pi_0 \| \pi_1 \| \ldots \| \pi_n] ; \\
38 & \{ \text{OK}(x_{0,a_1}, 1, a_1), \land_{j=1}^n w_j \in L(r_j) \} \\
39 & (x_{0,a_1} := 0 \sqcup \text{skip}) ; \\
40 & \{ x_{0,a_1} = 1 \Rightarrow L(r_1) \cap \ldots \cap L(r_n) \neq \emptyset \} \\
41 & \text{write}(x_{0,a_1})
\end{align*}
\]

It is obvious that Assertion [36] is established by the initialization. It is also easy to see that [36] implies all the preconditions of the parallel threads: the conjuncts
$w_i = \varepsilon$ in the preconditions of the $\pi_i$, $i = 1, \ldots, n$, are also present in \[36\]. All the other conjuncts found in the preconditions have the form $\text{OK}(x_{j-1}, c_j, c)$ for some $j = 1, \ldots, n + 1$ and $c \in A$. Of these, the predicate $\text{OK}(x_{n+1}, n+1, a_1)$, which is present in the precondition of $\pi_0$, holds, because all the variables $w_j$ are guaranteed by \[36\] to hold the same word $\varepsilon$; and all the other $\text{OK}(x_{j-1}, j, c)$-predicates are trivially valid as the corresponding variable $x_{j-1,c}$ is guaranteed by \[36\] to hold the value 0.

All the conjuncts in Assertion \[38\] are found in the postconditions of the parallel threads: $\text{OK}(x_{0,a_1}, 1, a_1)$ is a conjunct in the postcondition of $\pi_1$ and, for $j = 1, \ldots, n$, $w_j \in L(r_j)$ is a conjunct in the postcondition of $\pi_j$. In the following section we show that the proof outlines for the threads $\pi_i$ are interference-free. We can thus conclude by the parallel rule of the Owicki/Gries method that the step from Assertion \[36\] to \[38\] is valid.

Let us now consider the step from Assertion \[38\] to \[40\]. First of all, $x_{0,a_1} = 1$ establishes Assertion \[40\] for trivial reasons. Correctness of this step for skip holds, because Assertion \[40\] is implied by Assertion \[38\]: as a consequence of $\text{OK}(x_{0,a_1}, 1, a_1)$, $x_{0,a_1} = 1$ implies $w_0 = w_j \cdot a_1$ for $j = 1, \ldots, n$ which in turn implies that all the variables $w_1, \ldots, w_n$ contain the same word. By $\bigwedge_{j=1}^{n} w_j \in L(r_j)$, this word lies in $L(r_1) \cap \ldots \cap L(r_n)$, which consequently is non-empty.

### 5.9.6 Interference Freedom

We now check interference freedom of the local proof outlines for the threads $\pi_i$, $i = 0, \ldots, n$. As in Section 5.8 we look at each type of assignment found in one of the threads and check that it cannot invalidate conjuncts in assertions in other threads that refer to the left hand side variable of that assignment. Throughout this discussion, we suppose $i, j \in \{0, \ldots, n\}$ and use $i$ as the subscript of the thread in which the assignment in question appears. Subscripts of variables and threads are understood modulo $n + 1$.

- $w_i := w_i \cdot a$ in $\pi_i$: none of the assertions in a thread different from $\pi_i$ mentions the variable $w_i$.

- $y_i := x_{i-1,a}$ in $\pi_i$: in other threads $\pi_j$, $j \neq i$, variable $y_i$ is mentioned only in conjuncts of the form $y_i = 0$. However, these conjuncts always appear together with a conjunct $x_{i-1,a} = 0$, which ensures that $y_i := x_{i-1,a}$ does not destroy validity of the assertion. This in particular holds in the omitted straightforward proofs for KillAll due to the order in which the re-initializations are performed. The re-initialization order ensures that variable $x_{i-1,a}$ is re-initialized before $y_i$.

- $x_{i,a} := y_i$ in $\pi_i$: there are two different conjuncts in other threads in which variable $x_{i,a}$ is mentioned. Firstly, it is mentioned in conjuncts of the form $x_{i,a} = 0$. These, however, appear only together with the assertion $y_i = 0$. 
We can thus argue similar to the case of assignment statements of the form $y_i := x_{i-1,a}$.

Secondly, variable $x_{i,a}$ appears in conjuncts of the form $\text{OK}(x_{i,a}, i + 1, a)$ in assertions in $\pi_{i+1}$. Here the precondition of $x_{i,a} := y_i$, viz $\text{OK}(y_i, i + 1, a)$, ensures that $\text{OK}(x_{i,a}, i + 1, a)$ remains valid.

- $y_j := 0$, or $x_{j,c} := 0$ in $\text{KillAll}$: the left hand side variable of these re-initialization statements appears only in conjuncts of the form $z = 0$ or $\text{OK}(z, k, c)$. Both of them are made true by the re-initialization statement for trivial reasons.

### 5.10 Conclusion

In this chapter we have studied the complexity of copy-constant detection in parallel programs, in order to pinpoint limitations of synchronization-independent program analysis. By means of a reduction from the halting problem for two-counter machines, we have shown that the interprocedural problem is undecidable. If we consider programs without procedure calls (intraprocedural problem) copy-constant detection becomes decidable but is still intractable. More specifically, we have shown it to be PSPACE-hard by means of a reduction from the intersection problem for regular expressions. Finally, even if we restrict attention to parallel programs without loops, the problem remains NP-hard. These lower bounds are tight because matching upper bounds are easy to establish.

It is interesting to contrast the results of this chapter to the detection problem for \textit{strong} copy constants. Strong copy constants differ from (full) copy constants in that only constant assignments are taken into account by the analysis. In particular, each variable that is a strong copy constant at a program point $p$ is also a copy constant but not vice versa. The detection of strong copy constants turns out to be a much simpler problem as it can be solved in polynomial time [37, 71].

Previous complexity and undecidability results for dataflow problems for concurrent languages [77, 66] exploit in an essential way synchronization primitives of the considered languages. In contrast our results hold independently of any synchronization. They only exploit interleaving of atomic statements and are thus applicable to a much wider class of concurrent languages. Our results rely, however, on the assumption that basic statements execute atomically. We can show that this assumption is indeed crucial for the undecidability result: in Chapter 9 we show that the interprocedural copy-constants detection problem in parallel programs can indeed be solved (in exponential time) if this assumption is abandoned.

The techniques used here can be used to obtain similar results also for other optimal program analysis problems, in particular, the detection of \textit{truly live vari-
ables and the computation of optimal slices. In fact, the reductions have been presented for slicing originally [55]. True liveness of variables is a refinement of the more well-known notion of live variables that gives rise to a stronger form of dead code elimination known as faint-code elimination [23]. Program slicing [80] is an established program-reduction technique that has applications in program understanding, debugging, and testing [78]. It has also been proposed as a technique for ameliorating the state-explosion problem when formally verifying software or hardware [31, 25, 8, 48].
Chapter 6

Parallel Flow Graphs

In Chapter 5 we have seen that exact copy-constant detection is undecidable for parallel programs with procedures if we assume that assignments execute atomically, a quite common idealization. However, in many execution scenarios for concurrent programs this assumption is hardly realistic (see Chapter 7). Thus, it is interesting to investigate whether these results still hold without the assumption of atomic execution.

Surprisingly, exact copy-constant detection turns out to become decidable, if assignments execute non-atomically. Specifically, we develop an EXPTIME-algorithm for this problem as well as for the elimination of faint code. The crucial new idea is to abstract sets of runs to antichains of short dependence traces, an abstraction that turns out to be precise relative to a semantics capturing non-atomic execution of assignments. Based on the information in these antichains that can effectively be computed in exponential time, the two program analysis problems mentioned above can then be answered easily. As it is somewhat involved to set up the technical framework for these results, they are spread over a number of chapters. In the following we briefly outline the contents of these chapter.

In the current chapter we introduce a flow graph model for parallel programs (cf. [71, 40, 24]). Edges in the flow graph are annotated with a base statement, a call of a single procedure, or a parallel call of two procedures. As base statements we allow assignment statements and the do-nothing statement skip. We assume that branching is non-deterministic, a common abstraction in flow analysis. We define a symbolic operational semantics for parallel flow graphs that captures possible sequences of atomic actions. A sequence of atomic actions is called a run. The symbolic operational semantics is taken as a basis for defining a number of run sets of interest, reaching runs, terminating runs, and bridging runs. We then develop constraint systems that characterize these run sets as the smallest solution of systems of subset constraints. Setting up these constraint systems correctly is easier if we assume atomic execution of base statements. Therefore, in this chapter we still adopt this idealization.
By redefining the operators used in the constraint systems appropriately, we can capture non-atomic execution of base statements. In Chapter 7 we discuss why non-atomic execution is a more realistic assumption and develop a corresponding interpretation of the operators in the constraint systems. This results in a reference semantics that can be used to measure the precision of flow analyzers relative to non-atomic execution of base statements.

We can perform program analysis by solving the constraint systems over an abstract lattice with finite chain height by fixpoint iteration (Appendix A). In Chapter 8 we develop such a lattice, the most important component of which is given by the antichains of dependence traces mentioned above. We define abstract interpretations for the operators in the constraint systems on this lattice and show that these abstract operations are precise abstractions of the operations in the non-atomic execution semantics. By solving the constraint systems developed in the current chapter over this abstract lattice, we can thus do exact interprocedural dependence analysis in parallel programs relative to non-atomic execution. This in turn can be used for exact interprocedural copy-constant propagation and complete faint-code elimination in parallel programs. Corresponding EXPTIME-algorithms are developed in Chapter 9.

Although we have not yet been able to fully characterize the complexity of these two problems in the non-atomic execution scenario, we have made some progress into that direction (Chapter 10). We show that—as in the atomic execution scenario—the loop-free intraprocedural problem is NP-complete. While this implies that also the general intra- and interprocedural problem are intractable it gives no upper bound for their complexity. As a step into that direction we indicate that the general interprocedural problem is unlikely to be in NP, by showing that there are dependences that are mediated only by exponentially long runs. We conjecture that these problems are PSPACE-complete.

### 6.1 Parallel Flow Graphs

There are two reasons for using a flow graph model instead of syntactic programs as in Chapter 5. First of all, it is technically more convenient. The nodes of a flow graph directly correspond to program points. Thus, they provide a natural entity to associate dataflow information with. In contrast, in a syntactic program model there is no entity that directly corresponds to a program point and some way to work around this deficiency has to be found. Nielson, Nielson, and Hankin, for instance, require in their book [58] that each basic statement and condition in a program is annotated with a unique label. In the analyses covered in their book [58] these labels are associated with dataflow information. Using unique labels identifying base-statement instances as a substitute for program points is an elegant albeit non-standard approach.

The second reason for using a flow-graph model in this part of the thesis
is that such a model is slightly more general than a syntactic program model. It also covers programs with unstructured control flow. This makes the positive results shown in this part (decidability of various analysis problems) slightly more general.

It is not hard to describe a translation of syntactic parallel programs as used in Chapter 5 to parallel flow graphs. Because such a translation is tedious to specify and does not give any new insight it is omitted from this thesis.

Let $X$ be a finite set of program variables and $\text{Expr}$ a set of expressions (or terms) over $X$. The precise nature of expressions is immaterial for the moment; we only need that each variable $x \in X$ is also an expression: $X \subseteq \text{Expr}$, and that we can determine for an expression $t \in \text{Expr}$ the set of variables occurring in $t$, $\text{var}(t) \subseteq X$. Let $\text{Stmt} := \{ x := t \mid x \in X, t \in \text{Expr} \} \cup \{ \text{skip} \}$ be the set of base statements. We use $\text{stmt}$ to range over base statements.

Formally, a parallel flow graph comprises a finite set $\text{Proc}$ of procedure names that contains a distinguished procedure $\text{Main}$. Intuitively, $\text{Main}$ is the procedure with which execution starts. Each procedure name $p \in \text{Proc}$ is associated with a control flow graph $G_p = (N_p, E_p, A_p, e_p, r_p)$ that consists of:

- a set $N_p$ of program points;
- a set of edges $E_p \subseteq N_p \times N_p$;
- a mapping $A_p : E_p \to \text{Stmt} \cup \text{Proc} \cup \text{Proc}^2$ that annotates each edge with a base statement, a call of a single procedure, or a parallel call of two procedures; and
- a special entry (or start) point $e_p \in N_p$ and a special return point $r_p \in N_p$.

We assume that the program points of different procedures are disjoint: $N_p \cap N_q = \emptyset$ for $p \neq q$. This can always be enforced by renaming program points.

We write $N$ for $\bigcup_{p \in \text{Proc}} N_p$, $E$ for $\bigcup_{p \in \text{Proc}} E_p$, and $A$ for $\bigcup_{p \in \text{Proc}} A_p$. We also agree that $\text{Base} = \{ e \mid A(e) \in \text{Stmt} \}$ is the set of base edges, $\text{Call}_p = \{ e \mid A(e) = p \}$ is the set of edges that call procedure $p$, and $\text{Pcall}_{pq} = \{ e \mid A(e) = (p, q) \}$ is the set of edges that call procedure $p$ and $q$ in parallel. Moreover, we write $\text{Call}$ for $\bigcup_{p \in \text{Proc}} \text{Call}_p$ and $\text{Pcall}$ for $\bigcup_{p, q \in \text{Proc}} \text{Pcall}_{pq}$.

**Example 6.1** Figure 6.1 shows an example parallel flow graph with three procedures, $\text{Main}$, $p$, and $q$. The entry state of each procedure is marked by an arrow and the return state is indicated by a doubly circled state. The edge annotation $\text{skip}$ is suppressed for clarity.

The main procedure of the example flow graph sequentially starts procedures $p$ and $q$. Procedure $p$ sets variable $y$ to an arbitrary non-negative value and initializes $x$ by 0. Procedure $q$ has a choice: it can execute either the upper path, where it starts two new instances of $q$ in parallel, or the lower path, where it increments $x$ by 2. Note that arbitrarily many instances of $q$ can run in parallel.
Upon termination $y$ can hold an arbitrary non-negative number and $x$ can hold an arbitrary even positive number.

The purpose of the remainder of this chapter is to set up a number of constraint systems, the solutions of which capture certain run sets. In the next section we define an operational semantics that is useful as a reference point for setting up these constraint systems correctly.

6.2 Operational Semantics

We define a symbolic operational semantics of parallel flow graphs that specifies possible sequences of atomic actions. The evaluation of base statements is not described in this semantics. Thus, the configurations of the operational semantic represent control information only. In a sequential flow graph control information is simply given by a single flow-graph node. In a sequential program with procedures configurations would consist of sequences of flow-graph nodes. Such a sequence would model a stack of return addresses (or rather return nodes). In parallel flow graphs procedures can also be called in parallel. We model this by generalizing configurations from sequences to trees. Each node of the tree is labeled by a flow-graph node. Each inner node of the tree has either degree one—such nodes correspond to return addresses from simple calls or to return addresses from parallel calls where one of the parallel threads has terminated already—or degree two—such nodes correspond to return addresses from parallel calls. The active control points are given by the leaves of the tree. Correspondingly, transitions are induced by the leaves. Transitions are labeled by base edges $e$, procedure names $p$, pairs of procedure names $p_0||p_1$, or the symbol $\text{ret}$. There are four transition rules:

**Base Step Rule:** $c \xrightarrow{e} c'$, if $e = (u,v) \in \text{Base}$ and $c'$ results from $c$ by replacing a leaf labeled $u$ by a leaf annotated with $v$. 

Figure 6.1: An example of a parallel flow graph.

```plaintext
Main:

0 1 2
\( p \quad q \)

3 4 5
\( p \xrightarrow{y := 0} x := 0 \)

7 8 9
\( q \xrightarrow{x := x + 2} \)

y := y + 1
```
6.2. OPERATIONAL SEMANTICS

Simple Call Rule: $c \xrightarrow{p} c'$, if there is an edge $e = (u, v) \in \text{Call}_p$ such that $c'$ results from $c$ by replacing a leaf labeled $u$ by a tree consisting of two nodes, a root labeled $v$ and a successor node of the root labeled $e_p$.

Parallel Call Rule: $c \xrightarrow{p_0 \parallel p_1} c'$, if there is an edge $e = (u, v) \in \text{Call}_{p_0, p_1}$ such that $c'$ results from $c$ by replacing a leaf labeled $u$ by a tree consisting of three nodes, a root labeled $v$ with two successor nodes labeled $e_{p_0}$ and $e_{p_1}$.

Return Rule: $c \xrightarrow{\text{ret}} c'$, if $c'$ results from $c$ by removing a leaf labeled by $r_p$ for some $p \in \text{Proc}$.

Note that the father of the node labeled $r_p$ may become a leaf after application of this rule and may thus become active again. This models a return to the stacked return address. Just as well, however, the father of the node labeled $r_p$ may still have a child if it has degree two in $c$ as indicated by the dotted line in the picture. In this case the father becomes active only after the second leaf also vanishes. This models synchronized termination of threads started by a parallel call.

Note also that the application of this rule to a tree consisting of just a root results in the empty tree. Such a step models overall termination.
Let \( \text{Conf} \) be the set of configurations, i.e., trees the degree of which is bounded by two and in which each node is annotated by a program point \( u \in N \). We identify each program point \( u \in N \) with the tree consisting of just a root labeled with \( u \). We also write \( \text{nil} \) for the empty tree. A program point \( u \in N \) is active in a configuration \( c \), if it labels one of the leaves of \( c \). The predicate \( At_u(c) \) is true if \( u \) is active in \( c \) and false otherwise.

Let \( \text{Label} = N \cup \text{Proc} \cup \text{Proc}^2 \cup \{ \text{ret} \} \) be the set of transition labels and \( \rightarrow \subseteq \text{Conf} \times \text{Label} \times \text{Conf} \) be the transition relation defined by the rules above. We define the transitive generalization \( \Rightarrow \subseteq \text{Conf} \times \text{Label}^* \times \text{Conf} \) of \( \rightarrow \), by

\[
\xi \Rightarrow \text{Id} \quad \xi l \Rightarrow \xi l ; l \Rightarrow ,
\]

where ‘;’ denotes relational composition. We write \( \Rightarrow \) for \( \cup_{r \in \text{Label}} \Rightarrow_r \).

### 6.3 Atomic Runs

A sequence of base edges is called an (atomic) run. Correspondingly, the set of atomic runs is \( \text{Runs} = \text{Base}^* \). The classification ‘atomic’ refers to the fact that flow-graph edges constitute atomic entities of execution; in Chapter 7 we shall consider non-atomic runs at length. We define for a label sequence \( l, l \) to be the run obtained from \( l \) by retaining just the base edges and removing everything else:

\[
\hat{e} = \varepsilon \quad \text{and} \quad \hat{e} l = \begin{cases} \hat{e} l & \text{if } l \in \text{Base} \\ \hat{l} & \text{otherwise} \end{cases} \quad \text{for } r \in \text{Label}^*, l \in \text{Label}.
\]

In the following we are going to set up constraint systems for a variety of run sets. These constraint systems use the following small number of operators and constants on run sets.

**Semantics of base edges:** \( [e] = \{ (e) \} \) for \( e \in \text{Base} \). This characterizes the run induced by a base edge in isolation.

**Sequential composition operator:** \( R; S = \{ rs \mid r \in R, s \in S \} \). This characterizes the sequential composition of run sets.

**Interleaving operator:** In order to define the interleaving (or parallel composition) operator some notation is needed. Let \( r = (e_1, \ldots, e_n) \) be a sequence and \( I = \{ i_1, \ldots, i_k \} \) a subset of positions in \( r \) such that \( 1 \leq i_1 < i_2 < \cdots < i_k \leq n \). Then \( r|I \) is the sequence \( (e_{i_1}, \ldots, e_{i_k}) \). We write \( |r| \) for the length of \( r \), viz. \( n \).

Then the interleaving of \( R \) and \( S \) is defined by

\[
R \otimes S = \{ r \mid \exists I_R, I_S : I_R \cup I_S = \{ 1, \ldots, |r| \}, I_R \cap I_S = \emptyset, r|I_R \in R, r|I_S \in S \}.
\]
6.4. THE RUN SETS OF ULTIMATE INTEREST

**Prefix operator:** \( \text{pre}(R) = \{ r | \exists s : rs \in R \} \). This captures prefixes of the runs of \( R \).

**Postfix operator:** \( \text{post}(R) = \{ r | \exists s : sr \in R \} \). This captures postfixes of the runs in \( R \).

Alternatively, atomic runs may be defined as sequences of base statements instead of base edges. The only thing that needs to be changed is to re-define \text{Runs as Stmt} instead of \text{Base} and to re-define \( [e] \) by \( [e] = \{A(e)\} \). The other operators are not affected by this change. In this setting we should also re-define the hat-operator to incorporate the transition from base edges to base statements:

\[
\hat{\varepsilon} = \varepsilon \quad \text{and} \quad \hat{r}l = \begin{cases} \hat{r} \cdot A(l) & \text{if } l \in \text{Base} \\ \hat{r} & \text{otherwise} \end{cases} \quad \text{for } r \in \text{Label}^*, l \in \text{Label}.
\]

The remainder of this chapter can be read with both interpretations.

By re-defining the operators on run sets, we can obtain non-standard semantics. On the one hand, this is used in Chapter 7 for defining a semantics for parallel flow graphs in which execution of base edges is no longer assumed to be atomic. On the other hand, we can re-define these operators on an abstract domain with a finite chain height. Over such a domain we can effectively solve the constraint systems to be introduced soon. If we can show that all operators are correct or even precise abstractions of the concrete operators on atomic or non-atomic run sets, standard abstraction theorems from abstract interpretation ensure that the solution we get is a correct or even precise abstraction of the run sets characterized by the constraint systems. This is the idea of constraint-based program analysis.

6.4 The Run Sets of Ultimate Interest

We are ultimately interested in setting up constraint systems that characterize for each \( u \in N \) the following sets of runs:

**Reaching runs:** \( R(u) = \{ \hat{r} | e_{\text{Main}} \Rightarrow^r_c, At_u(c) \} \).

**Terminating runs:** \( T(u) = \{ \hat{r} | e_{\text{Main}} \Rightarrow c \Rightarrow^r \text{nil}, At_u(c) \} \).

In dataflow analysis one considers forward- and backward-analyses. Forward-analyses calculate abstractions of the reaching runs and backward-analyses abstractions of the terminating runs.

We are also interested for all program points \( u, v \in N \) in the set of those runs that potentially transfer information from \( u \) to \( v \). We call these the bridging runs from \( u \) to \( v \), or \( u-v \)-runs for short.

**Bridging runs:** \( B_{uv}(u) = \{ \hat{r} | e_{\text{Main}} \Rightarrow c_u \Rightarrow^r c_v, At_u(c_u), At_v(c_v) \} \).
In the sections that follow, we present constraint systems that characterize the above run sets. That is: the smallest solution of these constraint systems consists of the run sets defined above. In addition to the above run sets, auxiliary run sets are necessary in order to formulate these constraint systems. These auxiliary run sets are stepwise introduced. We always explain the underlying intuition and outline the correctness proof but leave the details of the proof to the reader. The constraint systems for same-level, reaching and terminating runs are essentially taken from [71] where, however, they are not justified with reference to an explicitly given underlying operational semantics. The constraint system for bridging runs is new.

### 6.5 The Constraint Systems

#### 6.5.1 Same-Level Runs

First of all, we characterize so-called same-level runs. Same-level runs of procedures capture complete runs of procedures in isolation.

**Same-level runs of procedures:** \( S(q) = \{ r \mid e_q \xrightarrow{r} \text{nil} \} \) for \( q \in \text{Proc} \).

As auxiliary sets we consider same-level runs to program nodes.

**Same-level runs to program nodes:** \( S(u) = \{ r \mid e_q \xrightarrow{r} u \} \) for \( u \in N_q, \ q \in \text{Proc} \).

Same-level runs of procedures form an important building block for the other constraint systems. They play a similar role to summary edges in interprocedural program analysis: \(^1\) the same-level runs of procedure \( q \) summarize the complete effect of call edges \( e \in \text{Call}_p \). Also the complete effect of a parallel call edge \( e \in \text{Pcall}_{p_0,p_1} \) is obtained easily from the same-level runs of procedures \( p_0 \) and \( p_1 \): it is given by \( S(p_0) \otimes S(p_1) \).

The same-level runs of procedures and program nodes are the smallest solution of the following constraint system:

- [S1] \( S(q) \supseteq S(r_q) \)
- [S2] \( S(e_q) \supseteq \{ \varepsilon \} \)
- [S3] \( S(v) \supseteq S(u); [e] \), if \( e = (u,v) \in \text{Base} \)
- [S4] \( S(v) \supseteq S(u); S(p) \), if \( e = (u,v) \in \text{Call}_p \)
- [S5] \( S(v) \supseteq S(u); [S(p_0) \otimes S(p_1)] \), if \( e = (u,v) \in \text{Pcall}_{p_0,p_1} \)

It is easy to see that the same-level runs satisfy all constraints:

\(^1\) Indeed, the information associated with a summary edge for \( p \) usually is an abstraction of the same-level runs of \( p \).
6.5. THE CONSTRAINT SYSTEMS

[S1]: A same-level run of the return point of procedure \( q \) gives rise to a same-level run of \( q \) by the Return Rule.

[S2]: It follows trivially from the definition that \( e \) is a same-level run of the entry point of a procedure.

[S3]: If \( e = (u, v) \) is a base edge, we get a same-level run to \( v \) by extending a same-level run to \( u \) with \( e \) by the Base Steps Rule.

[S4]: If \( e = (u, v) \) is an edge that calls \( p \), we get a same-level run to \( v \) if we extend a same-level run to \( u \) by a same-level run of \( p \): we follow the execution underlying the same-level run to \( v \) and apply then call \( p \) according to the Simple Call Rule; we then follow the execution underlying the same-level run of \( p \) (with \( v \) waiting on the stack to become active) and return to \( v \) according to the Return Rule.

[S5]: Similarly, if \( e = (u, v) \) is an edge that calls \( p_0 \) and \( p_1 \) in parallel, we can—after seeing a same-level run to \( u \)—follow this edge; then \( p_0 \) and \( p_1 \) are performed to completion in parallel, which results in an interleaving of a same-level run of \( p_0 \) and \( p_1 \); after that, execution returns to \( v \). We thus obtain a same-level run to \( v \) by extending a same-level run of \( u \) with an interleaving of same-level runs of \( p_0 \) and \( p_1 \).

On the other hand, we can easily prove by induction on the length of the transition sequences inducing same-level runs, that each same-level run lies in any solution of the constraint system, in particular in the smallest one: in the base case we consider the empty execution \( e \). It can only give rise to the same-level run \( e \) to \( e_q \) for some procedure \( q \). But \( e \) is enforced to lie in any solution of \( S(r_p) \) explicitly by constraint [S2].

In the induction step, we consider longer executions leading to same-level runs. The execution underlying a same-level run of a procedure \( q \) necessarily involves a final return from \( r_q \) after an execution that gives rise to a same-level run of \( r_q \). The latter execution is one step shorter and thus the same-level run of \( r_q \) is contained in any solution of \( S(r_q) \) by the induction hypothesis. Now, the constraint [S1] ensures that it is also contained in the set assigned to \( S(q) \) in a solution.

The last step of a non-empty execution \( r \) inducing a same-level run \( \hat{r} \) to a program point \( v \) must be induced either by the Base Rule or the Return Rule because the Simple and Parallel Call Rule never lead to a configuration which consists of just a single state. If the last step is induced by the Base Rule, the previous configuration is a program point \( u \). Then \( \hat{r} \) is composed of a same-level run to \( u \) and the base edge \( e = (u, v) \). The same-level run to \( u \) is induced by a shorter execution and hence contained in the set associated with \( S(u) \) in any solution by the induction hypothesis. Thus, \( \hat{r} \) is in \( S(v) \) by the constraint [S3].
If the last step is induced by the Return Rule, then there must be a simple or parallel call from which this step returns. The constraints for simple and parallel call edges ([S4] and [S5]) together with the induction hypothesis then ensure that \( \hat{r} \) is contained in \( S(v) \).

### 6.5.2 Inverse Same-Level Runs

We also consider a kind of dual to same-level runs of program points: runs from a program point to the return point of the corresponding procedure. We call these *inverse same-level runs of program point*. They are needed in order to capture terminating runs.

**Inverse same-level runs of program points:**

\[
S^i(u) = \{ \hat{r} \mid u \xrightarrow{r} \text{nil} \} \text{ for } u \in N.
\]

Inverse same-level runs of procedures and program nodes are obtained by backwards accumulation as the smallest solution of the following system of constraints:

1. \([\text{SI1}]\) \( S^i(r_q) \supseteq \{ \varepsilon \} \)
2. \([\text{SI2}]\) \( S^i(u) \supseteq [e] ; S^i(v) \), if \( e = (u, v) \in \text{Base} \)
3. \([\text{SI3}]\) \( S^i(u) \supseteq S(p) ; S^i(v) \), if \( e = (u, v) \in \text{Call}_p \)
4. \([\text{SI4}]\) \( S^i(u) \supseteq [S(p_0) \otimes S(p_1)] ; S^i(v) \), if \( e = (u, v) \in \text{Pcall}_{p_0,p_1} \)

The last two constraints refer to same-level runs of procedures. Therefore, it appears that we need to calculate same-level runs before we can calculate inverse same-level runs by the above constraint system. However, by adding for each procedure \( q \in \text{Proc} \) the constraint

\([\text{SI5}]\) \( S(q) \supseteq S^i(e_q) \)

we can calculate same-level runs of procedures simultaneously with inverse same-level runs. Thus, we can also calculate inverse same-level runs in isolation if we wish to do so.

It is easy to see that the sets of inverse same-level runs satisfy all constraints:

- **[SI1]:** By the Return rule, \( \varepsilon \) clearly is an inverse same-level run of the return point \( r_q \) of a procedure.

- **[SI2]:** If \( e = (u, v) \) is a base edge, we get an inverse same-level run of \( u \) by prefixing a same-level run of \( v \) with \( e \).

- **[SI3]:** If \( e = (u, v) \) is an edge that calls \( p \), we can follow this edge in an execution from \( u \); then first \( p \) is performed until termination, which results in a same-level run of \( p \); after that execution proceeds at \( v \). We thus obtain an inverse same-level run of \( u \) by prefixing an inverse same-level run of \( v \) by a same-level run of \( p \).
Similarly, if $e = (u, v)$ is an edge that calls $p_0$ and $p_1$ in parallel, we can follow this edge in an execution from $u$; then first $p_0$ and $p_1$ are performed to completion in parallel, which results in an interleaving of a same-level run of $p_0$ and $p_1$; after that execution returns to $v$. We thus obtain an inverse same-level run of $u$ by prefixing an inverse same-level run of $v$ with an interleaving of same-level runs of $p_0$ and $p_1$.

On the other hand, we can easily prove by induction on the length of the transition sequences inducing inverse same-level runs, i.e. those that lead to nil, that each inverse same-level run is in the smallest solution of the constraint system: in the base case we consider the shortest executions that lead to same-level runs. These are executions of the form $r_p \xrightarrow{\text{ret}} \text{nil}$ for some procedure $p$. They witness that $\varepsilon \in S(r_p)$. But $\varepsilon$ is enforced to be in a solution of $S(r_p)$ explicitly by constraint [SI1].

In the induction step, we consider longer executions leading to same-level runs. These necessarily start with a transition induced by a base edge, a simple, or a parallel call edge. The resulting run is then composed from shorter runs as specified in the constraints for base edges ([SI2]), simple calls ([SI3]), and parallel calls ([SI4]), respectively.

### 6.5.3 Two Assumptions and a Simple Analysis

The following two assumptions simplify the constraint systems that follow:

**ASS1:** every program point $u \in N_q$ in a procedure $q$ can be reached by a same-level run from the entry point $e_q$ of $q$:

$$\forall q \in \text{Proc}, u \in N_q : S(u) \neq \emptyset.$$  

**ASS2:** from every program point $u \in N_q$ the return point $r_q$ can be reached by a same-level run:

$$\forall q \in \text{Proc}, u \in N_q : S^1(u) \neq \emptyset.$$  

These assumptions are not as innocent as it may seem at first glance. In particular it does not suffice to require that there are paths from $e_q$ to $u$ and from $u$ to $r_q$ in the flow graph $G_q$ for $q$. This is just a necessary but not a sufficient condition. The paradigmatic counter-example is a procedure that calls itself and has no bypassing terminating branch:

![Diagram of a recursive call](image)

Although there is a path from $e_q$ to $r_q$ in the flow graph, no execution can reach $r_q$ from $e_q$, as there is no terminating bypass of the recursive call of $q$. Hence
both $S(r_q)$ and $S^i(e_q)$ are empty. Examples like this show that we cannot assume without loss of generality that practical flow graphs satisfy ASS1 and ASS2.

While assumptions ASS1 and ASS2 simplify the presentation and justification of the constraint systems in the remainder of this chapter, they are not strictly necessary. We can well design constraint systems that work in the general case, but they are more complex and therefore harder to explain. In order to avoid overloading the presentation we decided for a two-phase presentation, where we first present and justify the simpler constraint systems that work if ASS1 and ASS2 are satisfied. Afterwards we explain the changes for the general case, cf. Section 6.5.7.

In order to compute the information needed to decide ASS1 and ASS2, we design a simple analysis procedure based on an abstract interpretation of the operators and constants used in the constraint systems. The information computed by this analysis is also used for setting up the constraint systems for the general case. We work with a two point domain $(\mathbb{D} = \{\bot, \top\}, \leq)$ ordered as $\bot \leq \leq \top$. The idea is that $\bot$ represents emptiness of a run set and $\top$ non-emptiness. This is formally captured by the abstraction mapping $\alpha : 2^{\text{Runs}} \rightarrow \mathbb{D}$, defined by $\alpha(\emptyset) = \bot$ and $\alpha(R) = \top$ for $R \neq \emptyset$. Obviously, $\alpha$ is universally disjunctive. We define the abstract interpretation of the operators by

$$x_1 \# y = x \otimes \# y = x \land y, \quad \text{pre} \#(x) = \text{post} \#(x) = x, \quad \{e\} \# = \{e\} \# = \top$$

for $x, y \in \mathbb{D}$, $e \in E$. It is easy to see that the abstract operators are precise abstractions of the corresponding operators on run sets: a sequential or parallel composition of two run sets is non-empty iff both arguments are non-empty; the set of prefixes and the set of postfixes of a run set $R$ are non-empty iff $R$ is; and each base edge gives rise to a non-empty run set. In other words, $\alpha$ is a strong homomorphism in the sense of Appendix A. Therefore, by solving the constraint systems for same-level and inverse same-level runs over the abstract interpretation we get precise information about the emptiness of the sets of same-level and inverse same-level runs of program points.

This analysis is cheap: as $(\mathbb{D}, \leq)$ has chain height two, the information for each constraint variable can change at most once in the fixpoint iteration. By standard demand-driven fixpoint evaluation, we can organize the fixpoint computation such that each operator in the constraint system is evaluated at most once. Thus, the computation can be done in time $O(|E| + |\mathbf{Proc}|)$, the number of operators in the constraint systems. As in all practical flow graphs out-degrees of program nodes are bounded, typically by 2 and $|\mathbf{Proc}|$ is trivially bounded by $|N|$ as each procedure has a distinguished entry node, this is $O(|N|)$. In the following we assume that this analysis has been done such that for each program node $u$ and procedure $q$ the information whether $S(u)$, $S^i(u)$, $S(q)$, or $S^i(q)$ is empty or not is readily available.

Another analysis that can determine information about reachability of program points in parallel flow graphs has been described by Seidl and Steffen [71].
as an instance of their generic analysis framework for solving gen/kill dataflow problems for parallel programs.

### 6.5.4 Reaching Runs

As auxiliary sets for characterizing the runs that reach a program point \( u \), we consider the runs that reach \( u \) from a call to procedure \( q \).

**Reaching runs from procedures:** \( R(u, q) = \{ \hat{r} \mid e_q \xrightarrow{r} c, At_u(c) \} \) for \( u \in N, q \in \text{Proc} \).

With this definition, we obviously have \( R(u) = R(u, \text{Main}) \). Hence we are done with characterizing reaching runs if we succeed in characterizing reaching runs from procedures. The latter can be done by the following constraint system:

\[
\begin{align*}
\text{[R1]} & \quad R(u, q) \supseteq S(u), & \text{if } u \in N_q \\
\text{[R2]} & \quad R(u, q) \supseteq S(v) \cap R(u, p), & \text{if } (v, \_\_) \in E_q \cap \text{Call}_p \\
\text{[R3]} & \quad R(u, q) \supseteq S(v) \cap [R(u, p_i) \otimes \text{pre}(S(p_{i-1})))], & \text{if } (v, \_\_) \in E_q \cap \text{Pcall}_{p_0,p_1}
\end{align*}
\]

The last clause is meant to specify two constraint for \( i = 0 \) and \( i = 1 \).

The reaching runs satisfy the constraints:

**[R1]:** Firstly, each same-level run of \( u \) clearly is also a reaching run of \( u \).

**[R2]:** Secondly, if we have a program point \( v \) in \( q \) that has an outgoing edge calling \( p \)—the situation described in the second constraint—we obtain a run that reaches \( u \) from \( q \) when we extend a same-level run \( \hat{r} \) to \( v \) with a run \( r' \) that reaches \( u \) from \( p \) (where \( r \) and \( r' \) are the underlying executions). The corresponding execution is this: first we follow \( r \); this brings us to \( v \) where we call \( p \); we then follow \( r' \) (with the target node of the edge \((v, \_\_) \in \text{Call}_p \) on the stack).

**[R3]:** Thirdly, consider a program point \( v \) in \( q \) that has an outgoing edge calling \( p_0 \) and \( p_1 \) in parallel, the situation described in the third constraint. Similar to the second case, we get a run reaching \( u \) by extending a same-level run of \( v \) with a run that reaches \( u \) in the parallel call. The latter can happen either in \( p_0 \) or \( p_1 \) hence the two cases with \( i = 0, 1 \). Now until \( p_i \) has reached \( u \) in \( p_i \) the other procedure \( p_{1-i} \) can perform a prefix of a same-level run.

On the other hand, the constraint system captures all the ways how \( u \) may be reached from \( e_q \). There are just three possibilities: either \( u \) is on the same-level, in a simple call, or in a parallel call. These case are completely covered by the constraints.

Note that assumption ASS2 is crucial for making the constraint for parallel calls sufficiently rich. If it is violated, the partial run exhibited by \( p_{1-i} \) while \( p_i \) is
in the process of reaching \( u \) must not be a prefix of a same-level run. For example, the following procedure \( q \) might execute \( x := e \) arbitrarily often, although \( S(q) \) and hence \( \text{pre}(S(q)) \) is empty.

\[
\begin{array}{c}
q: \\
\xrightarrow{x := e} \\
\xrightarrow{e_q} \\
\xrightarrow{q} \\
\xrightarrow{r_q}
\end{array}
\]

A possible remedy is described in Section 6.5.7.

### 6.5.5 Terminating Runs

The approach for capturing terminating runs is dual to the one for reaching runs. As auxiliary sets we consider terminating runs of \( u \) in a call to procedure \( q \).

**Terminating runs in procedures:** \( T(u, q) = \{ \hat{r} \mid e_q \Rightarrow c \Rightarrow r \Rightarrow \text{nil}, At_u(c) \} \) for \( u \in N, q \in \text{Proc} \).

Obviously we have \( T(u) = T(u, \text{Main}) \) such that it suffices to capture terminating runs in procedure calls in the constraint system. The constraint system is dual to the one for reaching runs:

\[
\begin{align*}
[T1] & \quad T(u, q) \supseteq S^i(u), & \text{if } u \in N_q \\
[T2] & \quad T(u, q) \supseteq T(u, p) \odot S^i(w), & \text{if } (w, u) \in E_q \cap \text{Call}_p \\
[T3] & \quad T(u, q) \supseteq [T(u, p) \odot \text{post}(S(p_{1-i}))] \odot S^i(w), & \text{if } (w, u) \in E_q \cap \text{Pcall}_{p_0 \ldots p_1}
\end{align*}
\]

Again, \( i = 0, 1 \) in the last constraint. The justification of this constraint system is similar to reaching runs; therefore, the details are left to the reader. We should mention, however, that assumption ASS1 is crucial here, like ASS2 in the case of reaching runs, but for a quite different reason. The fundamental difference is the requirement that the configuration \( c \) with \( At_u(c) \) is reachable \((e_q \Rightarrow c)\) in terminating runs, a requirement that has no analogue for reaching runs. As a consequence, \( \text{post}(S(p_{1-i})) \) is now sufficient to capture the interleaving potential in the constraint for parallel calls even in the general case, in contrast to \( \text{pre}(R(p_{1-i})) \) in the corresponding constraint for reaching runs.

However, the reachability requirement for configuration \( c \), implies that some of the constraints are not satisfied by the sets \( T(u, q) \) in the general case. For example, an inverse same-level run \( r \) from a program point \( u \in N_q \) is not always a terminating run. Being an inverse same-level run just means that \( u \Rightarrow \text{nil} \) holds, but for a terminating run we additionally need \( e_q \Rightarrow u \). This is automatically true if ASS1 is valid but can be wrong in the general case. Similarly, we need that the start node of the edge \( e \) in the second and third constraint can be reached for making the constraints valid for the operationally defined sets. A possible remedy is to remove the constraints induced by non-reachable program points. This is detailed in Section 6.5.7.
6.5.6 Bridging Runs

Let \( v \in N \) be a fixed program point. We want to determine the bridging runs \( B_v(u) \) for each \( u \in N \) as defined in Section 6.4. As a first step we capture for each program points \( u \) the runs that reach \( v \), when execution is started directly with \( u \). We call these the simple bridging runs of \( u \) w.r.t. \( v \).

**Simple bridging runs:** \( B_v^s(u) = \{ \hat{r} \mid u \xrightarrow{\hat{r}} c, At_v(c) \} \) for \( u \in N \).

The simple bridging runs can be characterized as the smallest solution of the following constraint system:

\[
\begin{align*}
[\text{BS1}] \quad B^s(v) & \supseteq \{\varepsilon\} \\
[\text{BS2}] \quad B^s(u) & \supseteq \{e\}; B^s(w), \quad \text{if } e = (u, w) \in \text{Base} \\
[\text{BS3}] \quad B^s(u) & \supseteq S(p); B^s(w), \quad \text{if } e = (u, w) \in \text{Call}_p \\
[\text{BS4}] \quad B^s(u) & \supseteq B^s(e_p), \quad \text{if } e = (u, \_\_ ) \in \text{Call}_p \\
[\text{BS5}] \quad B^s(u) & \supseteq [S(p_0) \otimes S(p_1)]; B^s(w), \quad \text{if } e = (u, w) \in \text{Pcall}_{p_0, p_1} \\
[\text{BS6}] \quad B^s(u) & \supseteq B^s(e_{p_i}) \otimes \text{pre}(S(p_{1-i})), \quad \text{if } e = (u, \_\_ ) \in \text{Pcall}_{p_0, p_1}
\end{align*}
\]

The last constraint is again included for \( i = 0, 1 \).

Let us explain why these constraints cover all the ways how \( v \) can be reached from \( u \). If \( u = v \) then there is the trivial way to reach \( v \) from \( u \): by the empty execution; this is covered by Constraint [BS1]. Otherwise, we must proceed via an outgoing edge \((u, w)\) of \( u \). If this is a base edge \( e = (u, w) \), we first see \( e \) and then a run that reaches \( v \) from \( w \); this is covered by Constraint [BS2]. If \( e \) is an edge that calls a procedure \( p \), we distinguish two cases: either \( v \) is reached after \( p \) has terminated—this case is covered by Constraint [BS3]—or \( v \) is reached during the execution of \( p \)—this case is covered by [BS4]. Similarly, if \( e \) is a parallel call of two procedures \( p_0 \) and \( p_1 \), we can reach \( v \) either after both procedures have terminated, which is covered by [BS5]. Or we can reach \( v \) in one of the called procedures \( p_i \). In this case we see a run from \( e_{p_i} \) that reaches \( v \) interleaved with a prefix of a same-level run of procedure \( p_{1-i} \). If assumption ASS2 is violated we must again reckon with procedure \( p_{1-i} \) providing runs that are not prefixes of same-level runs, as was the case for reaching runs. We can solve this problem as for reaching runs, cf. Section 6.5.7.

The reader should face no difficulties in persuading himself, that the \( B^s(u) \) sets indeed solve all constraints.

As a second step we determine the bridging runs in a call to a procedure:

**Bridging runs in procedure calls:**

\[
B_v(u, q) = \{ \hat{r} \mid e_q \Rightarrow c_u \xrightarrow{\hat{r}} c_v, At_v(c), At_v(c) \} \text{ for } u \in N.
\]

Clearly, we have \( B_v(u) = B_v(u, \text{Main}) \) such that we are done, when we have successfully captured \( B_v(u, q) \) for all \( u, q \).
Basically, there are two ways how a bridging run may occur in a call to \( q \). One possibility is that both \( u \) and \( v \) are reached in the same simple or parallel call in \( q \). This case is captured by the following three types of constraints:

[B1] \( B(u, q) \supseteq B(u, p) \), \[ \text{if } e \in E_q \intersects \text{Call}_p \]

[B2] \( B(u, q) \supseteq B(u, p_i) \otimes \text{post}(\text{pre}(S(p_{1-i}))) \), \[ \text{if } e \in E_q \intersects \text{Pcall}_{p_0, p_1} \]

[B3] \( B(u, q) \supseteq \text{pre}(T(u, p_i)) \otimes \text{post}(R(v, p_{1-i})) \), \[ \text{if } e \in E_q \intersects \text{Pcall}_{p_0, p_1} \]

[B2] and [B3] apply for \( i = 0, 1 \).

Constraint [B1] captures the case that \( u \) and \( v \) are reached in the same simple call. Constraint [B2] is concerned with the case that \( u \) and \( v \) are reached in the same procedure \( p_i \) of a parallel call. Before \( u \) is reached in \( p_i \) the other procedure can already perform certain actions and it need not run to completion until \( v \) is reached. Therefore, \( p_{1-i} \) contributes a middle piece of a same-level run. Potential middle pieces can be characterized by \( \text{pre}(\text{post}(S(p_{1-i}))) \) as captured by the second constraint. Constraint [B3] captures the case that \( u \) is reached in procedure \( p_i \) and \( v \) in procedure \( p_{1-i} \). After \( p_i \) has reached \( u \) it can further proceed; specifically \( p_i \) contributes a prefix of a run from \( T(u) \) until \( v \) is reached in \( p_{1-i} \). In order to reach \( v \), \( p_{1-i} \) must execute a run from \( R(v, p_{1-i}) \). It can execute a prefix of this run before \( p_i \) leaves \( u \). Therefore, we see a postfix of a run from \( R(v, p_{1-i}) \) as part of the bridging run.

The second possibility is that \( u \) and \( v \) are not reached in the same simple or parallel call. This gives rise to the following constraints:

[B4] \( B(u, q) \supseteq B^a(u) \), \[ \text{if } u \in N_q \]

[B5] \( B(u, q) \supseteq T(u, p); B^a(w) \), \[ \text{if } (\_, w) \in E_q \intersects \text{Call}_p \]

[B6] \( B(u, q) \supseteq [T(u, p_i) \otimes \text{post}(S(p_{1-i}))]; B^a(w) \), \[ \text{if } (\_, w) \in E_q \intersects \text{Pcall}_{p_0, p_1} \]

where \( i = 0, 1 \) in the last constraint.

The first subcase is that \( u \) is reached on same-level, i.e. in the current instance of \( q \). Then we see a simple bridging run of \( u \) (Constraint [B4]). The second subcase is that \( u \) is reached in a procedure \( p \) called by a simple call edge \( e = (\_, v) \in E_q \). Then we see a run from \( T(u, p) \) followed by a simple bridging run from \( w \) (Constraint [B5]). The third subcase is that \( u \) is reached in a procedure \( p_i \) called by a parallel call edge \( e = (\_, v) \in E_q \). Then we see a run from \( T(u, p_i) \otimes \text{post}(S(p_{1-i})) \) followed by a simple bridging run from \( w \) (Constraint [B6]).

### 6.5.7 The General Case

In this section we describe the changes that are necessary in the general case, i.e., if assumptions ASS1 and ASS2 are potentially violated.

As explained in connection with constraint [R3] one of the problems is that in the general case \( \text{pre}(S(q)) \) does not capture all partial runs of procedure \( q \).
6.5. THE CONSTRAINT SYSTEMS

\[\begin{align*}
&P1 \quad P(q) \supseteq P(e_q) \\
&P2 \quad P(u) \supseteq \{e\} \\
&P3 \quad P(u) \supseteq \{e\}; P(v), \quad \text{if } e = (u, v) \in \text{Base} \\
&P4 \quad P(u) \supseteq P(p), \quad \text{if } (u, \_ ) \in \text{Call}_p \\
&P5 \quad P(u) \supseteq S(p); P(v), \quad \text{if } (u, v) \in \text{Call}_p \\
&P6 \quad P(u) \supseteq [P(p_0) \otimes P(p_1)], \quad \text{if } (u, v) \in \text{Pcall}_{p_0,p_1} \\
&P7 \quad P(u) \supseteq [S(p_0) \otimes S(p_1)]; S^i(v), \quad \text{if } (u, v) \in \text{Pcall}_{p_0,p_1}
\end{align*}\]

Figure 6.2: A constraint system characterizing finite prefixes.

Thus, interleaving \(R(u, p_i)\) with \(\text{pre}(S(p_{1-i}))\) does not capture all possible runs that reach \(u\) in a parallel call. This problem also arises in constraints [BS6] and [B2]. A possible remedy is to introduce new variables \(P(q), q \in \text{Proc}\), that characterize finite prefixes of (finite or infinite) runs, i.e. \(P(q) = \{\hat{r} \mid e_q \xrightarrow{r} c\}\), and to use \(P(p_{1-i})\) instead of \(\text{pre}(S(p_{1-i}))\) in [R3], [BS6], and [B2]. A simple way to calculate \(P(q)\) is to add a constraint of the following form for each procedure \(q\) and program point \(u\) to the constraint system for reaching runs:\(^2\)

\[\begin{align*}
&P \quad P(q) \supseteq R(u, q).
\end{align*}\]

While this way of calculating \(P(q)\) is easy to specify it has the disadvantage of introducing \(|N| \cdot |\text{Proc}|\) new constraints, i.e. quadratically many. Although this does not spoil the overall asymptotic complexity—already the constraint system for reaching runs has \(O(|N| \cdot |\text{Proc}|)\) constraints—we should mention that \(P(q)\) can be calculated also by \(O(|N|)\) constraints. A corresponding constraint system is given in Fig. 6.2. It determines as auxiliary information finite prefixes of (finite or infinite) runs from program points, defined by \(P(u) = \{\hat{r} \mid u \xrightarrow{r} c\}\) by backwards accumulation and is similar to the constraint system for simple bridging runs.

A similar problem arises in constraint [B3]: if assumption ASS2 is violated, \(\text{pre}(T(u, p_i))\) does not necessarily capture all partial runs exhibited by \(p_i\) after reaching \(u\) because \(u\) could be reached at a configuration from which termination

\(^2\)If we are working with a non-atomic interpretation of assignments we must use the following constraint instead of [P]:

\[\begin{align*}
&P' \quad P(q) \supseteq \text{pre}(R(u, q)).
\end{align*}\]

In the atomic interpretation, any configuration \(c\) satisfies \(\text{At}_u(c)\) for at least one program point \(u\). Therefore, the simpler constraint [P] without the pre-operator is sufficient. In the non-atomic interpretation, however, there are (implicitly) transient configurations that correspond to intermediate stages of executions in which no program point is active. Fortunately, from all transient configurations \(c\) a configuration \(c'\) with some active program point is reachable. Therefore, we can capture the runs to transient configurations by means of the pre-operator.
[Q1] \(Q(u, q) \supseteq P(u)\), if \(u \in S_q\)

[Q2] \(Q(u, q) \supseteq Q(u, p)\), if \((v, \omega) \in E_q \cap \text{Call}_p\)

[Q3] \(Q(u, q) \supseteq T(u, p); P(w)\), if \((v, w) \in E_q \cap \text{Call}_p\)

[Q4] \(Q(u, q) \supseteq Q(u, p_i) \otimes \text{post}(P(p_{1-i}))\), if \((v, \omega) \in E_q \cap \text{Call}_{p_0, p_1}\)

[Q5] \(Q(u, q) \supseteq \{T(u, p_i) \otimes \text{post}(S(p_{1-i}))\}; P(w)\), if \((v, w) \in E_q \cap \text{Call}_{p_0, p_1}\)

Figure 6.3: A constraint system for partial runs that can be exhibited in a procedure after a given program point has been reached. All constraints [Q1]-[Q5] are only for program points \(v\) with \(S(v) \neq \emptyset\). In [Q4] and [Q5], \(i = 0, 1\).

is impossible. The information needed in place of \(\text{pre}(T(u, p_i))\) is \(Q(u, p_i)\) where \(Q(u, q) = \{\tilde{r} \mid e_q \Rightarrow c \Rightarrow c', At_u(c)\}\) for \(u \in N, q \in \text{Proc}\). These sets can be characterized by the constraint system in Fig. 6.3

The above changes ensure that the run sets characterized by the constraint systems are sufficiently large. They are necessary to make flow analysis based on abstract interpretation of the constraint systems sound. The changes described now ensure that the run sets do not become too large. Thus, they are necessary to make analyses based on a precise abstract interpretation complete.

As explained in connection with terminating runs, constraints induced by unreachable program points are not satisfied by the run sets (defined from the operational semantics) that we intend to characterize. As these constraints pose unnecessary additional requirements they make the solutions larger than necessary. Fortunately, such constraints are also unnecessary for soundness and can simply be removed. Specifically, we must include the constraints [T1], [B1], and [B4] only for program points \(u\) with \(S(u) \neq \emptyset\), and the constraints [T2], [T3], [B2], [B3], [B5], and [B6] only for edges \(e = (v, w)\) with \(S(v) \neq \emptyset\). We have seen in Section 6.5.3, that we can determine this information with a very simple and cheap analysis.

With the changes described in this section we obtain constraint systems that are both sound and complete in the general case.

### 6.6 Discussion

In this chapter we have introduced parallel flow graphs. After that we defined a symbolic operational semantics. It works on configurations that take the form of a tree, the nodes of which are annotated by program points. Intuitively, such a tree models a generalization of a run time stack that may branch to parallel stacks in addition to the common stack operations. Branching is crucial to model
6.6. DISCUSSION

parallel calls. We have described the transitions of the operational semantics by rules that work directly on configurations of this form.

There are obvious alternatives to this way of describing the operational semantics: in particular, we could have used the approach chosen by Esparza, Knoop, and Podelski in their work on flow analysis of parallel programs [16, 17]. They map a parallel flow graph to a so-called PA-processes; PA is a process algebra which has both a sequential and a concurrent composition operator [5, 43]. Execution of PA-processes in turn is described by a structured operational semantics (SOS) [61]. In this way they could apply results about model-checking of PA-processes to flow analysis. For our purposes the approach chosen here is sufficient and produces less notational overhead.

Based on the operational semantics we have defined a number of run sets of particular interest and have then developed constraint systems that characterize these run sets. The constraint systems for same-level runs and reaching runs are essentially the ones used by Seidl and Steffen [71]. Also the constraint systems for inverse same-level runs and terminating runs are indicated in their work. The constraint system for bridging runs, however, is completely new. A further difference is that Seidl and Steffen postulate their constraint systems, while we use an operational semantics as a reference point. While this might be considered a minor or even trivial difference, in our opinion an operational justification of the constraint systems largely increases our understanding of what exactly is specified by the constraint systems.

Many reasonable variants of the run sets in question may be considered. For example, one could define reaching runs by

$$R'(u) = \{ \hat{r} \mid e_{\text{Main}} \xrightarrow{r} c \implies \varepsilon, At_u(c) \}.$$  

This definition deviates from the standard definition in that it considers only configurations $c$ from which termination is possible, i.e., it characterizes the runs that both reach $u$ and can be completed to a terminating run. In general, if assumption ASS2 is violated, this definition gives rise to smaller run sets than the standard definition. It might be preferable, if one is interested in terminating runs of programs only, like in total correctness reasoning. Similarly, many reasonable variants of the other run sets are conceivable and by techniques similar to the ones of Section 6.5.7 sound and complete constraint systems for these variants can be constructed. Operational specifications of the run sets in question allows to distinguish these variants much more clearly than implicit specifications by means of constraint systems.

Validating constraint systems with respect to an operational semantics has another advantage: it helps to uncover subtle bugs. In the absence of an operational semantics, Seidl and Steffen, for instance, fail to notice that constraint [R3] in the constraint system for reaching runs is not rich enough to characterize all reaching runs in a parallel composition if assumption ASS2 does not hold. We
detected this error while trying to justify the soundness of the constraint system. As a consequence their constraint system for reaching runs is unsound in the general case. To be fair, we should note that this does not affect the soundness of their analysis procedure that is not directly based on the constraint system for reaching runs. We should also say that they solve the problems that arise when assumption ASS1 is violated correctly. Here they validly propose to remove edges leaving unreachable program points before the analysis. This has essentially the same effect as the side conditions of the form \( S(u) \neq \emptyset \) added to the various constraints in Section 6.5.7.
Chapter 7

Non-Atomic Execution

The idealization that assignments execute atomically is quite common in the literature on program verification as well as in the theoretical literature on flow analysis of parallel programs. However, in a multi-processor environment where a number of concurrently executing processors share a common memory this assumption is hardly realistic. In such an environment two threads of control may well interfere while each of them is in the process of executing an assignment. The reason is that assignments are broken into smaller instructions before execution.

As a simple example, consider a program consisting of two parallel assignments both incrementing a shared variable $x$:

$$x := x + 1 \parallel x := x + 1.$$

Let us assume that $x$ holds 0 initially. If assignments execute atomically, this program clearly will increment $x$ twice and so terminate in a state in which variable $x$ holds 2. However, in a multi-processor environment this program may well set $x$ to 1. For example, the following execution may happen: first, one of the processors accesses the memory in order to get the value of $x$. While it is in the process of incrementing this value, but before it has written back the result, the second processors may access the memory, too, in order to get the value of $x$. In such a run, both processors read the initial value 0 for $x$, both will increment just this value, and both will write back 1 for $x$. Consequently, the program will terminate in a state where $x$ holds 1 instead of 2.

In order to be more specific and, at the same time, keep the discussion simple, let us assume that the processors are stack machines. Then a compiler might generate the following piece of code for the assignment $x := x + 1$:

1. PUSH $x$
2. PUSH 1
3. ADD
4. POP $x$

105
Using unprimed numbers for the statements of the first processor and primed numbers for the statements of the second one, the two processors may then, e.g., execute their instructions in the following order:

\[ 1, 2, 1', 2', 3', 4', 3, 4. \]

We leave it to the reader to check that this execution indeed increments \( x \) just by 1.

The morale of this discussion is that, in the real world of multi-processor execution, we cannot assume atomic execution of assignments. What we typically \textit{may} safely assume, however, is that single reads of variables and single writes of variables are atomic, because the access to the memory is usually synchronized, e.g., through a common bus.

This said, we should mention that there are indeed execution scenarios for concurrent programs that guarantee atomic execution of assignments. In particular in a time-shared multi-tasking environment, where concurrent execution of threads is simulated by a single processor that switches between execution of code pieces implementing the different threads, assuming atomic execution of assignments may be safe, if context switches happen only between assignments, but not in the process of executing the code implementing a single assignment. The built-in scheduler of the Transputer, for instance, performs context switches only after certain types of instructions that typically end execution of assignment code [30].

Note how non-atomic execution of assignments was modeled in the above example: first each assignment was broken into the smaller instructions of the stack machine; each of these instructions may be considered as an atomic unit of execution. Then the two threads 1, 2, 3, 4 and 1', 2', 3', 4' of more fine-grained stack machine instructions was interleaved. This example tells us that we can develop an interleaving semantics for parallel programs that adequately models non-atomic execution of assignments by means of breaking assignments into more fine-grained atomic actions, an observation that is exploited in a moment.

The purpose of this chapter is to provide parallel flow graphs with an interleaving semantics that models non-atomic execution of assignments adequately. For this purpose we define a domain \( \text{NR} \) of sets of (non-atomic) runs and provide adequate definitions for the constants and operators used in the constraint systems in Section 6.5. Specifically, we provide

\[ \forall \text{edge } e, \exists \text{ run } r \in \text{NR} \] for the non-atomic runs of a base edge; and

\footnote{The Transputer designers have chosen this strategy in order to make context switches cheap and fast. In typical code, the contents of certain registers used for expression evaluation is no longer needed after such instructions. Therefore, these registers are not stored during context switches, which makes context switches fast. Actually, it is the compiler writer's task to ensure that the generated code does not rely on the registers keeping their contents after such instructions. Atomic execution of assignments in typical code is a neat side-effect of this design.}
7.1 Modeling Non-Atomic Execution by Virtual Variables

Suppose given a parallel flow graph and let $X$ be the set of program variables which the statements of the flow graph refer to. In order to explain the meaning of non-atomic statements appropriately suppose furthermore given an infinite set $V$ of virtual (or internal) variables disjoint from $X$. Intuitively, virtual variables are used to store intermediate results that are private to the threads. The parallel composition (or interleaving) operator defined later ensures that parallel threads do not interfere on virtual variables. We use the letters $x, y$ to range over $X$, $u, v$ to range over $V$, and the letters $a, b$ to range over $X \cup V$.

For the purpose of the semantics, assignments are split into atomic operations. As an example consider an assignment statement $x := e(y_1, \ldots, y_k)$ in the program; $x, y_1, \ldots, y_k$ are program variables. There are many sensible atomicity assumptions. For example, we could work with the rather pessimistic assumption that just reads and writes of variables are atomic, then $x := e(y_1, \ldots, y_k)$ is replaced by a sequence of assignments

$$v_{\pi(1)} := y_{\pi(1)}; \ldots; v_{\pi(k)} := y_{\pi(k)}; x := e(v_1, \ldots, v_k),$$

where $v_1, \ldots, v_k$ are arbitrary distinct virtual variables and $\pi$ is a permutation of $\{1, \ldots, k\}$. The idea is that the other threads can execute atomic operations between these assignments.

More coarse-granular atomicity assumptions can be captured in a similar way. If we assume, for instance, that the evaluation of the right-hand-side expressions is atomic then we would replace $x := e(y_1, \ldots, y_k)$ by

$$v := e(y_1, \ldots, y_k); x := v.$$ 

The important observation is the following: whatever the specific atomicity assumption may be, if we assume that the execution of all assignments is non-atomic, then all assignments in a run that refer to a program variable on the left hand side have only virtual variables on the right hand side. Thus, all assignments belong to the set

$$\text{Asg} = \{ a := e(b_1, \ldots, b_k) \mid a \in X \Rightarrow b_1, \ldots, b_k \in L \}.$$
CHAPTER 7. NON-ATOMIC EXECUTION

One way of obtaining a semantics for non-atomically executing assignments is to transform the assignments in the program prior to semantic interpretation. As an example consider the program in Fig. 7.1(a) which corresponds to the example discussed in the introduction. We could transform it to the program in Fig. 7.1(b) and then apply the standard interpretation.

The problem with this approach is that we must be careful not to confuse virtual variables of different threads. This is simple if only instances of different procedures run in parallel: then we can simply use different names for the virtual variables in different procedures. However, it becomes problematic if different instances of the same procedure may run in parallel like in the program in Fig. 7.2. Then we must model the virtual variables by local variables of the procedures which is not supported by the flow-graph model developed up to now. Therefore, we are using a different approach. We do not transform flow graphs but incorporate the transformation implicitly into the semantic interpretation of assignments.

Before we turn to the technical details of the new semantic interpretation we revisit the example from Section 5.1 in order to show that it makes a difference for constant detection whether base statements are assumed to execute atomically or not. This example illustrates that the main mechanism underlying the undecidability proof of interprocedural parallel constant detection from Chapter 5 does not carry over to the non-atomic case.
7.2 A Motivating Example

Consider again the following program for which a control flow graph-like representation is shown in Figure 7.3 (a):

\[ a := 1; [(b := a; b := 0; c := 0; \text{write}(c)) \| c := b]. \]

Assume first that assignment statements execute atomically. From Section 5.1 we know that under this assumption variable \( c \) is a (copy) constant of value 0 at the write instruction. Let us briefly recall the underlying reasoning. In any execution \( c := 0 \) must be executed either after or before \( c := b \) in the parallel thread. If it is executed after \( c := b \) then \( c \) holds 0 at the write statement because 0 is assigned to \( c \) in the last executed assignment, \( c := 0 \). On the other hand, if \( c := 0 \) is executed before \( c := b \) then also the initialization of \( b, b := 0 \), must have been executed before \( c := b \) such that \( c := b \) also loads the value 0 to \( c \).

The situation is dramatically different if assignment statements may execute non-atomically. In particular, if the assignment \( c := b \) in the second thread is executed non-atomically, the first thread may execute the two statements \( b := 0 \) and \( c := 0 \) that kill \( b \) and \( c \) after \( b \) is loaded from the shared memory but before the loaded value is stored to \( c \). This results in a run of the program that propagates the value 1 from the initialization \( a := 1 \) to the final write-statement.

As explained in the previous section, we may model the two stage non-atomic execution of \( c := b \) by splitting it into two assignments \( v := b \) and \( c := v \), where \( v \) is a new virtual variable that cannot be accessed by the first thread (cf. Figure 7.3 (b)). We can think of virtual variable \( v \) as representing the register in which the
value loaded from the common memory is stored. This register is private to the second
thread and therefore there can be no interference on this variable. As there is no
interference on virtual variable \( v \), we can consider each of the virtual
assignments \( v := b \) and \( c := v \) to be atomic. The resulting program has the run
\[
\begin{align*}
\tau = & \langle a := 1, b := a, v := b, b := 0, c := 0, c := v, \text{write}(c) \rangle.
\end{align*}
\]
which—as the reader can easily verify—propagates the value 1 from the initial-
ization \( a := 1 \) to the write-statement. Thus, run \( \tau \) witnesses that \( c \) is not a copy
constant at the write statement, in sharp contrast to the state of affairs under
the assumption that assignments execute atomically.

7.3 The Domain of Non-Atomic Run Sets

A (non-atomic) run \( \tau \) is a sequence of assignments from the set \( \text{Asg} \) defined above:
\( \text{Runs} = \text{Asg}^\ast \). We write \( \text{virtual}(\tau) \) for the set of virtual variables appearing in run
\( \tau \) and denote the empty run by \( \varepsilon \) and the concatenation operator by an infix dot
or just by juxtaposition.

As the specific choice of virtual variables is immaterial, we assume that all
considered sets of runs are closed under bounded renaming of virtual variables.
This allows a simple and adequate definition of the composition operators. In
order to allow a technically clean treatment of this assumption, let \( \equiv \subseteq \text{Runs} \times
\text{Runs} \) be the equality of runs up to bounded renaming of virtual variables, i.e.
\( \tau \equiv \tau' \) hold if and only if \( \tau' \) can be obtained from \( \tau \) by bounded renaming of
virtual variables.

\textbf{Proposition 7.1} \( \equiv \) is an equivalence. \hfill \Box

For a set of runs \( \mathcal{R} \subseteq \text{Runs} \) we write \( \mathcal{R}^\equiv \) for the closure of \( \mathcal{R} \) w.r.t. \( \equiv \):
\[
\mathcal{R}^\equiv = \{ \tau \in \text{Runs} \mid \exists \tau' \in \mathcal{R} : \tau \equiv \tau' \}.
\]

Obviously, this defines a closure operator.

\textbf{Proposition 7.2}

1. \( \mathcal{R} \subseteq \mathcal{R}^\equiv \).

2. \( (\mathcal{R}^\equiv)^\equiv = \mathcal{R}^\equiv \).

3. \( \mathcal{R} \subseteq \mathcal{S} \) implies \( \mathcal{R}^\equiv \subseteq \mathcal{S}^\equiv \). \hfill \Box

The domain \( \text{NR} \) is given by the sets of runs that are closed under \( \equiv \):
\[
\text{NR} = \{ \mathcal{R} \subseteq \text{Runs} \mid \mathcal{R} = \mathcal{R}^\equiv \}.
\]

The members of \( \text{NR} \) model sets of runs in a scenario where assignments execute
non-atomically.

\textbf{Lemma 7.3} \( (\text{NR}, \subseteq) \) is a complete lattice with least element \( \bot_{\text{NR}} = \emptyset \) and greatest
element \( \top_{\text{NR}} = \text{Runs} \).
7.3. THE DOMAIN OF NON-ATOMIC RUN SETS

Proof. \((NR, \subseteq)\) is a sub-lattice of the power set lattice \((2^{R_{\text{Runs}}}, \subseteq)\). To show this, we have to check, that \(NR\) is closed under arbitrary intersections and unions.

Here is the proof for intersection. Suppose \(\mathcal{R} \subseteq NR\) and \(r, r' \in R_{\text{uns}}\) with \(r \equiv r'\). We have to show that \(r \in \bigcap \mathcal{R}\) if and only if \(r' \in \bigcap \mathcal{R}\) which is simple:

\[
\begin{align*}
  r & \in \bigcap \mathcal{R} \\
  \text{iff} & \quad \text{[Definition of } \bigcap \mathcal{R}] \\
  \forall R & \in \mathcal{R}: r \in R \\
  \text{iff} & \quad [\mathcal{R} \subseteq NR, \text{hence all } R \in \mathcal{R} \text{ are closed under } \equiv] \\
  \forall R & \in \mathcal{R}: r' \in R \\
  \text{iff} & \quad \text{[Definition of } \bigcap \mathcal{R}] \\
  r' & \in \bigcap \mathcal{R}.
\end{align*}
\]

The proof for unions is just as simple and, therefore, omitted.

The least and greatest element of \((R_{\text{uns}}, \subseteq)\) are \(\emptyset\) and \(R_{\text{uns}}\), respectively. It is obvious that both of them are closed under \(\equiv\) and hence are also the least and greatest elements, respectively, of \((NR, \subseteq)\).

In the sections that follow we provide definitions for the operators and constants appearing in the constraint systems and show their well-definedness.

7.3.1 Base Statements

We can work with various atomicity assumptions as discussed above. The most natural and conservative one is that just single reads and writes of variables are atomic. This is captured by defining the semantics of an assignment statement, \([x := e] \in NR\), where \(y_1, \ldots, y_k\) are the variables appearing in \(e\), as the set of runs of the form

\[
\langle v_{\pi(1)} := y_{\pi(1)}, \ldots, v_{\pi(k)} := y_{\pi(k)}, x := e(v_1, \ldots, v_k) \rangle,
\]

where \(\pi\) is a permutation of \(\{1, \ldots, k\}\) and \(v_1, \ldots, v_k\) are arbitrary distinct virtual variables. It is readily verified that \([x := e]\) is well-defined, i.e., that \([x := e] \in NR\). We have to show that \([x := e]\) is closed under \(\equiv\) which is obvious as we admitted an arbitrary choice of virtual variables.

We may also work with a more coarse-grained semantics of assignments. For our purposes the choice is arbitrary, as the dependence trace abstraction of an assignment will be precise with respect to any of these definitions.

Obviously, the only non-atomic run of statement \(\text{skip}\) is the empty run. Hence, \([\text{skip}] = \{\varepsilon\}\). Obviously, \([\text{skip}] \in NR\).
The non-atomic runs induced by a base edge \( e \in \text{Base} \) are the non-atomic runs of the statement associated with \( e \): \( [e] = [A(e)] \), where \( A(e) \) is the base statement associated with base edge \( e \) in the underlying flow graph.

### 7.3.2 Sequential Composition

The \textit{sequential composition operator}, \( \cdot ; \cdot : \text{NR} \times \text{NR} \rightarrow \text{NR} \), which is written as an infix operator, is defined by

\[
R ; S = \{ r \cdot s \mid r \in R, s \in S, \text{virtual}(r) \cap \text{virtual}(s) = \emptyset \}^= .
\]

Recall that \( \cdot \) denotes concatenation of sequences. The condition about the local variables ensures that runs composed sequentially do not interact on local variables. It could be replaced by a condition that in a run all local variables are initialized before they are used. However, the latter condition would not be preserved by the pre-operator and, therefore, we prefer the chosen solution. The outer closure operator ensures that \( ; \) is well-defined.

### 7.3.3 Interleaving Operator

In order to define the interleaving (or parallel composition) operator some notation is needed. Let \( r = \langle e_1, \ldots, e_n \rangle \) be a sequence and \( I = \{ i_1, \ldots, i_k \} \) a subset of positions in \( r \) such that \( 1 \leq i_1 < i_2 < \cdots < i_k \leq n \). Then \( r|I \) is the sequence \( \langle e_{i_1}, \ldots, e_{i_k} \rangle \). We write \( |r| \) for the length of \( r \), viz. \( n \). The \textit{interleaving operator}, \( \otimes : \text{NR} \times \text{NR} \rightarrow \text{NR} \), which we write in an infix form, is defined by

\[
R \otimes S = \{ r \mid \exists I_R, I_S : I_R \cup I_S = \{ 1, \ldots, |r| \}, I_R \cap I_S = \emptyset, r|I_R \in R, r|I_S \in S, \text{virtual}(r|I_R) \cap \text{virtual}(r|I_S) = \emptyset \}^= .
\]

The condition about the local variables in \( r|I_R \) and \( r|I_S \) ensures that parallel threads do not exchange values via local variables. The application of the closure operator \( (\cdot)^= \) guarantees well-definedness: \( R \otimes S \in \text{NR} \) for \( R, S \in \text{NR} \).

Suppose \( r, s, t \in \text{Runs} \) with \( \text{virtual}(r) \cap \text{virtual}(s) = \emptyset \). We call \( t \) an \textit{interleaving of} \( r \) and \( s \) if

\[
\exists I_r, I_s : I_r \cup I_s = \{ 1, \ldots, |r| \}, I_r \cap I_s = \emptyset, t|I_r = r, t|I_s = s
\]

and denote the set of interleavings of \( r \) and \( s \) by \( r \otimes s \).

### 7.3.4 Pre-Operator

The pre-operator, \( \pre : \text{NR} \rightarrow \text{NR} \) is defined as follows:

\[
\pre(R) = \{ r \in \text{Runs} \mid \exists r' \in \text{Runs} : r \cdot r' \in R \} .
\]

**Lemma 7.4** \( \pre \) is well-defined.
7.4. CONCLUSION

**Proof.** We have to show that, for any \( R \in \text{NR} \), \( \text{pre}(R) \) is closed under \( \equiv \). So suppose given \( r, s \in \text{Runs} \) with \( s \equiv r \in \text{pre}(R) \). Then there is \( r' \in \text{Runs} \) with \( r \cdot r' \in R \). By bounded renaming of local variables in \( r' \) we can construct a run \( s' \) such that \( s \cdot s' \equiv r \cdot r' \). As \( R \) is closed under \( \equiv \), \( s \cdot s' \in R \) and hence \( s \in \text{pre}(R) \). □

7.3.5 Post-Operator

Analogously to the pre-operator, the post operator \( \text{post} : \text{NR} \to \text{NR} \) is defined as follows:

\[
\text{post}(R) = \{ r \in \text{Runs} \mid \exists r' \in \text{Runs} : r' \cdot r \in R \}.
\]

**Lemma 7.5** \( \text{post} \) is well-defined. □

7.4 Conclusion

We have defined a complete lattice \((\text{NR}, \subseteq)\) the members of which model sets of runs in a scenario in which assignment statements execute non-atomically. In order to enable an interleaving semantics to adequately capture the effect of non-atomic execution of assignments, we resorted to *virtual variables* that model storage locations that are private to threads.

The members of \( \text{NR} \) are those sets of runs that are bounded under renaming of virtual variables. We have provided definitions for the operators and constants appearing in the constraint systems that capture reaching, terminating, and bridging runs in a parallel flow graph. The (smallest) solution of these constraint systems over this new interpretation induces a new semantics of parallel flow graphs that captures non-atomic execution of assignments. The new semantics provides a reference point for assessing flow analyses that are performed by means of an abstract interpretation of the constraint systems. We will put this idea to advantage in Chapter 8 where we show that the dependence trace interpretation developed there is a precise abstraction of the non-atomic interpretation of parallel flow graphs.
Chapter 8
Dependence Traces

We can indirectly detect copy constants and eliminate faint code on the basis of
the following information: given a program point \( u \) and a variable \( x \) of interest;
when control is at another program point \( v \), which variables \( y \) may influence
the value of \( x \) at \( u \)? This information is the abstraction of \( u-v \)-runs (bridging
runs) to the set of dependences mediated by these runs. Here we mean by a
dependence mediated by a run \( r \) a pair of variables \((x, y)\) such that the value of
\( y \) after execution of \( r \) depends on the initial value of \( x \) (cf. Section 8.1 for the
formal definition).

In Section 6.5 a constraint system characterizing the set of bridging runs of
a given parallel flow graph was given. We would like to perform the analysis by
evaluating this constraint system over an abstract interpretation. Unfortunately,
we cannot use dependences themselves as abstract domain because, in general,
we cannot obtain the dependences of a parallel composition of run sets from
the dependences of the components (cf. Section 8.2). Therefore, an abstraction
employing just dependences cannot be sound and complete at the same time. We
need to collect more information in the abstract domain.

In this chapter, we introduce an adequate abstract domain from which on
the one hand dependences can be inferred easily and for which on the other
hand abstract operation can be defined that mirror precisely the corresponding
operations on sets of (non-atomic) runs.

The basic idea is to collect not only dependences but sequences of dependences
(dependence sequences) that can successively be mediated by a run. For example,
the run \( r_1 = \langle c := b, e := d \rangle \) has \( \langle b, c), (d, e) \rangle \) as one of its dependence sequences.
This dependence sequence plays a dual role: on the one hand, it captures the
potential of \( r_1 \) to mediate the dependence \( (b, e) \) if a parallel run fills the gap
between \( c \) and \( d \) (like, e.g., \( r_2 = \langle d := c \rangle \)) and, on the other hand, its potential
to successively fill the gaps \( (b, c) \) and \( (d, e) \) in a parallel run (like, e.g., in \( r_3 = \langle b := a, d := c, f := e \rangle \).

Further information must be collected. To see why, compare the run \( r_4 = \langle b := 0, c := b, e := d, e := 0 \rangle \) to \( r_1 \). Unlike \( r_1, r_4 \) does not have the potential to
mediate the dependence \((b, e)\) if a parallel run fills the gap between \(c\) and \(d\), but it is still able to successively fill the gaps \((b, c)\) and \((d, e)\) in a parallel run. The difference is that in \(r_4\), unlike in \(r_1\), the part of the run before \(b\) is read and after \(e\) is written is not transparent for \(b\) and \(e\), respectively. Therefore, we refine dependence sequences to *dependence traces* in which we record in addition to a dependence trace by two Boolean values, whether the parts of the run before the source variable of the first dependence is read and after the target variable of the final dependence is written are transparent for these variables. Run \(r_1\) for instance has the dependence trace \((1, ((b, c), (d, e)), 1)\) which \(r_4\) has not, but both share the dependence trace \((0, ((b, c), (d, e)), 0)\). In order to allow a proper propagation of transparency information in sequential composition, we furthermore collect in the abstraction the set of variables for which a transparent run exists.

According to these ideas, we can abstract a set of (non-atomic) runs \(R\) to a pair \((T_R, D_R)\) consisting of the set of variables

\[
T_R = \{ x \mid \exists r \in R : r \text{ is transparent for } x \}
\]

and the set of dependence traces

\[
D_R = \{ \tau \mid \exists r \in R : \tau \text{ is a dependence trace of } r \}.
\]

On this abstraction of run sets, we can indeed define abstract operators that precisely mirror the operators on sets of non-atomic runs that are used in the constraint systems of Section 6.5. However, we are not yet done. The problem is, that this abstract domain is not effective, because \(D_R\) can be infinite. In order to obtain an effective domain, we have to go one step further.

For this purpose, we define a subsumption order, written \(\sqsubseteq\), on transition traces. The intuition is that a transition trace \(\tau\) is subsumed by another transition trace \(\tau'\) if \(\tau'\) has fewer gaps than \(\tau\)—we write this as \(\tau \sqsubseteq \tau'\) (cf. Section 8.4). Intuitively, \(\tau'\) is more useful than \(\tau\) in forming dependences. We then collect for a run set only the transition traces that are maximal with respect to the order \(\sqsubseteq\). This set forms an antichain with respect to \(\sqsubseteq\). It is not hard to show that all \(\sqsubseteq\)-maximal dependence traces of a run set are short in a certain sense made precise in Section 8.6. As there are only finitely many short dependence traces this makes the abstract domain finite, such that we can effectively perform fixpoint calculations.

Summarizing, the abstract domain consists of pairs \((D, T)\) where \(D\) is an \(\sqsubseteq\)-antichain of short dependence traces and \(T\) is a set of variables. It is not hard to define on this domain abstract counterparts to the sequential composition operator and to the pre- and post-operator on run sets and to show that these abstract operators are precise abstractions of the concrete ones. It is also straightforward to abstract the run sets associated with base edges precisely.

The interleaving operator, however, poses some complication. The natural way to compose two transition traces \(\tau\) and \(\tau'\) concurrently is to use \(\tau\) to fill gaps
8.1. TRANSPARENCY AND DEPENDENCES

in \( \tau' \) and vice versa. This was our motivation for considering transition traces in the first place; a precise formalization of this idea is given through the relation \( C \) in Section 8.11.1. However, if \( \tau'' \) is a transition trace obtained in this way from a transition trace \( \tau \) of a run \( r \) and a transition trace \( \tau' \) of a run \( r' \), it is not obvious that there is always a run constructed by interleaving \( r \) and \( r' \) that has \( \tau'' \) as one of its transition traces. Otherwise the abstraction would be imprecise. Indeed this would fail for run sets deriving from an atomic interpretation of base statements but we can show this for run sets deriving from a non-atomic interpretation of base statements.

On the other hand, short transition traces can be obtained from non-short ones in this way. There is thus some reason to suspect that we cannot obtain all \( \sqsubseteq \)-maximal dependence traces of the interleaving \( R_1 \otimes R_2 \) of two run sets from the \( \sqsubseteq \)-maximal dependence traces of the components. This would make the abstract operator unsound. Fortunately, we can show that this is not the case. The main insight is covered by a shortening lemma, Lemma 8.38.

As an auxiliary notion we introduce a further order on dependence traces, called the \textit{implication order} which is written as \( \leq \). Its name is justified by the fact that any run \( r \) that has \( \tau \) as a dependence trace also has \( \tau' \) as a dependence trace, if \( \tau \leq \tau' \). Therefore, the implication order captures implied knowledge about dependence traces of runs, hence its name. The implication order is crucial in particular for a concise formulation of the shortening lemma mentioned above.

In the remainder of this chapter we elaborate these topics in detail.

8.1 Transparency and Dependences

A run \( r \) is called \textit{transparent} for a variables \( a \) if it does not contain an assignment with \( a \) as left hand side variable. Thus, a run is transparent for \( a \) if its execution is guaranteed not to change the value held by \( a \).

\textbf{Example 8.1} The run \( \langle a := 0, b := c \rangle \) is transparent for all variables except of \( a \) and \( b \), in particular for \( c \). \hfill \( \Box \)

A \textit{dependence} is a pair \( d = (x, y) \) of program variables \( x, y \in X \). We call \( x \) the source variable and \( y \) the destination variable of \( d \). A run \( r \) is said to \textit{mediate the dependence} \( (x, y) \), if there are variables \( a_0, \ldots, a_l \), \( l > 0 \), expressions \( e_1, \ldots, e_l \), and (sub-) runs \( r_0, \ldots, r_l \) such that

1. \( r = r_0 \cdot \langle a_1 := e_1 \rangle \cdot r_1 \cdot \langle a_2 := e_2 \rangle \cdot r_2 \cdot \ldots \cdot \langle a_l := e_l \rangle \cdot r_l; \)
2. \( a_0 = x, a_l = y; \)
3. \( e_i \) contains \( a_{i-1} \) for \( i = 1, \ldots, l \); and
4. \( r_i \) is transparent for \( a_i \) for \( i = 0, \ldots, l \).
Example 8.2 The run \( \langle b := 0, b := a, c := b, e := 0, f := e \rangle \) mediates the dependencies \((a, b), (a, c), \) and \((b, c)\) but not the dependence \((e, f)\) because \(e\) is killed by the assignment \(e := 0\) before it is read. \(\square\)

8.2 Dependence Traces

In general, the dependencies of the interleaving \(R_1 \otimes R_2\) of two sets of runs cannot directly be inferred from the dependencies of the component run sets \(R_1\) and \(R_2\). As an example, consider the two sets of runs \(R_1 := \{ \langle b := a, d := c \rangle \}\) and \(R_2 := \{ \langle b := a, d := c \rangle \}\). Both mediate just the dependencies \((a, b)\) and \((c, d)\). But the interleaving of \(R_1\) with \(R_3 := \{ \langle c := b \rangle \}\) contains the run \(\langle b := a, c := b, d := c \rangle\) that mediates the dependence \((a, d)\) while there is no run in the interleaving of \(R_2\) and \(R_3\) that mediates this dependence.

Thus, an abstraction of sets of runs that faithfully mirrors dependencies must collect more information than just dependences. We propose to employ dependence traces that are defined in the remainder of this section.

The basic idea is to collect not just dependences but sequences of dependences that can successively be mediated by a run. For example, we would record the sequences \(\varphi = \langle (a, b), (c, d) \rangle\) for the run \(r_1 = \langle b := a, d := c \rangle\) from \(R_1\) but not for \(R_2\). Intuitively, \(\varphi\) shows us that \(r_1\) could mediate a dependence from \(a\) to \(d\) if a parallel component fills the gap from \(b\) to \(c\). Dually, it also indicates that \(r_1\) can successively fill the gaps \((a, b)\) and \((c, d)\).

A dependence sequence is a sequence \(\varphi = \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle\), \(k \geq 0\), of dependences. We also allow the empty dependence sequence \(\varepsilon\). This mostly smoothens the exposition that follows but sometimes requires a special treatment. We write \(\bar{\varphi}\) for \(x_1\) and \(\bar{\varphi}\) for \(y_k\), if \(\varphi \neq \varepsilon\); if \(\varphi = \varepsilon\), \(\bar{\varphi}\) and \(\bar{\varphi}\) are undefined. We denote the set of transfer sequences by \(\text{DS}\).

Example 8.3 \(\varphi = \langle (a, b), (x, y) \rangle\) is a dependence sequence with \(\bar{\varphi} = a\) and \(\bar{\varphi} = y\). \(\square\)

As explained in the introduction to this chapter, we must distinguish between runs like \(r_1\) above and runs like \(r'_1 := \langle a := 0, b := a, d := c, d := 0 \rangle\) by means of initial and final transparency bits. Unlike \(r_1\), \(r'_1\) does not have the potential to mediate the dependence \((a, d)\) if the gap \((b, c)\) is filled by a parallel run but like \(r_1\) it can successively fill the gaps \((a, b)\) and \((c, d)\).

A dependence trace is a triple \(\tau = (\iota, \varphi, \kappa)\) consisting of Boolean values \(\iota, \kappa \in \mathbb{B} = \{0, 1\}\) coding initial and final transparency and a dependence sequence \(\varphi\). We assume that \(\iota = 0\) and \(\kappa = 0\) if \(\varphi = \varepsilon\). The set of dependence traces is denoted by \(\text{DT}\):

\[
\text{DT} = \{(\iota, \varphi, \kappa) \in \mathbb{B} \times \text{DS} \times \mathbb{B} \mid \varphi = \varepsilon \Rightarrow (\iota = 0 \land \kappa = 0)\}.
\]
8.2. DEPENDENCE TRACES

The dependence trace \( \tau = (\iota, ((x_1, y_1), \ldots, (x_k, y_k)), \kappa) \) is called compatible with a run \( r \), \( r \vdash \tau \) for short, if there are sub-runs \( t_0, \ldots, t_k, r_1, \ldots, r_k \), such that

1. \( r = t_0 r_1 t_2 \cdots r_k t_k \);

2. \( r_i \) mediates the dependence \( (x_i, y_i) \) for \( i = 1, \ldots, k \);

3. \( \iota = 1 \) implies that \( t_0 \) is transparent for \( x_1 \); and

4. \( \kappa = 1 \) implies that \( t_k \) is transparent for \( y_k \).

In this case, we call \( t_0 r_1 t_2 \cdots r_k t_k \) a decomposition of \( r \) that witnesses \( r \vdash \tau \). Note that \( r \vdash (0, \varepsilon, 0) \) holds for all runs \( r \) as witnessed by the trivial decomposition \( r = t_0 \). The trivial dependence trace \( (0, \varepsilon, 0) \) allows us to distinguish the dependence trace abstraction of an empty run set from the abstraction of a run sets without interesting dependence traces.

Instead of saying “\( \tau \) is compatible with \( r \)” we often use the phrase “\( \tau \) is a dependence trace of \( r \)”.

**Example 8.4** Consider the run \( r = (a := 0, b := a, c := b, c := 0, f := e, e := 0) \). One of the dependence traces of \( r \) is \( \tau = (0, (a, c), (e, f)), 1) \) as witnessed by the decomposition \( r = t_0 r_1 t_2 t_2 \) where

\[
g := 0, b := a, c := b, c := 0, f := e, e := 0.
\]

Another decomposition witnessing \( \tau \) is

\[
g := 0, b := a, c := b, \quad c := 0, f := e, e := 0.
\]

The run \( r \) has also many other dependence traces, e.g., \( (1, (b, c), (e, f)), 1) \) and \( (1, (e, f)), 1) \).

Ultimately, we are interested in dependence traces without gaps that code complete transfers from one variable to another one, where a gap can either be a lack of initial or final transparency or a hole from \( y_i \) to \( x_{i+1} \). The other dependence traces are needed only to compute these perfect dependence traces in a compositional fashion. Thus, the dependence traces of ultimate interest are those of the form \( (1, (x, y), 1) \). They correspond to dependences.

**Proposition 8.5** \( r \vdash (1, (x, y), 1) \) if and only if \( r \) mediates the dependence \( (x, y) \).
We can abstract a set $R$ of runs to the set $D_R := \{ \tau \mid \exists r \in R : r \vdash \tau \}$ of compatible dependence traces and it is possible to define precise abstract operators on this abstraction.\footnote{For sequential composition we also need the set of variables for which a transparent run exists in order to allow a proper propagation of the transparency bits.} However, this abstraction is not effective, because $D_R$ is in general infinite.

Fortunately, it is not necessary to collect all compatible dependence traces in the abstraction, in order to describe the potential for forming dependences with a parallel context. It suffices to retain only certain short dependence traces in the abstraction that subsume the potential of all the other ones. A number of definitions and observations are necessary to make this precise. However, before we turn to the technical development, let us illustrate this kind of subsumption by a small example.

Consider the two dependence traces $\tau_1 = (1, (a, b) \cdot (c, d) \cdot (e, f), 1)$ and $\tau_2 = (1, (a, d) \cdot (e, f), 1)$. Intuitively, both have the gap $(d, e)$ but $\tau_1$ has the additional gap $(b, c)$. If a run $r$ of a parallel context can successively fill the two holes in $\tau_1$—i.e. if $r$ is compatible with the dependence trace $\tau_3 = (0, (b, c) \cdot (d, e), 0)$—it can also fill the single hole in $\tau_2$—i.e. $r$ is then also compatible with $\tau_4 = (0, (d, e), 0)$. Two interesting relationships between dependence traces popped up in this discussion. On the one hand, $\tau$ is “subsumed” by $\tau'$ in the sense sketched above as it has fewer gaps. On the other hand $\tau_1$ is “implied” by $\tau_3$ as it has less dependences: any run having $\tau_3$ as a dependence traces also has $\tau_4$ as a dependence trace.

We now define two orders on the set of dependence traces that capture these two relationships, the “implication order” and the “subsumption order”.

### 8.3 Implication Order

Let $\leq \subseteq \text{DT} \times \text{DT}$ be the smallest reflexive and transitive relation on the set of dependence traces that satisfies

1. $(\iota, \varphi \cdot (x, y) \cdot \psi, \kappa) \leq (\iota, \varphi \cdot \psi, \kappa)$, if $\varphi \neq \varepsilon \vee \iota = 0$ and $\psi \neq \varepsilon \vee \kappa = 0$;

2. $(1, \varphi, \kappa) \leq (0, \varphi, \kappa)$; and

3. $(\iota, \varphi, 1) \leq (\iota, \varphi, 0)$.

**Proposition 8.6** $\leq$ is a partial order on $\text{DT}$ called the implication order. \qed

The implication order $\leq$ allows us to weaken the information in a dependence trace in two ways. First of all, we can omit dependences (1.); here we must be careful not to omit the first or last dependence if the corresponding transparency bit is set, as otherwise the transparency bit might become invalid. Secondly, we
8.4. SUBSUMPTION ORDER

\[ \tau: \quad a \quad b \quad c \quad d \quad e \quad f \quad g \quad h \]
\[ \tau \leq \tau': \quad a \quad b \quad e \quad f \quad g \quad h \]
\[ \tau \sqsubseteq \tau'': \quad a \quad b \quad c \quad f \quad g \quad h \]

Figure 8.1: Illustration of implication and subsumption order.

can weaken the information about the transparency of the initial or final part of the run, by changing the transparency bits from 1 to 0 (2. & 3.).

The most appealing fact about \( \leq \) is that it preserves compatibility, which justifies the name “implication order”.

**Proposition 8.7 (\( \leq \) preserves compatibility)** Suppose \( r \vdash \tau \) and \( \tau \leq \tau' \). Then \( r \vdash \tau' \).

**Example 8.8** Consider the dependence trace \( \tau = (1, \langle (a, b), (c, d) \rangle, 0) \), which is, for instance, compatible with the run \( r = \langle b := a, c := 0, d := c, d := 0 \rangle \). Here is a list of the dependence traces that are implied by \( \tau \):

\[
\begin{align*}
\tau_1 &= (0, \langle (a, b), (c, d) \rangle, 0) \\
\tau_2 &= (1, \langle (a, b) \rangle, 0) \\
\tau_3 &= (0, \langle (a, b) \rangle, 0) \\
\tau_4 &= (0, \langle (c, d) \rangle, 0) \\
\tau_5 &= (0, \varepsilon, 0)
\end{align*}
\]

i.e., we have \( \tau \leq \tau_i \) for \( i = 1, \ldots, 5 \). All of them are dependence traces of \( r \). But we do not have \( \tau \leq \tau_6 \) for \( \tau_6 = (1, \langle (c, d) \rangle, 0) \). And indeed, \( \tau_6 \) is not a dependence trace of \( r \) because variable \( c \) is killed before it is read in \( r \). □

### 8.4 Subsumption Order

A transfer information with fewer gaps is more useful for the construction of dependences. We now define the subsumption order \( \sqsubseteq \subseteq DT \times DT \). Intuitively, \( \tau \sqsubseteq \tau' \) captures that \( \tau' \) has fewer gaps than \( \tau \) and thus subsumes the potential of \( \tau \) for forming dependences with a cooperating parallel context, hence the name subsumption order. We define \( \sqsubseteq \) as the smallest transitive and reflexive relation that satisfies

\[
(\iota, \varphi \cdot (x, y) \cdot \varphi' \cdot (x', y') \cdot \varphi'' \cdot \kappa) \sqsubseteq (\iota, \varphi \cdot (x, y') \cdot \varphi'' \cdot \kappa).
\]

In Figure 8.1 we illustrate the difference between the implication and the subsumption order. For simplicity, we only show the dependence sequences and omit
the transparency bits. In the top row we show a dependence trace \( \tau \), in the middle row a dependence trace \( \tau' \) that is implied by \( \tau \), and in the bottom row a dependence trace \( \tau'' \) that subsumes \( \tau \). With the implication order, dependences can be omitted (and transparency bits weakened). In contrast with the subsumption order, gaps are removed.

It is obvious from the defining rule that a transfer sequence \( \tau' \) that properly subsumes another transfer sequence \( \tau \) embodies a strictly shorter transfer sequence. Therefore, \( \sqsubseteq \) satisfies the ascending chain condition.

**Proposition 8.9** \( \sqsubseteq \) is a partial order on \( DT \) that satisfies the ascending chain condition: every strictly increasing sequence \( \tau_1 \sqsubseteq \tau_2 \sqsubseteq \cdots \) is finite.

Note that dependence traces of the form \((1, (x, y), 1)\), which by Proposition 8.5 correspond to dependences, are maximal w.r.t. \( \sqsubseteq \). This simple observation is important, as it implies that we cover all dependences even when we only consider \( \sqsubseteq \)-maximal dependence traces.

## 8.5 A Lattice of Antichains

An antichain with respect to \( \sqsubseteq \) (or \( \sqsubseteq \)-antichain for short) is a set \( D \sqsubseteq DT \) of dependence traces satisfying

\[
\neg \exists \tau, \tau' \in D : \tau \sqsubseteq \tau'.
\]

We denote the set of \( \sqsubseteq \)-antichains by \( AC \). We can lift the subsumption order to \( AC \) as follows:

\[
D \sqsubseteq D' \iff \forall \tau \in D \exists \tau' \in D' : \tau \sqsubseteq \tau'.
\]

The intuition is that \( D' \) subsumes \( D \), if every dependence trace in \( D \) is subsumed by some dependence trace in \( D' \). We call \( \sqsubseteq \) the antichain order. This is justified by the following lemma.

**Lemma 8.10** \( \sqsubseteq \) is a partial order on \( AC \).

**Proof.** It is straightforward to show that \( \sqsubseteq \) is reflexive and transitive. Let us show that \( \sqsubseteq \) is also antisymmetric and hence a partial order:

Suppose \( D \sqsubseteq D' \sqsubseteq D \). We show that \( D \sqsubseteq D' \), the reverse inclusion follows analogously. Suppose \( \tau \in D \). Then there is \( \tau' \in D' \) with \( \tau \sqsubseteq \tau' \) as \( D \sqsubseteq D' \). Because of \( D' \sqsubseteq D \), there is \( \tau'' \in D \) with \( \tau' \sqsubseteq \tau'' \). Thus, we have

\[
D \ni \tau \sqsubseteq \tau' \sqsubseteq \tau'' \in D.
\]

As \( D \) is an antichain, this implies that \( \tau = \tau'' \). Consequently, all these three dependence traces must be equal: \( \tau = \tau' = \tau'' \). But then \( \tau = \tau' \in D' \).
8.5. A LATTICE OF ANTIChAINS

A simple way to form an $\sqsubseteq$-antichain out of an arbitrary subset $D \subseteq DT$ is to consider the set of $\sqsubseteq$-maximal elements in $D$. We denote this set by $D^\dagger$:

$$D^\dagger = \{ \tau \in D \mid \neg \exists \tau' \in D : \tau \sqsubseteq \tau' \}.$$

The dependence traces in $D^\dagger$ subsume all dependence traces in $D$. In this sense, no interesting information is lost when going from $D$ to $D^\dagger$.

**Lemma 8.11 ($\dagger$ subsumes)** For any $\tau \in D$ there is a $\tau' \in D^\dagger$ such that $\tau \sqsubseteq \tau'$.

The lemma follows from the ascending chain condition.

The operator $\dagger$ is a co-closure operator that yields $\sqsubseteq$-antichains.

**Lemma 8.12 ($\dagger$ is a co-closure operator)**

1. $D^\dagger \subseteq D$.
2. $(D^\dagger)^\dagger = D^\dagger$.
3. $D^\dagger$ is an $\sqsubseteq$-antichain.
4. $\langle \cdot \rangle^\dagger$ is monotonic: $D \subseteq E$ implies $D^\dagger \subseteq E^\dagger$.

The proof of these properties is straightforward.

By 3., $\langle \cdot \rangle^\dagger$ is an operator from $2^{DT}$ to $AC$, by 4., this operator is monotonic. Indeed, as we will see in a minute, $\langle \cdot \rangle^\dagger$ is the lower adjoint of a Galois connection between $2^{DT}$ and $AC$. Before we elaborate this, let us show that the $\sqsubseteq$-antichains together with the lifted subsumption order form a complete lattice.

**Lemma 8.13 ($AC, \sqsubseteq$) is a complete lattice.** The least upper bound (lub) of a subset $\mathcal{D} \subseteq AC$ is $\bigcup \mathcal{D} := (\bigcup \mathcal{D})^\dagger$ and the least element of $(AC, \sqsubseteq)$ is $\bot_{AC} := \emptyset$.

**Proof.** In order to show that $(AC, \sqsubseteq)$ is a complete lattice, it suffices to demonstrate that any subset $\mathcal{D} \subseteq AC$ has a least upper bound. We show that, as claimed in the lemma, $E := (\bigcup \mathcal{D})^\dagger$ is indeed the least upper bound of $\mathcal{D}$.

Firstly, $E$ is an upper bound of $\mathcal{D}$: we have to show that $D \sqsubseteq E$ for any $D \in \mathcal{D}$, which is seen as follows:

$$\tau \in D \Rightarrow [D \in \mathcal{D}, \text{definition of } \bigcup \mathcal{D}]$$

$$\tau \in \bigcup \mathcal{D} \Rightarrow \exists \tau' \in E : \tau \sqsubseteq \tau'.$$
Secondly, \( E \) is smaller than any other bound \( \mathcal{D} \): suppose \( F \) is an arbitrary upper bound of \( \mathcal{D} \). Then \( E \subseteq F \) follows from the following chain of implications:

\[
\begin{align*}
\tau & \in E \\
\Rightarrow & \quad \text{[Definition of } \tau \text{, Lemma 8.12(1.)]} \\
\tau & \in \bigcup \mathcal{D} \\
\Rightarrow & \quad \text{[Definition of } \bigcup \mathcal{D}] \\
\exists D \in \mathcal{D} : \tau & \in D \\
\Rightarrow & \quad \text{[} D \subseteq F \text{ as } F \text{ is an upper bound of } \mathcal{D}, \text{ definition } \subseteq \text{]} \\
\exists \tau' & \in F : \tau \subseteq \tau'.
\end{align*}
\]

The least element of \((AC, \subseteq)\) is \( \perp_{AC} = \bigcup \emptyset = (\emptyset)^\downarrow = \emptyset\). \( \Box \)

Let us consider another operator on sets of dependence traces, the downwards closure operator \((\cdot)^\downarrow\). It is defined for sets \( D \in DT \) by

\[
D^\downarrow = \{ \tau \in DT \mid \exists \tau' \in D : \tau \subseteq \tau' \}.
\]

We can apply \((\cdot)^\downarrow\) in particular to antichains. Thus, we may consider \((\cdot)^\downarrow\) as an operator \((\cdot)^\downarrow : AC \rightarrow 2^{DT}\). It is not hard to see that \((\cdot)^\downarrow\) is monotonic.

**Proposition 8.14** Suppose \( A, B \in AC \). Then \( A \subseteq B \) implies \( A^\downarrow \subseteq B^\downarrow \). \( \Box \)

\((\cdot)^\downarrow\) and \((\cdot)^\uparrow\) are approximate inverses of each other.

**Lemma 8.15** For any \( D \in DT \), we have \( D^\downarrow \supseteq D \) and \( D^\uparrow \subseteq D \). For any \( A \in AC \), we even have \( A^\downarrow = A \). As a consequence, \((\cdot)^\uparrow, (\cdot)^\downarrow\) is a Galois surjection from \( 2^{DT} \) to \( AC \):

\[
2^{DT} \xrightarrow{(\cdot)^\uparrow} AC \xleftarrow{(\cdot)^\downarrow}
\]

**Proof.**

\( D^\uparrow \supseteq D \): By Lemma 8.11, there is, for any \( \tau \in D \), a dependence trace \( \tau' \in D^\uparrow \) such that \( \tau \subseteq \tau' \). This implies that \( \tau \in D^\uparrow \).

\( D^\downarrow \subseteq D \): If \( \tau \in D^\downarrow \), then \( \tau \) is a maximal element in \( D^\downarrow \). The maximal elements in \( D^\downarrow \), however, must already be in \( D \), as they cannot be added to \( D \) by lying strictly below another element of \( D \).

\( A^\uparrow = A \): It remains to show that \( A^\uparrow \supseteq A \). Any \( \tau \in A \) is maximal in \( A^\downarrow \). Therefore, any such \( \tau \) is also in \( A^\uparrow \). \( \Box \)
8.6. SHORT DEPENDENCE TRACES

The fact that \((\cdot)^\dagger, (\cdot)^\ddagger\) is a Galois surjection from \(2^D\) into \(AC\) shows us that \(\subseteq\)-antichains form a reasonable abstraction of sets of dependence traces. It also has other interesting consequences.

First of all, it implies that \((\cdot)^\dagger\) is universally disjunctive, which is important for ensuring that the abstraction mapping and the abstract operators defined later are universally disjunctive as well.

Proposition 8.16 \((\cdot)^\dagger : 2^D \rightarrow AC\) is universally disjunctive (‘distributive’).

Secondly, it shows us that we can present \((AC, \subseteq)\) isomorphically by downwards closed sets of dependence traces. From the theory of Galois connections, we know that the images of the upper and lower adjoint are isomorphic. This implies that \((AC, \subseteq)\), the image of \((\cdot)^\dagger\), is isomorphic to the image of \((\cdot)^\ddagger\), which is the set of downwards closed sets of dependence traces ordered by set inclusion. Note that this isomorphism depends on the fact that the underlying subsumption order on dependence traces satisfies the ascending chain condition. Otherwise, Lemma 8.11 would fail and we would not have the property \(D^\dagger \supseteq D\) that is crucial for the isomorphism between antichains and downwards closed sets.

For our purposes it is more convenient to work with antichains, because this leads to a more natural definition of the interleaving operator. If we work with downwards closed sets we may add dependence traces by means of downwards closure that are not compatible with any run in the abstracted run set. These additional dependence traces do not represent actual potential of the run set and in order to avoid imprecision, we must ensure that they are not considered for inferring dependence traces of interleavings.

8.6 Short Dependence Traces

A dependence sequence \(\varphi = \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle\) is called short if

1. all destination variables of dependences not counting the last one are distinct: for all \(1 \leq i < j < k\), \(y_i \neq y_j\); and
2. all source variables of dependences not counting the first one are distinct: for all \(1 < i < j \leq k\), \(x_i \neq x_j\).

A dependence trace \(\tau = (\iota, \varphi, \kappa)\) is called short if the embodied dependence sequence \(\varphi\) is short. We write \(DTS\) for the set of short dependence traces:

\[
DTS = \{ \tau \in DT \mid \tau\text{ is short} \}.
\]

Example 8.17 Consider the run \(r = \langle c := a, c := b, e := d \rangle\). One of its dependence traces is \(\tau = (1, \langle (a, c), (b, c), (d, e) \rangle, 1)\), which is not short due to the repetition of variable \(c\) as a target variable. But run \(r\) has also the dependence trace \(\tau' = (1, \langle (a, c), (d, e) \rangle, 1)\) which is short and subsumes \(\tau\). This is not a coincidence as we will see in a moment (Lemma 8.19). \(\square\)
We are interested in short dependence traces for two reasons. Firstly, there are only finitely many short dependence traces. This makes the abstract domain introduced in the next section finite as well and ensures that fixpoints for monotonic functions on this domain can be calculated effectively. The following lemma provides a formula for the cardinality of DTS and an asymptotic bound.

**Lemma 8.18** Let \( n = |X| \). Then \( |\text{DTS}| = 1 + 4n^2n!^2 \sum_{i=0}^{n} \frac{1}{i!^2} = O(n^{2n+2}) \).

**Proof.** By the pigeonhole principle, a dependence sequence cannot contain more than \( n + 1 \) dependences without violating the condition of shortness.

Let \( i \in \{0, \ldots, n\} \). For forming a short dependence trace \( \langle d_0, \ldots, d_i \rangle \) of length \( i + 1 \), we can choose arbitrary program variables as source variable of \( d_0 \) and as destination variable of \( d_i \); there are \( n^2 \) ways of doing this. We can choose the remaining source variables of \( d_1, \ldots, d_i \) as an arbitrary \( i \)-permutation of the variables in \( X \). (Recall that an \( i \)-permutation of \( X \) is an ordered sequence of \( i \) elements of \( X \), with no element appearing more than once in the sequence.) The same holds for the remaining destination variables of \( d_0, \ldots, d_{i-1} \). As there are \( \frac{n!}{(n-i)!} \) \( i \)-permutations \([11]\), there are thus \( n^2 \left( \frac{n!}{(n-i)!} \right)^2 \) short dependence sequences of length \( i + 1 \). There are four possible choices for the transparency bits in a dependence trace with a given non-empty dependence sequence. In addition we have a single dependence trace with an empty dependence sequence, viz. \((0, \varepsilon, 0)\). Summing up, the number of short dependence traces is thus

\[
1 + 4 \sum_{i=0}^{n} \left( n^2 \left( \frac{n!}{(n-i)!} \right)^2 \right) = 1 + 4n^2n!^2 \sum_{i=0}^{n} \frac{1}{(n-i)!^2} = 1 + 4n^2n!^2 \sum_{i=0}^{n} \frac{1}{i!^2}.
\]

Using the well-known fact that \( n! \leq n^n \) and bounding the sum by

\[
\sum_{i=0}^{n} \frac{1}{i!^2} \leq \sum_{i=0}^{n} \frac{1}{i!} \leq \sum_{i=0}^{\infty} \frac{1}{i!} = e
\]

the asymptotic bound \( O(n^{2n+2}) \) follows. \( \square \)

The asymptotic bound \( O(n^{2n+2}) \) for \( |\text{DTS}| \) is rather rough as it involves the rather bad estimate \( n^n \) for \( n! \). Using for instance Stirling's approximation \([11]\) for the factorial function, we could obtain tighter bounds. But the given bound suffices for our purposes.

The second reason why we are interested in short dependence traces is that they suffice to capture the potential of runs to aid in forming dependences 'up to subsumption' as the following lemma shows.

**Lemma 8.19 (Short dependence traces subsume)** Suppose \( r \vdash \tau \). Then there is a short dependence trace \( \tau' \) with \( r \vdash \tau' \) and \( \tau \subseteq \tau' \).
8.6. SHORT DEPENDENCE TRACES

Proof. Suppose \( r \vdash \tau = (\iota, \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle, \kappa) \). We describe a shortening procedure that can be iterated until a short dependence trace is obtained.

Suppose \( \tau \) is not already short. Let us assume that condition 1. is violated; if 2. is violated we can proceed analogously. Then there are indices \( i, j, 1 \leq i < j < k \), with \( y_i = y_j \). Consider the dependence trace \( \tau' \) obtained from \( \tau \) by removing the middle part \( \langle (x_{i+1}, y_{i+1}), \ldots, (x_j, y_j) \rangle \) of the dependence sequence:

\[
\tau' := (\iota, \langle (x_1, y_1), \ldots, (x_i, y_i), (x_{j+1}, y_{j+1}), \ldots, (x_k, y_k) \rangle, \kappa).
\]

It is not hard to see that both \( \tau \subseteq \tau' \) and \( \tau \leq \tau' \). By Proposition 8.7 the latter implies \( r \vdash \tau' \). \( \square \)

While this lemma shows us that short dependence traces are promising, we are not yet done. We still have to see that we can obtain the short dependence traces of a composed set of runs from the short dependence traces of the argument run sets. This is particularly challenging for run sets obtained by interleaving and will be the topic of Sections 8.8–8.12.

Shortening a dependence trace w.r.t. either \( \leq \) or \( \subseteq \) results again in a short dependence trace.

Lemma 8.20 (\( \leq \) and \( \subseteq \) preserve shortness) If \( \tau \) is short and \( \tau \leq \tau' \) or \( \tau \subseteq \tau' \), then \( \tau' \) is short.

Proof. All pairs of source or target variables in \( \tau' \) are also pairs of target variables in \( \tau \) if \( \tau \leq \tau' \) or \( \tau \subseteq \tau' \). \( \square \)

We denote the set of antichains of short dependence traces by ACS:

\[
\text{ACS} = \{ D \in \text{AC} \mid D \subseteq \text{DTS} \}.
\]

Lemma 8.19 implies that \( \subseteq \)-maximal dependence traces of a run (or run set) are always short. Therefore, if we restrict attention to short dependence traces of a run or run set, we still capture all maximal dependence traces. By working with ACS instead of AC, we code this knowledge into the domain. In particular, we do not loose dependences because the dependence traces of the form \( (1, \langle (a, b) \rangle, 1) \) that correspond to dependences are trivially short.

Lemma 8.21 (ACS, \( \subseteq \)) is a complete sub-lattice of (AC, \( \subseteq \)). Its height is \(|\text{DTS}| + 1 = O(n^{2n+2})\) where \( n = |X| \).

Proof. Suppose \( D \subseteq \text{ACS} \). In order to prove that (ACS, \( \subseteq \)) is a complete sub-lattice of (AC, \( \subseteq \)) we have to show that \( \bigcup D \in \text{ACS} \), i.e. that \( \bigcup D \subseteq \text{DTS} \): \( \bigcup D \) = \( \bigcup D \)

\[
\bigcup D = (\bigcup D)^\uparrow \subseteq \bigcup D \subseteq \text{DTS}.
\]

[Lemma 8.13] [Lemma 8.12] [\( D \subseteq \text{ACS} \)]
We can restrict the downwards closure operator to short dependence traces, i.e., redefine it by $D^+ = \{ \tau \in \text{DTS} \mid \exists \tau' \in D : \tau \sqsubseteq \tau' \}$ for $D \subseteq \text{DTS}$. It follows as in Lemma 8.15 that $\langle (\cdot)^\uparrow, (\cdot)^\downarrow \rangle$ is a Galois surjection from $2^{\text{DTS}}$ into ACS:

$$2^{\text{DTS}} \xrightarrow{\langle (\cdot)^\uparrow, (\cdot)^\downarrow \rangle} \text{ACS}$$

As a consequence $(\text{ACS}, \sqsubseteq)$ is isomorphic to the lattice of downwards closed subsets of DTS, ordered by set inclusion. The latter is a sub-lattice of $(2^{\text{DTS}}, \subseteq)$. Hence its height (and thus the height of $(\text{ACS}, \sqsubseteq)$) cannot be larger than the height of $(2^{\text{DTS}}, \subseteq)$ which is $|\text{DTS}| + 1$.

On the other hand, we can construct an ascending chain of size $|\text{DTS}| + 1$: Let $(x_1, \ldots, x_{|\text{DTS}|})$ be a topological sort of $(\text{DTS}, \sqsubseteq)$, i.e., a list containing all elements of DTS such that $x_i \sqsubseteq x_j$ implies $i \leq j$ for all $i, j \in \{1, \ldots, |\text{DTS}|\}$. Then we can define a chain of length $\text{DTS} + 1$ by choosing $A_0 = \emptyset$ and $A_i = (A_{i-1} \cup \{x_i\})^\uparrow$ for $i = 1, \ldots, |\text{DTS}|$. $A_{i-1} \sqsubseteq A_i$ is obvious, and $A_{i-1} \neq A_i$ holds because $A_{i-1} \subseteq \{x_1, \ldots, x_{i-1}\}$, which is seen by a straightforward induction, and thus $x_i$ is maximal in $A_{i-1} \cup \{x_i\}$ due to the topological sort property.

The asymptotic bound $|\text{DTS}| + 1 = O(n^{2n+2})$ follows from Lemma 8.18. □

### 8.7 The Abstract Domain

Let us now define the abstract domain. The values of the abstract domain are pairs $(T, D)$ consisting of a set $T \subseteq X$ of variables and an $\sqsubseteq$-antichain $D$ of short dependence traces. In the applications the dependence traces in $D$ form the more interesting piece of information. $T$ represents the variables for which a transparent run exists. This information is necessary in order to allow a proper propagation of initial and final transparency information in sequential contexts.

Thus, the abstract domain, AD, is given by

$$\text{AD} = 2^X \times \text{ACS}.$$  

The order on the abstract domain, which we also denote by the symbol $\sqsubseteq$, is defined as the lift of the inclusion order on the $T$ component and the antichain order $\sqsubseteq$ on the $D$ component: $(T, D) \sqsubseteq (T', D')$ iff

1. $T \subseteq T'$ and
2. $D \sqsubseteq D'$.

$(\text{AD}, \sqsubseteq)$ is the product lattice of the complete lattices $(2^X, \subseteq)$ and $(\text{ACS}, \sqsubseteq)$ and hence also a complete lattice. Both of these lattices have $\emptyset$ as their least element. Hence, $(\emptyset, \emptyset)$ is the least element of $\sqsubseteq$.

**Lemma 8.22** $(\text{AD}, \sqsubseteq)$ is a complete lattice with least element $(\emptyset, \emptyset)$. Its height is $O(n^{2n+3})$ where $n = |X|$.
Proof. It only remains to prove the asymptotic bound for the height. The height of \( \text{AD} \) is the product of the height of \( (2^X, \subseteq) \), which is \( n + 1 \), and the height of \( (\text{ACS}, \sqsubseteq) \), which is \( O(n^{2n+2}) \) by Lemma 8.21. This implies the stated bound. \( \square \)

Let us now define an abstraction mapping \( \alpha : \text{NR} \rightarrow \text{AD} \) that captures the intuition how non-atomic run sets are abstracted to values from \( \text{AD} \):

\[
\alpha(R) = (T_R, D_R), \quad \text{where}
\]

\[
T_R = \{ x \in X \mid \exists r \in R : r \text{ is transparent for } x \} \quad \text{and}
\]

\[
D_R = \{ \tau \in \text{DT} \mid \exists r \in R : r \vdash \tau \}^\dagger.
\]

Before we proceed, let us show that this is a proper definition.

**Lemma 8.23** \( \alpha \) is well-defined.

**Proof.** We have to show two things for an arbitrary \( R \in \text{NR} \):

1. \( D_R \) consists of short dependence traces.

2. \( D_R \) is an \( \sqsubseteq \)-antichain.

To 1.: Assume there is \( \tau \in D_R \) that is not short. Then there is \( r \in R \) with \( r \vdash \tau \). By Lemma 8.19, there is a short dependence trace \( \tau' \) with \( r \vdash \tau' \) and \( \tau \sqsubseteq \tau' \). In particular \( \tau' \in \{ \tau \in \text{DT} \mid \exists r \in R : r \vdash \tau \} \) and, as \( \tau' \) is short and \( \tau \) is not, we even have \( \tau \sqsubseteq \tau' \). But this shows that \( \tau \) is not maximal in \( \{ \tau \in \text{DT} \mid \exists r \in R : r \vdash \tau \} \) and hence is not a member of \( D_R \), a contradiction.

To 2.: This is ensured by Lemma 8.12(4.). \( \square \)

The abstraction \( \alpha(R) \) of a run set \( R \) is induced by the following abstraction \( \beta(r) \) of the single runs \( r \in R \):

\[
\beta(r) = (T_r, D_r), \quad \text{where}
\]

\[
T_r = \{ x \in X \mid r \text{ is transparent for } x \} \quad \text{and}
\]

\[
D_r = \{ \tau \in \text{DT} \mid r \vdash \tau \}^\dagger.
\]

**Lemma 8.24** Suppose \( R \in \text{NR} \). Then \( \alpha(R) = \bigcup \{ \beta(r) \mid r \in R \} \).
Proof. We have $\bigcup \{ \beta(r) \mid r \in R \} = (\bigcup_{r \in R} T_r, \bigcup_{r \in R} D_r)$. It is obvious that $T_R = \bigcup_{r \in R} T_r$. On the other hand, we have $\bigcup_{r \in R} D_r = (\bigcup_{r \in R} \{ t \mid r \vdash t \})^{\dagger}$, by Lemma 8.13. It is not hard to show that this equals $D_R$ by considering the $\sqsubseteq$- and the $\sqsupset$-direction separately.

The fact that $\alpha$ is induced by an abstraction on single runs has nice consequences. First of all, it immediately implies that $\alpha$ is monotonic.

**Proposition 8.25** $\alpha$ is monotonic: $R \subseteq R'$ implies $\alpha(R) \subseteq \alpha(R')$.

Secondly, it even implies that $\alpha$ is universally disjunctive.

**Proposition 8.26** $\alpha$ is universally disjunctive.

This property is crucial for preciseness of the abstract interpretation of constraint systems, cf. Chapter 9, and shows us that $\alpha$ provides a proper abstraction of run sets by being the lower adjoint of a Galois connection. For completeness let us introduce the corresponding upper adjoint. It is $\gamma : \text{AD} \to \text{NR}$, defined by

$$\gamma(T, D) = \{ r \mid T_r \subseteq T, \; D_r \sqsubseteq D \}.$$

**Proposition 8.27** $(\alpha, \gamma)$ is a Galois connection between $\text{NR}$ and $\text{AD}$:

$$\begin{array}{ccc}
\alpha & \downarrow & \gamma \\
\text{NR} & \iff & \text{AD}
\end{array}$$

We leave the proof that $\gamma$ is well-defined and forms a Galois connection with $\alpha$ to the reader.

In the sections that follow we define composition operators on $\text{AD}$ and show that they are precise abstractions of the corresponding operators on $\text{NR}$. We start with the pre- and the post-operator that are rather simple. Then we discuss sequential composition. Afterwards we consider the most interesting and challenging operator: interleaving. Finally, we discuss the abstract semantics of base edges.

### 8.8 Pre-Operator

We define the (abstract) pre-operator, $\text{pre}^\#: \text{AD} \to \text{AD}$, as follows:

$$\text{pre}^\#(T, D) = \begin{cases} 
(\emptyset, \emptyset), & \text{if } D = \emptyset \\
(\{ \xi \mid \xi \in D \mid (\iota, \varphi, \kappa) \in DT \}, \{ \iota, \varphi, \kappa \} \in D), & \text{if } D \neq \emptyset.
\end{cases}$$

**Lemma 8.28** $\text{pre}^\#$ is well-defined: for any $(T, D) \in \text{AD}$, $\text{pre}^\#(T, D) \in \text{AD}$.
8.8. PRE-OPERATOR

Proof. The only property that is not obvious is that \( A := \{ (\iota, \varphi, \kappa) \in DT \mid (\iota, \varphi, 0) \in D \} \) is an antichain of short dependence traces. First of all, any dependence trace \((\iota, \varphi, \kappa) \in A\) inherits being short from the dependence trace \((\iota, \varphi, 0) \in D\) that induces its inclusion in \(A\). Secondly, assume that there are distinct dependence traces \(\tau, \tau' \in A\) with \(\tau \subseteq \tau'\). By the definition of the subsumption order, the transparency bits in \(\tau\) and \(\tau'\) must coincide, i.e. we can write them in the form \(\tau = (\iota, \varphi, \kappa)\) and \(\tau' = (\iota, \varphi', \kappa)\). From \(\tau \subseteq \tau'\) it follows that also \((\iota, \varphi, 0) \subseteq (\iota, \varphi', 0)\). But then \((\iota, \varphi, 0)\) and \((\iota, \varphi', 0)\) are two distinct comparable dependence traces in \(D\), which is a contradiction to \(D\) being an antichain. Hence \(\text{pre}^\#(T, D)\) must be an antichain of short dependence traces.

The crucial observation for the adequacy of the definition of \(\text{pre}^\#\) is this.

**Lemma 8.29** \( r \vdash (\iota, \varphi, 0) \) if and only if there is a prefix \(r'\) of \(r\) with \(r' \vdash (\iota, \varphi, \kappa)\).

Proof. Let \(\varphi = \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle\).

\(\Rightarrow\): Suppose \(r \vdash (\iota, \varphi, 0)\). If \(\kappa = 0\), we can choose \(r' = r\). So assume \(\kappa = 1\). Choose a decomposition \(t_0 r_1 \cdots r_k t_k\) of \(r\) that witnesses \(r \vdash (\iota, \varphi, 0)\). Let \(r' = t_0 r_1 \cdots r_k\). Then, clearly, \(r'\) is a prefix of \(r\) and \(t_0 r_1 \cdots r_k t'_k\) with \(t'_k = \varepsilon\) is a decomposition of \(r'\) that witnesses \(r' \vdash (\iota, \varphi, 1)\).

\(\Leftarrow\): Suppose \(r'\) is a prefix of \(r\) with \(r' \vdash (\iota, \varphi, \kappa)\). Choose \(r''\) with \(r = r' r''\), and let \(t_0 r_1 \cdots r_k t_k\) be a decomposition of \(r'\) that witnesses \(r' \vdash (\iota, \varphi, \kappa)\). Then \(t_0 r_1 \cdots r_k t'_k\) with \(t'_k = t_k r''\) is a decomposition of \(r\) that witnesses \(r \vdash (\iota, \varphi, 0)\).

We can now show that the abstract pre-operator is a precise abstraction of the concrete pre-operator.

**Theorem 8.30 (Abstract pre-operator is precise)** Suppose \(R \in \text{NR}\). Then \(\alpha(\text{pre}(R)) = \text{pre}^\#(\alpha(R))\).

Proof. If \(R = \emptyset\), then \(\alpha(\text{pre}(R)) = \alpha(\emptyset) = (\emptyset, \emptyset) = \text{pre}^\#(\emptyset, \emptyset) = \text{pre}^\#(\alpha(R))\). So let us assume \(R \neq \emptyset\).

By unfolding the definitions, we see that \(\alpha(\text{pre}(R)) = (T_{\text{pre}(R)}, D_{\text{pre}(R)})\) with

\[
T_{\text{pre}(R)} = \{ x \mid \exists r, r' \in \text{Runs} : r \cdot r' \in R \wedge r \text{ is transparent for } x \}
\]

\[
D_{\text{pre}(R)} = \{ \tau \mid \exists r, r' \in \text{Runs} : r \cdot r' \in R \wedge r \vdash \tau \}^\dagger .
\]

In order to evaluate the right hand side, note first that \(D_R\) is non-empty: there is a run \(r \in R\) and any such run satisfies \(r \vdash (0, \varepsilon, 0)\); moreover, \((0, \varepsilon, 0)\) is \(\sqsubseteq\)-maximal and hence contained in \(D_R\). Consequently, the second case applies in the definition of \(\text{pre}^\#\) and we have \(\text{pre}^\#(\alpha(R)) = \text{pre}^\#(T_R, D_R) = (X, D)\) with

\[
D = \{ (\iota, \varphi, \kappa) \in DT \mid (\iota, \varphi, 0) \in D_R \}^\dagger
\]
Thus, we have to show $T_{\text{pre}(R)} = X$ and $D_{\text{pre}(R)} = D$.

$T_{\text{pre}(R)} \subseteq X$ is trivial. In order to see the reverse inclusion, i.e. that $T_{\text{pre}(R)}$
contains any $x \in X$, choose an arbitrary $r \in R$ and observe that the empty run
$\varepsilon$ is a prefix of $r$ that is transparent for any variable $x$.

The following chain of implications shows $D_{\text{pre}(R)} \subseteq D$:

$$(\iota, \phi, \kappa) \in D_{\text{pre}(R)}$$

$\Rightarrow$ [Equation above, Lemma 8.12(1.)]

$\exists r, r' \in \text{Runs} : r \cdot r' \in R \land r \vdash (\iota, \phi, \kappa)$

iff [Lemma 8.29]

$\exists r \in R : r \vdash (\iota, \phi, 0)$

iff [Set comprehension]

$$(\iota, \phi, 0) \in \{ \tau \in \text{DT} \mid \exists r \in R : r \vdash \tau \}$$

$\Rightarrow$ [Lemma 8.11, definition $D_R$]

$\exists r' \in D_R : (\iota, \phi, 0) \subseteq \tau'$

$\Rightarrow$ [See below]

$\exists \tau \in D : (\iota, \phi, \kappa) \subseteq \tau$.

The reasoning for the last step is as follows. The fewer gaps ordering $\subseteq$ is concerned only with removing gaps from the dependence sequence $\phi$ in a dependence trace but leaves the initial and final transparency information untouched. Hence, the dependence trace $\tau' \in D_R$ with $(\iota, \phi, 0) \subseteq \tau'$ must have the form $\tau' = (\iota, \psi, 0)$. But then $\tau := (\iota, \psi, \kappa) \in D$ and $(\iota, \phi, \kappa) \subseteq (\iota, \psi, \kappa)$.

Finally, we show $D \subseteq D_{\text{pre}(R)}$:

$$(\iota, \phi, \kappa) \in D$$

$\Rightarrow$ [Above equation for $D$, Lemma 8.12(1.)]

$$(\iota, \phi, 0) \in D_R$$

$\Rightarrow$ [Definition of $D_R$, Lemma 8.12(1.)]

$\exists r \in R : r \vdash (\iota, \phi, 0)$

iff [Lemma 8.29]

$\exists r, r' : r \cdot r' \in R \land r \vdash (\iota, \phi, \kappa)$

iff [Set comprehension]

$$(\iota, \phi, \kappa) \in \{ \tau \in \text{DT} \mid \exists r, r' : r \cdot r' \in R \land r \vdash \tau \}$$

$\Rightarrow$ [Lemma 8.11]

$\exists \tau \in \{ \tau \in \text{DT} \mid \exists r, r' : r \cdot r' \in R \land r \vdash \tau \} \uparrow : (\iota, \phi, \kappa) \subseteq \tau$

iff [Above equation for $D_{\text{pre}(R)}$]

$\exists \tau \in D_{\text{pre}(R)} : (\iota, \phi, \kappa) \subseteq \tau$. 

\[\text{CHAPTER 8. DEPENDENCE TRACES}\]
This completes the proof. \hfill \Box

## 8.9 Post-Operator

We define the (abstract) post-operator, \( post^\# : AD \to AD \), in complete analogy to the pre-operator as follows:

\[
post^\#(T, D) = \begin{cases} 
(\emptyset, \emptyset), & \text{if } D = \emptyset \\
(X, \{(i, \varphi, \kappa) \in DT \mid (0, \varphi, \kappa) \in D\}), & \text{if } D \neq \emptyset.
\end{cases}
\]

By symmetry to the pre-operator we obtain that the post operator is well-defined and a precise abstraction of the post-operator on non-atomic run sets.

**Theorem 8.31 (Abstract post-operator is precise)** Suppose \( R \in NR \). Then \( \alpha(post(R)) = post^\#(\alpha(R)) \). \hfill \Box

## 8.10 Sequential Composition

The (abstract) sequential composition operator, \( ;^\# : AD \times AD \to AD \), which we write as an infix operator, is defined by

\[
(T, D);^\#(T', D') = (T \cap T', (D \cdot D')^\dagger),
\]

where

\[
D \cdot D' = \{(i, \varphi, \kappa) \in D \mid \kappa = 1 \Rightarrow \bar{\varphi} \in T'\} 
\cup \{(i, \varphi, \kappa) \in D' \mid i = 1 \Rightarrow \bar{\varphi} \in T\} 
\cup \{(i, \varphi \cdot \psi, \kappa) \in DTS \mid (i, \varphi, 0) \in D, (0, \psi, \kappa) \in D'\} 
\cup \{(i, \varphi \cdot (x, z) \cdot \psi, \kappa) \in DTS \mid 
\exists y : (i, \varphi \cdot (x, y), 1) \in D, (1, y, z) \cdot \psi, \kappa) \in D'\}.
\]

Before we explain the intution underlying this definition we show well-definedness.

**Lemma 8.32** The abstract sequential composition operator \( ;^\# \) is well-defined.

**Proof.** We have to show that \( (D \cdot D')^\dagger \in ACS \) for all \( D, D' \in ACS \), i.e. that \( (D \cdot D')^\dagger \) is an \( \sqsubseteq \)-antichain of short dependence traces.

It is easy to see that \( D \cdot D' \) (and hence its subset \( D \cdot D')^\dagger \)) contains only short dependence traces: the first two sets contain only dependence traces from \( D \) or \( D' \), which consequently are short, and the constructions in the third and forth set are explicitly restricted to contain short dependence traces. The application of the \( ^\dagger \)-operator ensures that \( (D \cdot D')^\dagger \in ACS \) is an \( \sqsubseteq \)-antichain. \hfill \Box
A sequentially composed run $r$: \[ r' \cdot r'' \]

How subruns mediating dependences in $\varphi$ may overlap with pieces of $r$: \[
\begin{align*}
1) & \quad r_1 \ldots r_k \\
2) & \quad r_1 \ldots r_i \quad r_i \ldots r_k \\
3) & \quad r_1 \ldots r_i \quad r_{i+1} \ldots r_k \\
4) & \quad r_1 \ldots r_i \quad r_i \ldots r_k
\end{align*}
\]

Figure 8.2: Intuition of sequential composition.

Obviously, a run $r = r' \cdot r''$ composed of two runs $r'$ and $r''$ is transparent for a variable $x$ if and only if both $r'$ and $r''$ are. Therefore, transparency information must be intersected in a sequential composition.

Let us explain the intuition underlying the definition of $D \cdot D'$. Suppose given a run $r = r' \cdot r''$ which is composed of two runs $r' \in D$ and $r'' \in D'$ that use distinct virtual variables (virtual($r'$) \cap virtual($r''$) = $\emptyset$). Assume that $\tau = (\iota, \varphi, \kappa)$ with $\varphi = \langle d_1, \ldots, d_k \rangle$ is a dependence trace compatible with $r$. Each dependence $d_i$ in $\varphi$ is mediated by a sub-piece $r_i$ of $r$; we can choose the $r_i$ as short as possible (i.e., such that it starts with an assignment that reads the source variable of $d_i$ and ends with an assignment to the destination variable of $d_i$). There are four possibilities, how these sub-pieces can be situated in $r$ as illustrated in Fig. 8.2:

1) all of them can lie in $r'$;
2) all of them can lie in $r''$;
3) there is an $i$, $1 \leq i < k$, such that $r_1, \ldots, r_i$ lie in $r'$ and $r_{i+1}, \ldots, r_k$ lie in $r''$;
4) there is an $i$ such that $r_i$ overlaps with the join point of $r'$ and $r''$.

These four cases are handled by the four sets appearing in the definition of $D \cdot D'$:

1) in this case, $\tau$ is also a dependence trace of $r'$. Vice versa, dependence traces $\tau' = (\iota', \varphi', \kappa')$ of $r'$ give rise to dependence traces of $r$. However, if $\kappa' = 1$, no statement that kills $\varphi'$, the destination variable of the last dependence in $\varphi'$, is allowed after $r_k$. Therefore, $r'$ must be transparent for $\varphi'$; hence the side condition in set (8.1).

2) this case is symmetric to case 1).

3) in this case, $r'$ has the dependence trace $(\iota, \langle d_1, \ldots, d_i \rangle, 0)$ and $r''$ the dependence trace $(0, \langle d_{i+1}, \ldots, d_k \rangle, \kappa)$. Vice versa, dependence traces of $r'$ and $r''$ of this form give rise to a dependence trace of $r$. 

4) choose variables $x, z \in X$ such that $d_i = (x, z)$. Sub-run $r_i$ accomplishes the transfer from $x$ to $z$ via certain intermediate variables. One of these intermediate variables, say $y$, must bridge the joint point between $r'$ and $r''$ (i.e., it is assigned to in $r'$, read from in $r''$ and not killed in between). As $r$ and $r'$ use distinct virtual variables, $y$ must be a program variable: $y \in X$. Then $\langle s, \langle d_1, \ldots, d_{i-1}, (x, y) \rangle, 1 \rangle$ is a dependence of $r'$ and $\langle 1, \langle d_1, \ldots, d_{i-1}, (x, y) \rangle, 1 \rangle$ is a dependence of $r''$. 1 as the final component of $r'$ and first component of $r''$ is justified, as $y$ is not killed from the place where it is assigned to in $r'$ and read in $r''$. Similarly, dependences of $r'$ and $r''$ of the above form give rise to a dependence trace of $r$.

It is not hard to see that in all four cases the dependence traces of $r'$ and/or $r''$ in question are short and $\sqsubseteq$-maximal if $r$ is and, vice versa, that each short and $\sqsubseteq$-maximal dependence trace of $r$ can be composed of short and $\sqsubseteq$-maximal dependence traces compatible with $r'$ and $r''$ in the described way.

**Lemma 8.33 (Abstract sequential composition operator is precise)**

*Suppose $R, S \in NR$. Then $\alpha(R; S) = \alpha(R); \# \alpha(S)$.**

**Proof.** By formalizing the intuition described above. \qed

### 8.11 Interleaving

Transparency information for the interleaving $R \otimes S$ of two run sets $R$ and $S$ is easy to obtain from transparency information of the components: a transparent run for a variable $x$ exists in $R \otimes S$ if and only if each component set contains a transparent run. Therefore, the transparency information in $T_R$ and $T_S$ must simply be intersected.

By far more interesting is to consider the dependence traces in $D_{R \otimes S}$ as the two threads modeled by $R$ and $S$ can cooperate in order to mediate dependences. More specifically, a dependence $(u, v)$ can be composed of complementary dependence sequences of two runs $r \in R$ and $s \in S$, e.g., as illustrated here:

**Transfers of $r$:** $u = x_1 \rightarrow y_1 \quad x_2 \rightarrow y_2 \quad x_3 \rightarrow y_3 \quad \cdots \quad x_{k-1} \rightarrow y_{k-1} \quad x_k \rightarrow y_k = v$

**Transfers of $s$:** $y_1 \rightarrow x_2 \quad y_2 \rightarrow x_3 \quad \cdots \quad y_{k-1} \rightarrow x_k$

Of course such a combination of complementary dependence sequences can also start and/or end with a dependence of $s$. And, as a border case, one of the dependence sequences can be empty; the other then just consists of a single dependence. Before we define the abstract interleaving operator, we present in the next section the general definition of when two dependence sequences complement each other to a single dependence and also introduce a relation $C$ that extends this definitions to dependence traces.
8.11.1 Complementary Dependence Traces

Let \( \varphi, \psi \in \text{DS} \) be two dependence sequences (one of them can be empty) and \( u, v \in X \). Choose variables such that \( \varphi = \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle \), \( k \geq 0 \). We say that \( \psi \) complements \( \varphi \) to \((u, v)\) if one of the following cases applies:

1. \( \varphi \neq \varepsilon, u = \overleftarrow{\varphi}, v = \overrightarrow{\varphi}, \) and \( \psi = \langle (y_1, x_2), \ldots, (y_{k-1}, x_k) \rangle \);

2. \( \varphi \neq \varepsilon, \psi \neq \varepsilon, u = \overleftarrow{\varphi}, v = \overrightarrow{\psi}, \) and \( \psi = \langle (y_1, x_2), \ldots, (y_{k-1}, x_k), (y_k, v) \rangle \);

3. \( \varphi \neq \varepsilon, \psi \neq \varepsilon, u = \overleftarrow{\psi}, v = \overrightarrow{\varphi}, \) and \( \psi = \langle (u, x_1), (y_1, x_2), \ldots, (y_{k-1}, x_k) \rangle \); or

4. \( \psi \neq \varepsilon, u = \overleftarrow{\psi}, v = \overrightarrow{\psi}, \) and \( \psi = \langle (u, x_1), (y_1, x_2), \ldots, (y_{k-1}, x_k), (y_k, v) \rangle \).

Intuitively, \( \psi \) complements \( \varphi \) to \((u, v)\) if the two of them can alternately be combined to a gap-free transfer from \( u \) to \( v \). The different cases are distinguished by whether the first read in this gap-free transfer comes from \( \varphi \) (cases 1/2) or \( \psi \) (cases 3/4) and whether the last write is in \( \varphi \) (cases 1/3) or \( \psi \) (cases 2/4).

Now, consider a dependence trace \( \tau \) compatible with a run \( t \in R \otimes S \) which is an interleaving of the runs \( r \in R, s \in S \). Then every single dependence in \( \tau \) must be obtained in the above described fashion from pieces of dependence traces compatible with \( r \) and \( s \). We, therefore, generalize this notion of completion to dependence traces as follows: suppose given dependence traces \( \tau, \tau_0, \tau_1 \), where \( \tau = (\iota, \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle, \kappa) \), \( \tau_0 = (\iota_0, \varphi, \kappa_0) \), \( \tau_1 = (\iota_1, \psi, \kappa_1) \). Then we say that \( \overline{\tau_1} \) complements \( \tau_0 \) to \( \tau \), \( C(\tau_0, \tau_1, \tau) \) for short, if there are dependence sequences \( \varphi_1, \ldots, \varphi_k, \psi_1, \ldots, \psi_k \) such that

1. \( \varphi = \varphi_1 \cdot \ldots \cdot \varphi_k \) and \( \psi = \psi_1 \cdot \ldots \cdot \psi_k \);

2. \( \psi_1 \) complements \( \varphi_i \) to \((x_i, y_i)\) for \( i = 1, \ldots, k \).

3. \( \iota = 1 \) implies \( \iota_0 = 1 \) and \( \psi_1 \) complements \( \varphi_1 \) to \((x_1, y_1)\) according to cases 1 and 2, or \( \iota_1 = 1 \) and \( \psi_1 \) complements \( \varphi_1 \) according to cases 3 and 4; and

4. \( \kappa = 1 \) implies \( \kappa_0 = 1 \) and \( \psi_k \) complements \( \varphi_k \) to \((x_k, y_k)\) according to cases 1 and 3, or \( \kappa_1 = 1 \) and \( \psi_k \) complements \( \varphi_k \) according to cases 2 and 4.
The typical situation of two dependence traces $\tau_0$ and $\tau_1$ that complement each other to a dependence trace $\tau$ is illustrated in Fig. 8.3. For clarity we omit the transparency bits. The dashed vertical lines indicate equality of variables.

A number of elementary properties of the relation $C$ is collected in the following lemma.

**Lemma 8.34 (Basic properties of $C$)** Suppose $\tau, \tau_0, \tau_1 \in DT$. Then

1. $C$ is symmetric in the first two parameters: $C(\tau_0, \tau_1, \tau)$ if and only if $C(\tau_1, \tau_0, \tau)$.
2. $(0, \varepsilon, 0)$ is a ‘neutral element’: $C((0, \varepsilon, 0), \tau, \tau)$.
3. In particular, $C((0, \varepsilon, 0), (0, \varepsilon, 0), (0, \varepsilon, 0))$.

**Proof.** Left to the reader. $\square$

### 8.11.2 Interleaving Operator

We are now in the position to define the (abstract) interleaving operator, $\otimes^\#: AD \times AD \to AD$, which we write again as an infix operator:

$$(T, D) \otimes^\# (T', D') = \{ (T \cap T', \{ \tau'' \in DTS \mid \exists \tau \in D, \tau' \in D' : C(\tau, \tau', \tau''))^\uparrow \}^\uparrow \}$$

By restricting the set construction to short dependence traces and applying the $(\cdot)^\uparrow$ operator, the interleaving operator is trivially well-defined. The goal of the remainder of this section is to show that it is a precise abstraction of the interleaving operator on sets of non-atomic runs.

**Theorem 8.35 (Abstract interleaving operator is precise)**

Suppose $R, S \in NR$. Then $\alpha(R \otimes S) = \alpha(R) \otimes^\# \alpha(S)$.

The proof is deferred to Section 8.11.5. Before that, we establish a number of lemmas that capture the main insights underlying the proof.

### 8.11.3 Soundness Lemmas

The lemmas in this section are concerned with the soundness of the abstract interleaving composition operator, i.e. they are crucial for the proof that $\alpha(R \otimes S) \subseteq \alpha(R) \otimes^\# \alpha(S)$ for any two run sets $R, S$. The critical point here is to guarantee that our definition of the abstract interleaving operator includes enough dependence traces.

As a first step, we show that each dependence trace that is compatible with some interleaving of two runs $r, s$ can also be obtained by combining two dependence traces of the component runs $r$ and $s$ via the relation $C$.

Let $r, s, t \in \text{Runs}$ with virtual$(r) \cap \text{virtual}(s) = \emptyset$ and $\tau \in DT$. 
Lemma 8.36  Suppose \( t \in r \otimes s \) and \( t \vdash \tau \). Then there are \( \tau_r, \tau_s \in \mathbb{D}T \) with \( r \vdash \tau_r, s \vdash \tau_s \), and \( C(\tau_r, \tau_s, \tau) \).

Proof. Assume that \( t \) is an interleaving of \( r \) and \( s \) and \( \tau = (\iota, (d_1, \ldots, d_k), \kappa) \) is a dependence trace of \( t \). Each dependence \( d_i \) is mediated by a certain sub-run \( t_i \) of \( t \) and each \( t_i \) is an interleaving of certain sub-runs of \( r \) and \( s \).

From \( t_i \) we can construct dependence traces \( \varphi_i \) and \( \psi_i \) of these sub-pieces of \( r \) and \( s \) such that \( \varphi_i \) complements \( \psi_i \) to dependence \( d_i \). This is described below. Then \( \varphi_1 \cdots \varphi_k \) and \( \psi_1 \cdots \psi_k \) are dependence sequences of \( r \) and \( s \), resp., and we can choose transparency bits \( \iota_r, \kappa_r, \iota_s, \kappa_s \in \mathbb{B} \) such that \( \tau_r = (\iota_r, \varphi_1 \cdots \varphi_k, \kappa_r) \) and \( \tau_s = (\iota_s, \psi_1 \cdots \psi_k, \kappa_s) \) are dependence traces of \( r \) and \( s \), resp., such that \( C(\tau_r, \tau_s, \tau) \) holds. Specifically, we choose \( \iota_r = \iota \) if the first assignment instance involved in the mediation of \( d_1 \) belongs to \( r \) and \( \iota_s = \iota \) if it belongs to \( s \), and similarly for the final transparency bits and the last assignment instance involved in the mediation of \( d_k \). All other transparency bits are chosen 0.

Let us now explain how to construct the dependence sequences \( \varphi_i \) and \( \psi_i \) mentioned above. Choose program variables \( x, y \) such that \( d_i = (x, y) \). Sub-run \( t_i \) of \( t \) mediates \( d_i \) via certain assignment instances \( a_j := e_j, j = 1, \ldots, l \), as specified in the definition of mediation. In particular, \( a_i = y \). Each of these assignment instances lies either in a sub-piece of \( r \) or a sub-piece of \( s \). Let us consider the case that the first assignment instance \( a_1 := e_1 \) lies in a sub-piece of \( r \); the case that it lies in a sub-piece of \( r \) is analogous. We can then find indices \( 0 < j_0 < j_1 < \ldots < j_n \) such that \( a_j := e_j \) lies in a sub-piece of \( r \) if \( j_m < j \leq j_{m+1} \) for an even \( m \in \{0, \ldots, n-1\} \) and in a sub-piece of \( s \) otherwise. In particular, for any \( j \in \{j_1, \ldots, j_{n-1}\} \) one of the assignments instances \( a_j := e_j \) and \( a_{j+1} := e_{j+1} \) lies in a sub-piece of \( r \) and the other one in a sub-piece of \( s \). This implies that \( a_j \) must be a program variable, because it appears in \( e_{j+1} \) according to the definition of mediation and \( \text{virtual}(r) \cap \text{virtual}(s) = \emptyset \). Choose now

\[
\begin{align*}
\varphi_i &= \langle (x, a_j), (a_j, a_j), \ldots, (a_{j-2}, a_{j-1}), (a_{j-1}, a_j) \rangle, \\
\psi_i &= \langle (a_j, a_j), (a_j, a_{j-1}), \ldots, (a_{j-1}, y) \rangle
\end{align*}
\]

if \( n \) is even and

\[
\begin{align*}
\varphi_i &= \langle (x, a_j), (a_j, a_j), \ldots, (a_{j-1}, y) \rangle, \\
\psi_i &= \langle (a_j, a_j), (a_j, a_{j-1}), \ldots, (a_{j-2}, a_{j-1}) \rangle
\end{align*}
\]

if \( n \) is odd. Then \( \varphi_i \) and \( \psi_i \) are dependence sequences of the sub-runs of \( r \) and \( s \) that comprise \( t_i \) and, obviously, \( \varphi_i \) complements \( \psi_i \) to \( d_i \). \( \square \)

Example 8.37  Fig. 8.4 illustrates the construction in the proof of Lemma 8.36. The run \( t \) is an interleaving of the runs \( r \) and \( s \). We can thus decompose \( r \)
and $s$ into sub-runs such that $t$ is obtained by alternately shuffling these sub-runs together; in the example $r = r_1r_2r_3r_4r_5$, $s = s_1s_2s_3s_4$, and $t = r_1s_1r_2s_2r_3s_3r_4s_4r_5$.

Let us assume that $\tau = (\iota, \langle (a, b), (c, d) \rangle, \kappa)$ is a dependence trace of $t$. Then there are sub-runs $t_1$ and $t_2$ of $t$ that mediate the two dependences $(a, b)$ and $(c, d)$, e.g., as shown in the figure. These sub-runs overlap in a certain way with the decompositions of $r$ and $s$; in the example in the figure, for instance, $t_1$ overlaps with a postfix of $r_1$, all of $s_1, r_2, s_2$, and a prefix of $r_3$. The dependence $(a, b)$ is mediated via certain intermediate assignments $a_i := e_i$ (not shown in the figure); we call these assignments crucial in the following.

There may be sub-runs of $r$ and/or $s$ that overlap with $t_i$ but do not contain a crucial assignment. Such sub-runs must be transparent for the variable that transfers the dependence at this moment and can be ignored. In our example, $r_2$ is such a sub-run and $g$ is the variable that transfers the dependence while $r_2$ is executed.

Whenever two successive crucial assignments lie in sub-pieces of different runs, the dependence must be transferred in a program variable between these assignments because $r$ and $s$ do not share virtual variables. In the figure, e.g., $e$ is the variable that transfers the dependence from the last crucial assignment in $r_1$ to the first crucial assignment in $s_1$ and $f$ transfers it from the last crucial assignment in $s_1$ to the first crucial assignment in $r_2$. From these variables we can construct dependence traces $\tau_r$ and $\tau_s$ of $r$ and $s$ such that $C(\tau_r, \tau_s, \tau)$ holds. In Fig. 8.4, for instance, we have $\tau_r = (\iota, \langle (a, e), (f, b), (h, i), (j, d) \rangle, \kappa)$ and $\tau_s = (0, \langle (e, \kappa), (c, h), (i, j) \rangle, 0)$.

Lemma 8.36 ensures that combining dependence traces of component runs via $C$ is fundamentally rich enough to give us all dependence traces of potential interleavings. However, in our abstract domain, we do not collect all dependence traces but only the maximal ones. Therefore, we only combine the maximal dependence traces of component runs in the definition of interleaving, which is the best we can do with the available information. A legitimate question to ask
is, whether this is sufficient. Can we really obtain all the maximal dependence traces just from the maximal dependence traces of the components? The answer to this question must be yes; otherwise Theorem 8.35 would be wrong. But Lemma 8.36 alone does not suffice to prove this.

The next lemma provides a kind of shortening rule that is crucial for the proof that maximal dependence traces of component run sets suffice to infer the maximal dependence traces of their interleaving.

Suppose \( \tau_0, \tau'_0, \tau_1, \tau \in \text{DT} \).

**Lemma 8.38** Suppose \( C(\tau_0, \tau_1, \tau) \) and \( \tau_0 \subseteq \tau'_0 \). Then there are dependence traces \( \tau'_1, \tau' \in \text{DT} \) such that \( \tau_1 \leq \tau'_1, \tau \subseteq \tau' \), and \( C(\tau'_0, \tau'_1, \tau') \). By symmetry of \( C \), Lemma 8.34(1), an analogous property holds with the roles of \( \tau_0 \) and \( \tau_1 \) exchanged.

**Proof.** The proof is illustrated in Fig. 8.5. In a) the typical situation of dependence traces \( \tau_0, \tau_1 \) and \( \tau \) with \( C(\tau_0, \tau_1, \tau) \) is shown. For clarity the transparency bits are omitted. In b) a typical dependence trace \( \tau'_0 \) with \( \tau_0 \subseteq \tau'_0 \) is shown. It is obtained from \( \tau_0 \) by removing all gaps between the target variable \( u \) of a certain dependence \( d \) in \( \tau_0 \) and the destination variable \( v \) of a later dependence \( e \). We can remove all the dependencies from \( \tau_1 \) that are used to fill some or all of these gaps in \( C(\tau_0, \tau_1, \tau) \). This results in a dependence trace \( \tau'_1 \) with \( \tau_1 \leq \tau'_1 \) as shown in b). Then the dependence traces \( \tau'_0 \) and \( \tau'_1 \) complement each other to a dependence trace \( \tau' \) with \( \tau \leq \tau' \) as shown. As border cases, we may have \( \tau'_0 = \tau_0 \), if none of the gaps between \( d \) and \( e \) is filled in \( C(\tau_0, \tau_1, \tau) \), or \( \tau' = \tau \) if \( d \) and \( e \) are used in \( C(\tau_0, \tau_1, \tau) \) in the same dependence of \( \tau \). But this does not invalidate our reasoning as \( \subseteq \) and \( \leq \) are reflexive.

By iteratively applying this shortening rule, we obtain the following lemma that is of direct use in the proof of Theorem 8.35.

**Lemma 8.39** Suppose \( \tau_r \in \{ \tau \mid \exists r \in R : r \vdash \tau \}, \tau_s \in \{ \tau \mid \exists s \in S : s \vdash \tau \} \), and \( C(\tau_r, \tau_s, \tau) \). Then there are \( \tau'_r \in D_R, \tau'_s \in D_S \), and \( \tau' \in \text{DT} \) with \( C(\tau'_r, \tau'_s, \tau') \) and \( \tau \subseteq \tau' \).
Proof. The problem is that \( \tau_r \) and \( \tau_s \) need not be \( \sqsubseteq \)-maximal in their respective set. Hence they may not belong to \( D_R \) and \( D_S \), respectively. By iteratively applying Lemma 8.38, however, we can determine dependence traces \( \tau_r^\uparrow \) and \( \tau_s^\uparrow \) that are \( \sqsubseteq \)-maximal in these sets (and hence belong to \( D_R \) and \( D_S \), respectively) as well as a dependence trace \( \tau^\uparrow \) with \( C(\tau_r^\uparrow, \tau_s^\uparrow, \tau^\uparrow) \) and \( \tau \sqsubseteq \tau^\uparrow \).

We start with \((\tau_r^\uparrow, \tau_s^\uparrow, \tau^\uparrow) := (\tau_r, \tau_s, \tau)\). This initialization trivially ensures \( \tau_r^\uparrow \in \{ \tau \mid \exists r \in R : r \vdash \tau \} \), \( \tau_s^\uparrow \in \{ \tau \mid \exists s \in S : s \vdash \tau \} \), \( C(\tau_r^\uparrow, \tau_s^\uparrow, \tau^\uparrow) \), and \( \tau \sqsubseteq \tau^\uparrow \), which is an invariant of the loop we describe in the following.

If \( \tau_r^\uparrow \) is not \( \sqsubseteq \)-maximal in \( \{ \tau \mid \exists r \in R : r \vdash \tau \} \), we can choose a dependence trace \( \tau'_r \in \{ \tau \mid \exists r \in R : r \vdash \tau \} \) which is strictly larger: \( \tau_r^\uparrow \sqsubset \tau'_r \). Then, by Lemma 8.38, there are \( \tau'_s \) and \( \tau' \) with \( \tau'_r \leq \tau'_s \), \( \tau \sqsubseteq \tau^\uparrow \sqsubseteq \tau' \), and \( C(\tau'_r, \tau'_s, \tau') \). By Proposition 8.7, \( \tau'_s \in \{ \tau \mid \exists r \in R : r \vdash \tau \} \), hence the invariant remains valid. We then set \((\tau'_r, \tau'_s, \tau^\uparrow) := (\tau'_r, \tau'_s, \tau')\). We can proceed analogously, if \( \tau'_s \) is not maximal in \( \{ \tau \mid \exists s \in S : s \vdash \tau \} \).

This shortening procedure is applied iteratively until both \( \tau_r^\uparrow \) and \( \tau_s^\uparrow \) are \( \sqsubseteq \)-maximal in their respective sets. Termination is guaranteed, because in each step either the dependence sequence in \( \tau_r^\uparrow \) or in \( \tau_s^\uparrow \) becomes shorter and the dependence sequence in the other dependence trace does not become longer. \( \Box \)

8.11.4 Completeness Lemmas

The lemmas in this section are concerned with completeness of the interleaving operator, i.e. they are important for the proof that \( \alpha(R \otimes S) \equiv \alpha(R) \otimes \# \alpha(S) \) for any two non-atomic run sets \( R, S \). They crucially depend on runs being non-atomic.

A dependence mediated by a non-atomic run \( r \) must involve a virtual variable at a certain stage as assignments that have program variables on both the left- and the right-hand-side do not occur in non-atomic runs. But when the execution of \( r \) is in such a stage, no parallel thread can disturb propagation of the dependence because parallel threads do not interfere on virtual variables. This is the idea underlying the proof of the following lemma.

Lemma 8.40 Suppose \( r, s \) are runs with virtual\((r) \cap virtual(s) = \emptyset \), and \( x, y \in X \). If \( r \) mediates \((x, y)\) then there is a run \( t \in r \otimes s \) that mediates \((x, y)\).

Proof. Suppose \( r \) mediates \((x, y)\). This means that \( r \) can be written in the form \( r = r_0 \cdot \langle a_1 := e_1 \rangle \cdot r_1 \cdot \langle a_2 := e_2 \rangle \cdot r_2 \cdot \ldots \cdot \langle a_{i-1} := e_{i-1} \rangle \cdot r_i \) as in the definition of mediation. Then in particular \( e_1 \) contains the variable \( x \). As \( x \) is a program variable, this implies by the form of assignments appearing in runs that \( a_1 \) must be a virtual variable (cf. the definition of Asg). As virtual\((r) \cap virtual(s) = \emptyset \), \( s \) therefore cannot contain an assignment to \( a_1 \). Consequently, \( s \) is transparent for

\begin{equation}
8.11. \text{INTERLEAVING} \quad 141
\end{equation}
$a_1$. Hence the run $t \in r \otimes s$ defined by

$$t := r_0 \cdot (a_1 := e_1) \cdot s \cdot r_1 \cdot (a_2 := e_2) \cdot r_2 \cdot \ldots \cdot r_{l-1} \cdot (a_l := e_l) \cdot r_l$$

still mediates the dependence $(x, y)$. \hfill \Box

Note that this argument crucially depends on the assumption about the form of assignments in runs that derives from the assumption that assignments execute non-atomically. If assignments execute atomically, the above lemma is no longer valid.

**Example 8.41** Consider the parallel execution of the two straight-line programs

$$\pi_1 = (y := x) \quad \text{and} \quad \pi_2 = (x := 0; y := 0).$$

If assignment statements execute atomically, there are just three possible runs,

1) $\{x := 0, y := 0, y := x\}$,

2) $\{x := 0, y := x, y := 0\}$, and

3) $\{y := x, x := 0, y := 0\}$.

None of these runs mediates the dependence $(x, y)$ because either $x$ is killed before $y := x$ is executed as in 1) and 2), or $y$ is killed after $y := x$ is executed as in 2) and 3).

If, on the other hand, assignment statements may execute non-atomically, then the two initialization statements in $\pi_2$ could well be executed after $x$ is read but before $y$ is written. This is witnessed by the run

4) $\{v := x, x := 0, y := 0, y := v\}$,

where $v$ is a virtual variable, in our model of non-atomic execution. In contrast to the runs 1)-3), run 4) mediates the dependence $(x, y)$. \hfill \Box

Lemma 8.40 provides an intuitive explanation why precise analysis of parallel programs is simpler if we assume non-atomic execution of assignments. With this assumption dependences once generated by a thread cannot be definitely destroyed by its environment. Thus, an analysis that collects positive information about potential dependences is precise. (In order to do this in a compositional fashion it must collect more information, namely (maximal, short) dependence traces.)

This is different if we analyze with respect to the assumption that assignments execute atomically. Then there is a complex interplay between the way dependences are generated by a thread and the order of re-initializations performed by its environment as illustrated by the above example. Therefore, an analysis that just collects positive information is doomed to be imprecise.

**Lemma 8.42** Suppose $r_0, r_1$ are runs with $\text{virtual}(r_0) \cap \text{virtual}(r_1) = \emptyset$ and $\tau_0, \tau_1, \tau$ are dependence traces with $r_0 \vdash \tau_0$, $r_1 \vdash \tau_1$, and $C(\tau_0, \tau_1, \tau)$. Then there is a run $r \in r_0 \otimes r_1$ such that $r \vdash \tau$. 

8.11. INTERLEAVING

**Proof.** For notational convenience, we discuss the case that the dependence sequence in \( \tau \) consists of just a single transfer; the generalization to arbitrary transfer sequences is left to the reader. Let \( \tau = (\iota, \langle (u, v) \rangle, \kappa) \). Furthermore, let \( \tau_0 = (\iota_0, \phi, \kappa_0) \) and \( \tau_1 = (\iota_1, \psi, \kappa_1) \).

Let us assume that case 2 in the definition of \( C(\tau_0, \tau_1, \tau) \) applies; the other cases are similar. Then we can choose variables \( u = x_1, \ldots, x_{k+1} = v \) such that

\[
\varphi = \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle,
\psi = \langle (y_1, x_2), \ldots, (y_k, x_{k+1}) \rangle,
\]

and it is \( \iota_0 = 1 \) if \( \iota = 1 \) and \( \kappa_1 = 1 \) if \( \kappa = 1 \). As \( r_0 \vdash \tau_0 \) and \( r_1 \vdash \tau_1 \) we can write \( r_0 \) and \( r_1 \) in the form

\[
r_0 = t_0^0 t_1^0 r_2^0 \cdots t_{k-1}^0 r_k^0, \quad r_1 = t_0^1 t_1^1 r_2^1 \cdots t_{k-1}^1 r_k^1,
\]

such that

- a) \( r_i^0 \) mediates \( (x_i, y_i) \) and \( r_i^1 \) mediates \( (y_i, x_{i+1}) \) for \( i = 1, \ldots, k \);
- b) \( t_0^0 \) is transparent for \( u \) if \( \iota = 1 \) (and hence \( \iota_0 = 1 \)); and
- c) \( t_k^1 \) is transparent for \( v \) if \( \kappa = 1 \) (and hence \( \kappa_1 = 1 \)).

The run \( r_0^0 r_1^0 r_2^0 \cdots r_k^0 r_k^1 \) clearly mediates the dependence \( (u, v) \), but in order to construct an interleaving of \( r_0 \) and \( r_1 \), we must also execute the intermediate code pieces \( t_i^1 \). Fortunately, each of the dependences realized by some \( r_i^1 \) must involve a virtual variable; and, while the transfer is in such a stage, code pieces of the other run, \( r_{1-j} \), can safely be executed without destroying the dependence, due to the disjointness of the virtual variables used in \( r_0 \) and \( r_1 \). Thus, we can execute each code piece \( t_i^1 \) at such a stage of execution of \( r_i^0 \) and, similarly, \( t_i^0 \) during such a stage of \( r_i^1 \). The rest of the proof pursues this argument more formally.

By Lemma 8.40, there are interleavings \( s_i^0 \in r_i^0 \otimes t_i^1 \) and \( s_i^1 \in r_i^1 \otimes t_i^0 \) such that, for \( i = 1, \ldots, k \),

- \( s_i^0 \) still mediates \( (x_i, y_i) \) and
- \( s_i^1 \) still mediates \( (y_i, x_{i+1}) \).

Then the run

\[
r := t_0^0 s_1^0 s_2^1 \cdots s_k^0 s_k^1
\]

is an interleaving of \( r_0 \) and \( r_1 \) (i.e. \( r \in r_1 \otimes r_2 \)). On the other hand, \( r \vdash \tau \) because \( s_1^0 s_2^0 s_3^1 \cdots s_k^0 s_k^1 \) mediates the dependence \( (u, v) \) and items b) and c) above give the transparency properties. \( \square \)

Note that the proof relies on Lemma 8.40. Like that lemma, Lemma 8.42 fails to hold if assignments execute atomically as illustrated by the following example.
Example 8.43 Consider the two programs $\pi_1 = (y := x)$ and $\pi_2 = (x := 0; y := 0; z := y)$ and the three dependence traces $\tau_1 = (1, \langle (x, y) \rangle, 1)$, $\tau_2 = (1, \langle (y, z) \rangle, 1)$, and $\tau = (1, \langle (x, z) \rangle, 1)$.

If assignments execute atomically, $\pi_1$ has only the run $r_1 = (y := x)$ and $\pi_2$ has only the run $r_2 = (x := 0, y := 0, z := y)$. Clearly, $\tau_1$ is a dependence trace of $r_1$ and $\tau_2$ is a dependence trace of $r_2$, independently of whether assignments execute atomically or not. Moreover, $C(\tau_1, \tau_2, \tau)$ holds.

But only the following four runs are possible interleavings of $r_1$ and $r_2$:

1) $\langle x := 0, y := 0, z := y, y := x \rangle$,

2) $\langle x := 0, y := 0, y := x, z := y \rangle$,

3) $\langle x := 0, y := x, y := 0, z := y \rangle$, and

4) $\langle y := x, x := 0, y := 0, z := y \rangle$.

It is not hard to see that $\tau$ is compatible with none of these runs.

If, on the other hand, assignments do not execute atomically, there are also runs like

5) $\langle v := x, x := 0, y := 0, y := v, u := y, z := u \rangle$,

where $u, v$ are virtual variables, which possess $\tau$ as a dependence trace. □

8.11.5 Proof of Theorem 8.35

We can now put the pieces together and prove Theorem 8.35. By unfolding the definitions, we have

$$\alpha(R \otimes S) = (T_{R \otimes S}, D_{R \otimes S}) \text{ and } \alpha(R) \otimes^\# \alpha(S) = (T_R \cap T_S, D), \text{ where }$$

$$D = \{ \tau \in \text{DTS} | \exists \tau_R \in D_R, \tau_S \in D_S : C(\tau_R, \tau_S, \tau) \}^\dagger.$$

Consequently, we have to show $T_{R \otimes S} = T_R \cap T_S$ and $D_{R \otimes S} = D$.

"$T_{R \otimes S} \subseteq T_R \cap T_S$": If $x \in T_{R \otimes S}$, then there is a run $t \in R \otimes S$ that is transparent for $x$. By definition, $t$ is an interleaving of runs $r \in R$ and $s \in S$. These runs $r, s$ must then also be transparent for $x$. Thus, $x \in T_R \cap T_S$.

"$T_{R \otimes S} \supseteq T_R \cap T_S$": If $x \in T_R \cap T_S$, then there are runs $r \in R$ and $s \in S$ that are transparent for $x$. By bounded renaming of virtual variables these runs can be chosen such that they do not share virtual variables. Then all interleavings of these two runs are in $S \otimes R$, and all of them are transparent for $x$. Thus, $x \in T_{R \otimes S}$.
“$D_{R\otimes S} \subseteq D$”: In order to show this relationship, assume that we are given $\tau \in D_{R\otimes S}$. Then we have, by the definition of $D_{R\otimes S}$ and Lemma 8.12(1.):

$$\exists t \in R \otimes S : t \vdash \tau$$

iff [Definition $R \otimes S$]
$$\exists r, s \in S, t \in r \otimes s : t \vdash \tau$$

⇒ [Lemma 8.36]
$$\exists r, s \in S, \tau_r, \tau_s \in \text{DT} : r \vdash \tau_r \land s \vdash \tau_s \land C(\tau_r, \tau_s, \tau)$$

⇒ [Shunting, set comprehension]
$$\exists \tau_r \in \{ \tau \mid \exists r \in R : r \vdash \tau \}, \tau_s \in \{ \tau \mid \exists s \in S : s \vdash \tau \} : C(\tau_r, \tau_s, \tau).$$

⇒ [Lemma 8.39]
$$\exists \tau_r \in D_R, \tau_s \in D_S, \tau' \in \text{DT} : C(\tau_r, \tau_s, \tau') \land \tau \subseteq \tau'$$

iff [Set comprehension, see below]
$$\exists \tau' \in \{ \tau \in \text{DTS} \mid \exists \tau_R \in D_R, \tau_S \in D_S : C(\tau_R, \tau_S, \tau) \} : \tau \subseteq \tau'$$

⇒ [Definition $D$, Lemma 8.11]
$$\exists \tau' \in D : \tau \subseteq \tau'.$$

There is a little snag in the step marked by ‘see below’: for the direction $\Rightarrow$, we must prove that $\tau'$ can be chosen as a short dependence trace, which is not true for this step in isolation. But, it is true under the assumption that $\tau \in D_{R\otimes S}$ which underlies the whole calculation: as a consequence of this assumption $\tau$ is short and this implies that any $\tau'$ with $\tau \subseteq \tau'$ must also be short (Lemma 8.20). A calculation, in which this step is valid in isolation, requires to furnish each of the preceding predicates with the conjunct $\tau \in D_{R\otimes S}$, which would clutter the calculation. This is the reason why we resort to this explanation.

“$D_{R\otimes S} \supseteq D$”: This is shown by the following chain of implications:

$$\tau \in D$$

⇒ [Definition of $D$, Lemma 8.12(1.)]
$$\exists \tau_R \in D_R, \tau_S \in D_S : C(\tau_R, \tau_S, \tau)$$

⇒ [Definition $D_R, D_S$, Lemma 8.12(1.)]
$$\exists r, s \in S, \tau_R, \tau_S : r \vdash \tau_R \land s \vdash \tau_S \land C(\tau_R, \tau_S, \tau)$$

iff [By bounded renaming of virtual variables in $s$]
$$\exists r, s \in S, \tau_R, \tau_S :$$
$$r \vdash \tau_R \land s \vdash \tau_S \land C(\tau_R, \tau_S, \tau) \land \text{virtual}(r) \cap \text{virtual}(s) = \emptyset$$
⇒ [Lemma 8.42, definition $R \otimes S$]
\[ \exists t \in R \otimes S : t \vdash \tau \]
iff [Set comprehension]
\[ \tau \in \{ \tau \in DT | \exists r \in R \otimes S : r \vdash \tau \} \]
⇒ [Lemma 8.11, definition $D_{R \otimes S}$]
\[ \exists \tau' \in D_{R \otimes S} : \tau \subseteq \tau'. \]

This ends the proof of Theorem 8.35. \hfill \Box

8.12 Base Edges

In Chapter 7 we discussed that the atomicity assumptions about assignments may vary and that this would give rise to different definitions of the non-atomic run sets $[x := e]$ assigned to an assignment statement $x := e$. Fortunately, all reasonable choices give rise to the same abstraction which is given by the following definition:

\[ [[x := e]]^\# = (X \setminus \{x\}, \{ (\iota, ((y, x)), \kappa) | \iota, \kappa \in \mathbb{B}, y \text{ appears in } e \}). \]

Whatever atomicity assumption we are working with, all runs in $[x := e]$ will contain certain auxiliary assignments to virtual variables and a single assignment to $x$. No program variable except $x$ will ever be the target of an assignment in a run in $[[x := e]]$. Hence all non-atomic runs are transparent just for the program variables in $X \setminus \{x\}$, which explains the adequacy of the first component of $[[x := e]]^\#$.

The fact that any non-atomic run contains just a single assignment to $x$ and no other assignments to program variables implies that no dependence trace of a non-atomic run can embody a dependence sequence that is longer than one or has a destination variable different from $x$. Each reasonable non-atomic run induces the same dependences between program variables as $x := e$, hence the induced dependences are $(y, x)$ where $y$ is a variable appearing in $e$. Moreover, no reasonable run kills a variable in $e$ before it reads it or kills $x$ after it has written it, which implies that the transparency bits can be chosen arbitrarily.

All dependence traces included in the second component of $[[x := e]]^\#$ are trivially short and $\sqsubseteq$-maximal, which implies well-definedness.

**Proposition 8.44** Suppose $x := e \in \text{Stmt}$. Then $\alpha([[[x := e]]) = [[x := e]]^\#$. \hfill \Box

Statement **skip** has just the single run $e$, which is obviously transparent for all variables and has just the dependence trace $(0, e, 0)$. Hence, we define $[[\text{skip}]]^\# = (X, \{(0, e, 0)\})$. 

8.13 Run-Time

The goal of this section is to show that we can compute the abstract operations $\text{pre}^\#, \text{post}^\#, ;^\#$, and $\otimes^\#$ in time $2^{p(|X|)}$, where $p(x)$ is a polynomial. We emphasize that we do neither intend to develop efficient implementations of the operations nor to present a very precise analysis. The results of this section will mainly be used in order to establish the qualitative complexity statement that the algorithms developed later run in exponential time. We are, however, interested in uncovering the parameter of exponential growth: it is the number of program variables $|X|$ rather than the size of the parallel flow graph.

Let us investigate the most expensive operation, interleaving, to some detail. First of all, we recall its definition from Section 8.11:

$$(T, D) \otimes^\# (T', D') = (T \cap T', D'^\dagger) \text{ where } D'' = \{\tau'' \in \text{DTS} \mid \exists\tau \in D, \tau' \in D' : C(\tau, \tau', \tau'')\}.$$  

The sets $T$ and $T'$ are subsets of $X$, the set of program variables. Computing the intersection of $T$ and $T'$ is cheap: if we represent these sets as bit-strings (of length $|X|$), we can clearly calculate the intersection in time $\mathcal{O}(|X|)$ by looking through the bit-strings for $T$ and $T'$ once.

$D$ and $D'$ are antichains of short dependence traces, hence $D, D' \subseteq \text{DTS}$. By Lemma 8.18, the cardinality of DTS and hence of $D$ and $D'$ is $\mathcal{O}(|X|^{2|X|+2})$. This clearly is $\mathcal{O}(2^{p_0(|X|)})$ for some polynomial $p_0(x)$ because $x^{2x+2} = \log_2(x)(2x+2) \leq 2^{2x^2+2x}$. We can hence consider at most $\mathcal{O}(2^{p_0(|X|)} \cdot 2^{p_0(|X|)}) = \mathcal{O}(2^{2p_0(|X|)})$ pairs of dependence traces $\tau$ and $\tau'$ when computing $D''$. For each fixed pair of dependence traces $\tau, \tau'$ all dependence traces $\tau''$ with $C(\tau, \tau', \tau'')$ can be determined in time $\mathcal{O}(2^{p_1(|X|)})$ for some polynomial $p_1(x)$. We leave it to the reader to invent some procedure for this task that realizes this rather brutal bound. Even a very naive procedure that lists all short dependence traces $\tau''$ and then checks for each listed dependence trace whether $C(\tau, \tau', \tau'')$ holds will do. The observation that $\tau, \tau'$, and $\tau''$ are short, and hence the length of their dependence sequences is bounded by $|X| + 1$ is helpful. As a consequence, we can calculate $D''$ in time $\mathcal{O}(2^{2p_0(|X|)} \cdot 2^{p_1(|X|)}) = \mathcal{O}(2^{2p_0(|X|) + p_1(|X|)})$. Again $\mathcal{O}(2^{p_0(|X|)})$ is an asymptotic bound for the size of $D''$ because $D'' \subseteq \text{DTS}$. It is, therefore, not hard to see that $D'^\dagger$, the second component of $(T, D) \otimes^\# (T', D')$, can be computed from $D''$.
in time $\mathcal{O}(2^{p_2(|X|)})$ for some polynomial $p_2(x)$. Hence the overall cost of computing $(T, D) \otimes^# (T', D')$ is $\mathcal{O}(|X| + 2^{2p_0(|X|) + p_1(|X|)} \cdot 2^{p_2(|X|)}) = \mathcal{O}(2^{p(|X|)})$ for some polynomial $p(x)$.

By similar considerations we can show that the other operations can be computed in time $\mathcal{O}(2^{p(|X|)})$ too.

**Lemma 8.47** The operations $\text{pre}^#$, $\text{post}^#$, $;^#$, and $\otimes^#$ can be computed in time $\mathcal{O}(2^{p(|X|)})$ for some polynomial $p(x)$. \hfill $\square$

## 8.14 Conclusion

In this chapter, we have defined an abstraction of sets of non-atomic runs from which the mediated dependences can be derived. Run sets are abstracted to antichains of short dependence traces that capture the potential to mediate dependences in cooperation with a parallel environment. The abstraction also records the set of program variables for which a transparent run exists in the abstracted run set. This information is necessary to propagate the transparency bits of the dependence traces properly in sequential contexts.

We have defined abstract interpretations of the operations and constants used in the constraint systems of Section 6.5 and have shown that they precisely abstract the corresponding operations on sets of non-atomic runs. We can thus effectively determine the dependences mediated by the sets of runs characterized by the constraint systems of Section 6.5 by solving these constraint systems over the abstract lattice $(\mathbb{A}, \sqsubseteq)$ domain. This can be done effectively because this lattice is finite. In particular, we can determine the dependences mediated by bridging runs in procedural parallel flow graphs. This information can in turn be used to detect all copy constants and eliminate faint code completely, which is explained in detail in the next chapter.

In summary, the dependence traces abstraction provides us with a means to perform precise interprocedural dependence analysis in parallel programs.
Chapter 9

Detecting Copy Constants and Eliminating Faint Code

In this chapter we show that we can detect copy constants and eliminate faint code in parallel flow graphs completely relative to the non-atomic semantics. The basic idea is to evaluate the constraint system for bridging runs over the abstract domain AD from the previous section and to exploit this information.

We have seen that the abstract counterparts of the operators and constants appearing in the constraint systems in Chapter 6 precisely abstract the corresponding operators on non-atomic run sets. Moreover, the abstraction mapping \( \alpha : \text{NR} \to \text{AD} \) is universally disjunctive (Proposition 8.26). This implies that the least solution of the constraint systems over domain AD consists just of the abstractions of the least solution over domain \( \text{NR} \). This is commonly known in the area of abstract interpretation [12, 13] and follows directly from the following fixpoint-theoretic lemma known as Transfer-Lemma [4] or \( \mu \)-Fusion Rule [44].

**Lemma 9.1 (Transfer lemma)** Suppose \( L, L' \) are complete lattices, \( f : L \to L \) and \( g : L' \to L' \) are monotonic functions and \( h : L \to L' \) (Fig. 9.1).

If \( h \) is universally disjunctive and \( h \circ f = g \circ h \) then \( h(\mu f) = \mu g \), where \( \mu f \) and \( \mu g \) are the least fixpoints of \( f \) and \( g \), respectively.

\( \Box \)

The least solution of a constraint system over some domain corresponds in a straightforward way to the least fixpoint of a function derived from the constraints. The facts recalled above ensure that the premises of the Transfer-Lemma hold for the functions \( f \) and \( g \) derived from the concrete and abstract interpretation of the constraint systems over non-atomic runs and over AD, respectively, and the transfer function \( h \) that component-wise maps the concrete interpretation \( x \) of each variable \( X \) of the constraint system to its abstraction \( \alpha(x) \). Hence, by solving the constraint system for bridging runs over domain AD, we can determine the abstractions of the non-atomic bridging runs precisely. From the abstract values we can read off in particular all the dependences mediated by bridging runs:
if \((T, D) \in AD\) is the precise abstraction of a set \(R\) of (non-atomic) runs, i.e., if \(\alpha(R) = (T, D)\), then \((x, y)\) is a dependence mediated by a run in \(R\) if and only if 
\((1, (x, y)), 1\) \(\in D\) (Proposition 8.5).

Based on this information we can detect copy constants and eliminate faint code. Corresponding algorithms are developed in this chapter. These algorithms are not efficient: they run in exponential time. More precise statements about the dependence of the run time from the different input parameters can be found in Theorem 9.6. The point here is not to develop efficient algorithms—we will see in the next chapter that all these problems are intractable already for loop-free parallel programs—the point is that these problems are effectively solvable at all! This comes as a surprise, because the corresponding problems are uncomputable, if we assume atomic execution of assignments (Chapter 5).

Without further ado, we present, in the remainder of this chapter, the algorithms for detection of copy constants (Section 9.1) and faint-code elimination (Section 9.2). While we do not perform formal correctness proofs for these algorithms, we argue (hopefully convincingly) that the presented algorithms solve the respective problems. In our opinion a more formal argumentation would obscure rather than clarify matters here. After the presentation of the algorithms, we analyze their asymptotic run-time in Section 9.3 and finish the chapter with some concluding remarks. Throughout this chapter we assume that execution of base statements is non-atomic.

### 9.1 Copy-Constant Detection

Roughly speaking, a variable \(x\) is a copy constant either if it is assigned a constant value (e.g., through \(x := 42\)) or if it is assigned the value of another copy constant (e.g., in \(y := 42; x := y\)). Thus, in copy-constant detection only assignments of the simple form \(x := k\), where \(k\) is a constant or variable are interpreted, all other forms of assignments (e.g. \(x := y + 1\)) are (conservatively) assumed to make \(x\) non-constant [70].

Algorithm 9.1 in Fig. 9.2 reads a parallel flow graph, a program point \(v \in N\), and a program variable \(y \in X\) and decides whether \(y\) is a copy constant at \(v\) or not. For this purpose it first computes (in Steps 1 and 2) for each program point \(w\) the set

\[
I[w] = \{x \mid e_{Main} \Rightarrow c_w \xrightarrow{r} c_v, At_w(c_w), At_v(c_v), \bar{r} \text{ mediates dep. } (x, y)\}.
\]
Algorithm 9.1

**Input:** A parallel flow graph as defined in Chapter 6, a program point $v \in N$ and a program variable $y \in X$.

**Output:** “yes” if $y$ is a copy constant at $v$; “no” otherwise.

**Method:**

1) Compute—by standard fixpoint iteration—the least solution over domain $(\text{AD, } \sqsubseteq)$ of the constraint system for bridging runs to program point $v$; this gives us a value $B_v^\#[u]$ for each program point $u$.

2) Set $I[w] := \{ x \mid (1, \langle(x, y), 1\rangle) \in B^\#_v[w]\}$ for each program point $w \in N$.

3) Set $flag := \text{false}$ and $val := \text{unset}$.

4) If $I[e_{\text{Main}}] \neq \emptyset$ then $flag := \text{true}$.

5) For all base edges $e = (u, w)$ annotated by an assignment statement $x := e$ with $x \in I[w]$:

5.1) If $e$ is a composite expression then $flag := \text{true}$;

5.2) If $e$ is a constant expression then

if $val = \text{unset}$ then $val := e$ else if $val \neq e$ then $flag := \text{true}$.

6) If $flag$ then output “no” else output “yes”.

---

**Figure 9.2:** An algorithm that detects copy constants in parallel programs.

Intuitively, $I[w]$ is the set of variables that can influence the value of $y$ at $v$ when some computation is at $w$. Clearly, in $I[w]$ dependences of bridging runs from $w$ to $v$ are considered. By solving the constraint system for bridging runs from Chapter 6 over the domain $(\text{AD, } \sqsubseteq)$ (Step 1), we can compute the dependence traces of bridging runs; they are given by the second component of the value $B^\#_v[w]$ that is computed. From the dependence traces we can read off the dependences by Proposition 8.5 and hence determine $I[w]$ (Step 2).

The rest of the algorithm is based on the following observation: variable $y$ is not a copy constant at $v$ if and only if one of the following is true:

a) there is a variable $x$ the initial value of which can influence $y$ at $v$;

b) there is a base edge $e = (u, w)$ annotated by an assignment $x := e$ with a composite expression $e$ on the right hand side such that $x$’s value at $w$ can influence $y$’s value at $v$;
c) there are two distinct base edges \( e = (u, w) \) and \( e' = (u', w') \) each of them 
annotated by a constant assignment \( x := k and x' := k' \), respectively, such 
that both \( x \) at \( w \) and \( x' \) at \( w' \) can influence \( y \) at \( v \) and \( k \neq k' \).

In Step 3-6 we check whether one of these conditions is true. We use a Boolean 
variable \( \text{flag} \) that is initialized to false and is set to true once we encounter a 
reason for \( y \) not being a copy constant at \( v \). Step 4 tests whether condition 
a) holds true. Step 5 is concerned with conditions b) and c). Each base edge 
is examined in turn. Step 5.1 tests whether b) holds. In order to check c), we 
memorize in a variable \( \text{val} \) the value of the constant assignment that can influence 
\( y \) at \( w \) encountered first. In order to check c) we simply compare the value of 
constant assignments encountered later with the value memorized in \( \text{val} \). Variable 
\( \text{val} \) is initialized with a special value \( \text{unset} \) that indicates that we have not seen 
a constant assignment so far. Finally, Step 6 outputs the answer.

Of course we could stop the algorithm immediately, once the flag is set to 
true. Moreover, we can output the value stored in \( \text{val} \) as additional information, 
if we have identified \( y \) as a copy constant at \( v \). It is the value guaranteed for \( y \) 
at \( v \). It may happen that \( \text{val} \) has still the value \( \text{unset} \); this indicates that \( v \) is an 
unreachable program point.

We conclude:

**Theorem 9.2** Algorithm 9.1 solves the interprocedural copy-constant detection 
problem in parallel flow graphs relative to non-atomic interpretation of base statements.

### 9.2 Faint-Code Elimination

A variable \( x \) is *live* at a program point \( p \) if there is a run from \( p \) to the end of 
the program on which \( x \) is used before it is overwritten. By referring to [23], 
Horwitz et. al. [28] define a variable \( x \) as *truly live* at a program point \( p \) if there is 
a run from \( p \) to the end of the program on which \( x \) is used in a truly live context 
before being defined, where a truly live context means: in a predicate, or in a 
call to a library routine, or in an expression whose value is assigned to a truly 
live variable. True liveness can be seen as a refinement of the ordinary liveness 
property. We call a use of a variable \( x \) in a predicate or call to a library routine 
a *relevant use* of \( x \).

Assignments to variables that are not truly live at the program point immediately 
after the assignment are called *faint*. Intuitively, faint assignments can not 
influence any predicate in the program or call of a library routine. Thus, they 
cannot influence the observable behavior of the program (except of producing 
run-time errors) and may safely be eliminated from the program. This is called 
*faint-code elimination*. 
Algorithm 9.2

**Input:** A parallel flow graph \( \pi \) as defined in Chapter 6; a mapping \( R : N \rightarrow 2^X \) that associates each program point \( u \) with the set of variables relevant at \( u \).

**Output:** An updated edge annotation \( A_{\text{new}} \) of the parallel flow graph in which faint code is eliminated.

**Method:**

1) Initialize the new annotation of flow graph edges: \( A_{\text{new}} := A \).
2) For each base edge \( e \in \text{Base} \): \( A_{\text{new}}[e] := \text{skip} \).
3) For each \( v \in N \) with \( R(v) \neq \emptyset \):
   3.1) Compute—by standard fixpoint iteration—the least solution over domain \((AD, \sqsubseteq)\) of the constraint system for bridging runs to program point \( v \); this gives us a value \( B^#_e[u] \) for each program point \( u \).
   3.2) Set \( I[w] := \{ x \mid \exists y \in R(v) : (1, \langle (x, y) \rangle, 1) \in B^#_e[u] \} \) for each program point \( w \in N \).
   3.3) For each base edge \( e = (\_ w) \in \text{Base} \) with \( A[e] = (x := t) \):
       if \( x \in I[w] \) then \( A_{\text{new}}[e] := (x := t) \).
4) Output the new edge annotation \( A_{\text{new}} \).

Figure 9.3: An algorithm that eliminates faint code in parallel programs.

Faint-code elimination is a stronger form of the classic transformation of dead-code elimination [51]. Indeed, any assignment that is dead is also faint but not vice versa. The paradigmatic example is shown in Fig. 9.4. Thus, faint-code elimination in general can eliminate more code from a program.

We present now an algorithm for faint-code elimination in parallel programs. Faint-code elimination is based on information about the relevant uses of variables. Typically, this information is derived from the output and branching statements in the program: each output statement that refers to a variable \( x \) means that the value of \( x \) is relevant at the program point just before the output statement. Similarly, a branching statement guarded by a condition \( b \) means that all variables occurring in \( b \) are relevant at the program point just before the branching statement. As our view of the source program, a parallel flow graph, is an abstraction of the actual program in which I/O statements as well as the condi-
Figure 9.4: An example of an assignment that is faint but not dead. The value computed by \( x := x + 1 \) in the loop is immediately overwritten after the loop and thus never used in a relevant context. Hence \( x := x + 1 \) is faint. However, it is not dead because \( x \) is potentially (non-relevantly) used by the same statement in the next iteration of the loop.

In the abstract flow graph view of the program this statement gives rise to a skip edge \( e = (u, v) \). Then both \( x \) and \( y \) are relevant at \( u \), hence \( R(u) = \{x, y\} \).

Similarly, we might find a branching statement, e.g.,

\[
\text{if } (z > x \cdot y) \text{ then } \{\ldots\} \text{ else } \{\ldots\}
\]

in the source program. In the abstract flow graph view of the program this if-statement gives rise to two skip-edges \((u, v)\) and \((u, w)\); \( u \) is the start node for the flow graph for the whole if-statement; at \( v \) the flow graph for the then part and at \( w \) the flow graph for the else part is found. In this case, we have \( R(u) = \{x, y, z\} \).

Algorithm 9.2 in Fig. 9.3 reads a parallel flow graph and a mapping \( R : N \to 2^X \) as described above. Based on this information it calculates an updated version of the edge annotation mapping of the given flow graph in which faint code is eliminated. More precisely, faint instances of base statements are replaced by skip.

First the new edge annotation mapping is initialized by the original edge annotation (Step 1) and all annotations of base edges are removed, i.e. replaced by skip (Step 2). The rest of the algorithm restores the original edge annotation
for the base edges that are not faint. The algorithm is based on the simple idea that an instance of a base statement is not faint if and only if it can influence a relevant value.

We explore all program points $v$ at which at least one variable is relevant and restore the base edges that perform a computation that can influence a variable $y$ that is relevant at $v$ (Step 3). For this purpose we calculate in Steps 3.1 and 3.2 for all program points $w$ the set

$$I[w] = \{ x \mid e_{Main} \Rightarrow c_w \Rightarrow c_v, At_w(c_w), At_v(c_v), \exists y \in R(v) : \hat{r} \text{ med. dep.} (x, y) \}.$$ 

Intuitively, $I[w]$ contains the variables that can influence the value of a relevant variable $y$ at $v$ when some computation is at $w$. The computation is analogous to the one of the similar set $I[w]$ in Algorithm 9.1; therefore, we omit a detailed explanation. Step 3.3 restores the annotation of those base edges that assign to a variable that can influence a relevant variable at $v$ from the target node of the base edge. Finally, Step 4 outputs the computed new edge annotation mapping.

We conclude:

**Theorem 9.4** Algorithm 9.2 solves the interprocedural faint-code elimination problem in parallel flow graphs relative to non-atomic interpretation of base statements.

## 9.3 Run-Time

The goal of this section is to analyze the asymptotic run-time of the algorithms from the previous sections. We do not determine very sharp estimates but are satisfied with showing that the algorithms run in time exponential in the number of program variables, $|X|$ and polynomial in the size of the parallel flow graph. The latter is measured by the parameters $|N|$, the number of program points, $|E|$, the number of edges, and $|Proc|$, the number of procedures.

In both algorithms the bulk of the work is done during the least fixpoint computation(s) for the constraint system(s) for bridging runs over the domain $(AD, \sqsubseteq)$. Let us, first of all, determine an asymptotic bound for the complexity of such a fixpoint computation. As we are heading only for a rough bound, we can assume that the least fixpoint is computed na"ively by standard fixpoint iteration: starting from an assignment of the bottom value to each variable appearing in the constraint system we iteratively determine a new assignment to the variables by re-evaluating all constraints until convergence, i.e. until we observe no further modification. Of course the asymptotic complexity of this naive fixpoint algorithm is bounded by the product of the maximal number of iterations and the maximal cost of a single step.

In each iteration except of the last one, at least one constraint variable must change its value. It is well-known that the values assigned to a constraint variable can only increase during fixpoint iteration. Therefore, for each constraint
variable the value can change at most $O(\lvert X^{2\lvert X \rvert + 3} \rvert)$ times, because $O(\lvert X^{2\lvert X \rvert + 3} \rvert)$ is a bound for the height of $\mathbf{AD}$ by Lemma 8.22. Moreover, it is a simple counting exercise to show that the complete constraint system for bridging runs (it comprises the constraint systems for same-level runs, inverse same-level runs, reaching runs, etc.) has $O(\lvert \text{Proc} \rvert \cdot \lvert N \rvert) \cdot \lvert X^{2\lvert X \rvert + 3} \rvert)$ constraint variables.\footnote{This asymptotic bound holds in the special case where ASS1 and ASS2 are true as well as in the general case.} Thus, we can have at most $O(\lvert \text{Proc} \rvert \cdot \lvert N \rvert \cdot \lvert X^{2\lvert X \rvert + 3} \rvert)$ iterations. This clearly is $O(\lvert \text{Proc} \rvert \cdot \lvert N \rvert \cdot 2^{p_0(\lvert X \rvert)})$ for some polynomial $p_0(x)$ in $x$, because $x^{2x^{3} + 3} = 2^{\log_2(x)(2x^{3} + 3)} \leq 2^{2x^{3} + 3}$.

Let us now bound the costs of a single iteration. In each iteration we must reevaluate all constraints. As the number of operations in a single constraint is bounded, we can get an asymptotic bound for the costs of a complete reevaluation of all constraints by multiplying a bound for the number of constraints with a bound for the maximal costs of a single operation. It is again a simple counting exercise to show that the complete constraint system for bridging runs has $O(\lvert N \rvert \cdot \lvert E \rvert)$ constraints.\footnote{Again this asymptotic bound holds for both the special and the general case.} From Lemma 8.47 we know that all operations can be computed in time $O(2^{p_1(\lvert X \rvert)})$ for some polynomial $p_1(x)$. Hence the cost of a single iteration is $O(\lvert N \rvert \cdot \lvert E \rvert \cdot 2^{p_1(\lvert X \rvert)})$.

Summarizing, the constraint system for bridging runs can be evaluated over domain $(\mathbf{AD}, \sqsubseteq)$ in time $O(\lvert \text{Proc} \rvert \cdot \lvert N \rvert \cdot 2^{p_0(\lvert X \rvert)} \cdot \lvert N \rvert \cdot \lvert E \rvert \cdot 2^{p_1(\lvert X \rvert)}) = O(\lvert \text{Proc} \rvert \cdot \lvert N \rvert^2 \cdot \lvert E \rvert \cdot 2^{p(\lvert X \rvert)})$ for $p(x) = p_0(x) + p_1(x)$. Let us fix this result for later reference as a lemma.

**Lemma 9.5** The constraint system for bridging runs can be evaluated over domain $(\mathbf{AD}, \sqsubseteq)$ in time $O(\lvert \text{Proc} \rvert \cdot \lvert N \rvert^2 \cdot \lvert E \rvert \cdot 2^{p(\lvert X \rvert)})$, where $p(x)$ is a polynomial. \hfill $\Box$

Let us now turn attention to the algorithms. Clearly, in the copy-constant detection algorithm, Algorithm 9.1, the bulk of the work is done in Step 1 such that the time taken for Step 1 majorizes the time taken for the other steps. Hence this algorithm runs in time $O(\lvert \text{Proc} \rvert \cdot \lvert N \rvert^2 \cdot \lvert E \rvert \cdot 2^{p(\lvert X \rvert)})$ by Lemma 9.5.

In the faint-code elimination algorithm, Algorithm 9.2, the work performed in Step 3.1 majorizes the work done in the other steps. Step 3.1 is executed at most $\lvert N \rvert$ times. Consequently, Algorithm 9.2 runs in time $O(\lvert \text{Proc} \rvert \cdot \lvert N \rvert^3 \cdot \lvert E \rvert \cdot 2^{p(\lvert X \rvert)})$.

Clearly, only those program variables are of interest in the algorithms that appear in the parallel flow graph. We can thus assume without loss of generality, that all program variables in $X$ appear in the parallel flow graph. As the latter constitutes part of the input to all algorithm, the input size cannot be smaller than the size of $X$. Obviously, the same holds for $\text{Proc}$, $N$, and $E$ such that the size of the input clearly bounds all the parameters appearing in above run-time estimations. Hence all algorithms run in time exponential in the size of the input.
Theorem 9.6 Algorithms 9.1 and 9.2 run in exponential time. More precisely, Algorithm 9.1 runs in time \( O(|\text{Proc}| \cdot |N|^2 \cdot |E| \cdot 2^{|X|}) \) and Algorithm 9.2 in time \( O(|\text{Proc}| \cdot |N|^3 \cdot |E| \cdot 2^{|X|}) \).

Corollary 9.7 If base statements are interpreted non-atomically, the following two problems can be solved interprocedurally in parallel flow graphs in exponential time: (1) copy-constant detection and (2) faint-code elimination.

9.4 Conclusion

We have shown in this chapter that we can detect copy constants and eliminate faint code in parallel flow graphs in exponential time, if we do not assume that base statements execute atomically. This should be contrasted to the result that all these problems are undecidable if assignment statements are assumed to execute atomically (Chapter 5). So, the (unrealistic) idealization from program verification “atomic execution of assignment statements” that presumably simplifies matters actually increases the difficulty of these problems from the point of view of program analysis: amazingly these problems become more tractable if we adopt a less idealized, more realistic view of execution!

These results raise the question whether there are also efficient algorithms for these problems. Sadly, the answer to this question is ‘no’, unless \( \text{P}=\text{NP} \), as we show in the next chapter.
Chapter 10

Complexity in the Non-atomic Scenario

In the previous chapter, we have seen that we can detect all copy constants and eliminate faint code completely in parallel programs, if we abandon the assumption that base statements execute atomically. The presented algorithms run in exponential time, which raises the question whether there are also efficient algorithms for these problems. In this chapter we show that the answer is ‘no’, unless P=NP. In the conclusions of this thesis, Chapter 11, we sketch possible remedies and discuss directions of future research that may still lead to algorithms of practical interest.

The hardness proofs from Chapter 5 rely on well-directed re-initialization of variables in order to ensure that runs which do not correspond to behavior to be simulated do not contribute to propagation. The example in Section 7.2 indicates that this technique does not work, if the assumption of atomic execution of base statements is abandoned. This also follows from the fact that the above analysis problems become decidable, which trivially implies that the un-decidability proofs cannot be valid any more.

In Section 10.1 we exhibit a co-NP-hardness proof by means of a reduction from the well-known SAT-problem [10, 60] that applies to both flow analysis problems. We have first presented this reduction in [53] where atomic execution of base statement has been assumed, but it remains valid if this assumption is abandoned. Unlike the reductions in Chapter 5, it only relies on active propagation along copying assignments but not on well-directed re-initialization.

The hardness proof constructs loop-free programs and it is easy to see that the co-NP lower bound is indeed sharp for loop-free programs. We have not yet been able to fully characterize the complexity for the other classes: the general intraprocedural problem and the interprocedural problem. Up to now we have the EXPTIME upper bound through the algorithms from Chapter 9 and the NP lower bound through the SAT reduction from Section 10.1. A natural idea for an NP-easiness proof would be to show that non-constancy and non-faintness is
always witnessed by runs of polynomial length. We show in Section 10.2 that this idea does not work. Specifically, we exhibit a family of programs in which the length of the shortest witnessing runs is exponential in the program size. This justifies the conjecture that the general inprocedural problem does not belong to NP, i.e., cannot be solved by a non-deterministic algorithm that runs in polynomial time.

For ease of presentation we represent parallel programs in this chapter, like in Chapter 5, by syntactic programs rather than flow graphs.

## 10.1 The SAT-Reduction

We now describe the SAT reduction. An instance of SAT is a conjunction $c_1 \land \ldots \land c_k$ of clauses $c_1, \ldots, c_k$. Each clause is a disjunction of literals; a literal $l$ is either a variable $x$ or a negated variable $\overline{x}$, where $x$ ranges over some set of variables $X$. We write $X = \{ \overline{x_1}, \ldots, \overline{x_n} \}$ for the set of negated variables. It is straightforward to define when a truth assignment $T : X \rightarrow \mathbb{B}$, where $\mathbb{B} = \{\text{tt}, \text{ff}\}$ is the set of truth values, satisfies $c_1 \land \ldots \land c_k$. The SAT problem asks us to decide for each instance $c_1 \land \ldots \land c_k$ whether there is a satisfying truth assignment or not.

From a given SAT instance $c_1 \land \ldots \land c_k$ with $k$ clauses over $n$ variables $X = \{x_1, \ldots, x_n\}$ we construct a loop-free parallel program. In the program we use $k + 1$ variables $z_0, z_1, \ldots, z_k$. Intuitively, $z_i$ is related to clause $c_i$ for $1 \leq i \leq k$; $z_0$ is an extra variable.

For each literal $l \in X \cup X$ we define a statement $\pi_l$. $\pi_l$ consists of a sequential composition of assignments of the form $z_i := z_{i-1}$ in increasing order of $i$. The assignment $z_i := z_{i-1}$ is in $\pi_l$ if and only if the literal $l$ makes clause $i$ true. Formally, $\pi_l = \pi_l^k$, where

$$
\begin{align*}
\pi_l^0 & \overset{\text{def}}{=} \text{skip} \\
\pi_i^l & \overset{\text{def}}{=} \begin{cases} 
\pi_{i-1}^l ; z_i := z_{i-1}, & \text{if clause } c_i \text{ contains } l \\
\pi_{i-1}^l, & \text{if clause } c_i \text{ does not contain } l
\end{cases}
\end{align*}
$$

for $i = 1, \ldots, k$. Now, consider the following program $\pi$:

```plaintext
procedure Main;
  z_0 := 1;
  z_1 := 0; \ldots; z_k := 0;
  [(\pi_{x_1} \sqcap \pi_{\overline{x}}) \sqcap \cdots \sqcap (\pi_{x_n} \sqcap \pi_{\overline{x}})];
  (z_k := 0 \sqcap \text{skip});
  \text{write}(z_k)
end
```

This reduction has first been presented in [53]
Clearly, $\pi$ can be constructed from the given SAT instance $c_1 \land \ldots \land c_k$ in polynomial time or logarithmic space.

It is not hard to see that the value 1 from the initialization of $z_0$ can be propagated to the final write statement if and only if the given SAT instance is satisfiable:

"If": On the one hand, we can construct from a satisfying truth assignment $T : X \to B$ a run that propagates $z_0$’s initialization to the write-statement. In each parallel component $\pi_{x_i} \land \pi_T$ we choose the left branch $\pi_{x_i}$ if $T(x_i) = \text{tt}$ and the right branch $\pi_T$ otherwise. As $T$ is a satisfying truth assignment, there will be, for any $i \in \{1, \ldots, k\}$, at least one assignment $z_i := z_{i-1}$ in one of the chosen branches. We interleave the branches now in such a way that the assignment(s) to $z_1$ are executed first, followed by the assignment(s) to $z_2$ etc. This results in a propagating run.

"Only if": On the other hand, a run can propagate the value with which $z_0$ is initialized to the write-statement only via copying it from $z_0$ to $z_1$, from $z_1$ to $z_2$ etc., because all assignments have the form $z_i := z_{i-1}$. Such a run must thus contain the assignment $z_i := z_{i-1}$ for all $i = 1, \ldots, k$. But from the way in which the non-deterministic choices are resolved in such a run we can easily construct a satisfying truth assignment.

The arguments for both directions hold independently from the atomicity assumption for assignment statements.

**Example 10.1** Fig. 10.1 shows an example clause and program for illustration. Assignments to different variables are shown on different levels. Intuitively a satisfying truth assignment corresponds to a way of resolving the non-deterministic choices in the three threads such that at each level at least one assignment is present in one of the chosen branches. This is the case if and only if the value 1 from $z_0$’s initialization may propagate to the write instruction.

It is not hard to infer from this propagation property that the given SAT instance is satisfiable if and only if any of the following two conditions holds:

1. $z_0 := 1$ is not a faint assignment.
2. $z_k$ is not a copy constant at the write statement.

The second point deserves additional explanation. Observe first that $z_k$ can hold only 0 or 1 at the write-statement because all variables are initialized by 0 or 1 and the other assignments only copy these values. Clearly, due to the non-deterministic choice just before the write-statement, $z_k$ may hold 0 finally. Thus, $z_k$ is a constant at the write-statement if and only if it cannot hold 1 there. The latter obviously holds if and only if the initialization value of $z_0$ cannot be propagated.
The program constructed in the above reduction is loop-free and does not employ procedures. Therefore, the reduction already applies to the intraprocedural problems for loop-free programs. It is easy to see that the problems can also be solved in non-deterministic polynomial time for loop-free programs: imagine non-deterministic algorithms that guess two runs that witnesses non-constancy or a single run that witnesses non-faintness, respectively. Each of these runs can visit any program point at most once because the program is loop-free. Hence it can be guessed even in time linear in the program size.

These considerations prove:

**Theorem 10.2** Independently of the atomicity assumption for base statements, detecting copy constants and detecting faint code in loop-free parallel programs are co-NP-complete problems.

**Corollary 10.3** Independently of the atomicity assumption, detecting copy constants, and detecting faint code are co-NP-hard problems in arbitrary parallel programs.
10.2 Towards Stricter Lower Bounds

A natural question is whether the lower bound provided by Corollary 10.3 for the three flow analysis problems, NP-hardness, is sharp, i.e., whether there are non-deterministic algorithms that run in polynomial time and solve the general intraprocedural (or even interprocedural) version of one (or both) of these problems. While we have not yet been able to settle this complexity question, we have achieved some progress towards an answer.

A natural approach for showing NP-easiness would be to exhibit a proof that shortest propagating runs are always of a length polynomial in the program size. This would guarantee that non-deterministic algorithms that guess runs that witness non-constancy or non-faintness would run in polynomial time.

At first glance this approach seems promising, at least for the intraprocedural problem which has a fixed process architecture. One is tempted to believe that each assignment instance \( x := e \) in the program can be used at most once for propagation in a shortest propagating run: if it is used twice in a propagating run \( r \) it seems possible to shorten this run. The intuition is that the thread \( T \) that contains this assignment instance \( x := e \) could store the value to be propagated in a virtual variable \( v \) when \( x := e \) is reached the first time in \( r \); then it could wait until the environment has evolved to its state when \( x := e \) is reached the second time in \( r \). As virtual variables are private to \( P \) the evolution of the environment cannot affect the stored value. This reasoning seems similar to the intuition underlying Lemma 8.40 that is crucial for the completeness of the abstract interleaving operator \( \otimes^\# \).

This intuition, however, is wrong as we show in Section 10.2.1. Specifically, we present an example program in which any propagating run must necessarily use a certain instance of an assignment twice. In Section 10.2.2 we exploit the construction of this example to exhibit a family of programs in which the length of shortest propagating runs grows exponentially with the program size. This proves that a non-deterministic algorithms that guesses witnessing runs is doomed to run in time exponential in the input size. While this does not rule out the possibility that small certificates other than witnessing runs exist, it nevertheless justifies the conjecture that the two flow analysis problems that accompany us through this thesis probably do not belong to the class NP but that their complexity is higher. It is an open problem, whether the technique of these examples can be used to show better lower bounds than NP-hardness, e.g., PSPACE-hardness.

What is the error in the argument sketched above? It is that the thread \( T \) can prevent the environment from certain evolutions by waiting after it has stored the value to virtual variable \( v \). For constructing programs in which this happens (like the ones shown in the remainder of this section), we can exploit the causality inherent in sequential and parallel composition and looping. Most importantly, we can exploit that the parallel composition operation synchronizes termination of the component threads.
10.2.1 Assignment Statements that Propagate Twice

Recall that a run $r$ is said to mediate the dependence $(x, y)$, $r \vdash (x, y)$ for short, if there are variables $a_0, \ldots, a_l$, $l > 0$, expressions $e_1, \ldots, e_l$, and (sub-) runs $r_0, \ldots, r_l$ such that

1. $a_0 = x, a_l = y$;
2. $e_i$ contains $a_{i-1}$ for $i = 1, \ldots, l$;
3. $r = r_0 \cdot \langle a_1 := e_1 \rangle \cdot r_1 \cdot \langle a_2 := e_2 \rangle \cdot r_2 \cdot \ldots \cdot \langle a_l := e_l \rangle \cdot r_l$; and
4. $r_i$ is transparent for $a_i$ for $i = 0, \ldots, l$.

If this is the case, we say that the run $r$ propagates from $x$ to $y$ via the assignments $a_i := e_i$. When $r$ is a run of a program $\pi$, the assignments with a program variable on their left hand side correspond to certain assignment statements in $\pi$. Then we say that the run propagates via these assignment statements.

We now present a program $\pi$ that can mediate the dependence $(a, c)$, but in which any run that mediates $(a, c)$ must use a certain assignment instance twice. Consider the following program $\pi$:

$$\text{loop } \left( \begin{array}{l}
  x := b; \\
  c := y; \\
  x := a; \\
  b := y; \\
  x := 0
\end{array} \right) \parallel \left( \begin{array}{l}
  y := x; \\
  y := 0
\end{array} \right) \text{ end.}$$

Program $\pi$ can mediate the dependence $(a, c)$ even when assignments execute atomically (and hence also when they execute non-atomically): by iterating the loop twice and interleaving the two components of the parallel processes appropriately, we see that it has the run

1. Iteration: $x := b, c := y, x := a, y := x, b := y, x := 0, y := 0,$
2. Iteration: $x := b, y := x, c := y, x := a, b := y, x := 0, y := 0$

This run mediates the dependence $(a, c)$ via the assignments printed in bold face. The interesting point of this example is that—even when we assume non-atomic execution of assignments—there is no run that mediates this dependence without copying via the assignment $y := x$ in the second parallel component twice. In order to see this, consider the following: as variable $a$ is read only by the assignment $x := a$, a propagating run must use this assignment for propagation in some iteration of the loop, say in the $k$'th iteration. Before this iteration of the loop ends, $x$ must be further propagated, because otherwise propagation is prohibited by the execution of $x := 0$. This can only happen in the second thread by means of $y := x$. Again, in order to successfully proceed with the
propagation, $y$ must be propagated before the end of the iteration of the loop, because otherwise $y := 0$ prohibits further propagation. Hence, $b := y$ must be executed before the end of the $k$'th loop iteration after complete execution of $y := x$. After the $k$th loop iteration, the value in $b$ must be further propagated to $c$, which requires a second use of $y := x$.

Note that this example exploits the synchronous termination of the parallel composition operator as well as the causality inherent in sequential composition.

### 10.2.2 Propagating Runs of Exponential Length

By iterating the technique of the previous example we can construct a family of processes in which exponentially long runs are necessary to mediate a particular dependence.

We inductively define processes $P_i$, $i \geq 0$. These processes have the ability to propagate from a variable $a_i$ to a variable $c_i$. We will show below that the shortest runs that do so have length $\Omega(2^i)$.

$i = 0$: Process $P_0$ is defined as $c_0 := a_0$. It plays the role of the instruction $y := x$ in the previous example; $a_0$ corresponds to $x$ and $c_0$ to $y$.

$i > 0$: For $i > 0$, the process $P_i$ relies on the ability of $P_{i-1}$ to propagate from $a_{i-1}$ to $c_{i-1}$. The construction from Section 10.2.1 is used to enforce that $P_{i-1}$ has to contribute two runs that propagate from $a_{i-1}$ to $c_{i-1}$ in any run of $P_i$ that propagates from $a_i$ to $c_i$. For this purpose an intermediate variable $b_i$ is used. This is the definition of $P_i$:

\[
\text{loop} \left( \begin{array}{l}
    a_{i-1} := b_i; \\
    c_i := c_{i-1}; \\
    a_{i-1} := a_i; \\
    b_i := c_{i-1}; \\
    a_{i-1} := 0
\end{array} \right) \| \left( \begin{array}{l}
    P_{i-1}; \\
    c_{i-1} := 0
\end{array} \right) \text{ end}; b_i := 0
\]

Let us now prove by induction on $i$ that process $P_i$ has a run that propagates from $a_i$ to $c_i$ and that (for $i > 0$) any run of $P_i$ that does so must include at least two runs of $P_{i-1}$ that propagate from $a_{i-1}$ to $c_{i-1}$. This proves the $\Omega(2^i)$ claim for the length of shortest propagating runs. In order to enable an inductive proof, the following additional property is proved simultaneously: any run of $P_i$ finally kills all variables that are assigned to except of $c_i$, more precisely: if a run $r$ of $P_i$ can be written as $r = r_0 \cdot \langle x := e \rangle \cdot r_1$ with $x \neq c_i$ and $e \neq 0$, then $r_1$ can be written as $r_1 = r_2 \cdot \langle x := 0 \rangle \cdot r_3$.

For $P_0$ these properties are trivial. So suppose $i > 0$ and assume that the properties are valid for $P_{i-1}$. Let $r$ be a shortest run of $P_{i-1}$ with $r \vdash (a_{i-1}, c_{i-1})$. 

Then we can define a run $s$ of $P_i$ (with atomically executed assignments) in analogy to the run considered in the previous example:

1. Iteration: $a_{i-1} := b_i, c_i := c_{i-1}, a_{i-1} := a_i, r, b_i := c_{i-1}, a_{i-1} := 0, c_{i-1} := 0,$
2. Iteration: $a_{i-1} := b_i, r, c_i := c_{i-1}, a_{i-1} := a_i, b_i := c_{i-1}, a_{i-1} := 0, c_{i-1} := 0,$

After loop: $b_i := 0$

The parts written in bold face witness that $s \vdash (a_i, c_i)$ and, obviously, this run contains $r$ twice.

In order to see that any run $s$ of $P_i$ with $s \vdash (a_i, c_i)$ necessarily contains two runs of $P_{i-1}$, we argue similar to Section 10.2.1: as variable $a_i$ is read only by the assignment $a_{i-1} := a_i$, a propagating run must use this assignment for propagation in some iteration of the loop, say in the $k$th iteration. Before the $k$th iteration of the loop ends, $a_{i-1}$ must be read, because otherwise propagation is prohibited by the execution of $a_{i-1} := 0$. This can only happen in the second thread in a run $r$ of $P_{i-1}$. By the induction hypothesis this run kills all variables except $c_{i-1}$ finally, and $c_{i-1}$ is also killed explicitly after the execution of $P_{i-1}$ before the $k$th iteration of the loop ends. Thus, successful propagation requires that $r$ is a run that propagates to $c_{i-1}$ and that afterwards $b_i := c_{i-1}$ is executed. In order to propagate from $b_i$ to $c_i$ in a later iteration of the loop, a further run of $P_{i-1}$ that propagates from $a_{i-1}$ to $c_{i-1}$ is needed.

That all runs of $P_i$ kill all the variables they assign to except $c_i$ is easy to see from the corresponding property for $P_{i-1}$ and the places of the assignments $a_{i-1} := 0, c_{i-1} := 0,$ and $b_i := 0$ in $P_i$.

These considerations justify the following conjecture.

**Conjecture 10.4** For parallel programs, the intraprocedural copy-constant detection problem does not belong to co-NP. The same holds for faint-code elimination.

### 10.3 Summary

In this chapter we have seen that both detecting copy constants and eliminating faint code are intractable problems, even if the assumption that base statements execute atomically is abandoned. Both problems have been shown to be co-NP-hard by means of a reduction from the SAT problem. Unlike the reductions in [55], this reduction applies under the assumption that assignments execute atomically as well as when this assumption is abandoned. Moreover, we have exhibited a family of example programs in which the length of shortest propagating runs is exponential in the program size. This indicates that the lower bound, NP-hardness, probably can be improved for the general intraprocedural problem as well as the interprocedural problem.
Chapter 11

Conclusion

For fundamental recursion-theoretic reasons, program analyzers are doomed to give only approximate answers. By applying abstractions to programs, we can come to precisely defined, weaker analysis problems that can be solved exactly. By classifying such problems with the means provided by the theory of computational complexity, we hope to shed light on the trade-off between efficiency and precision for approximate analyzers and to uncover potential for more precise analysis algorithms.

In this thesis we studied various version of the constant propagation problem. More specifically, our contributions are the following:

1. We characterized the complexity of constant detection for a three-dimensional taxonomy of constants in sequential flow graphs that work on integer variables almost completely. The first dimension selects a subset of expressions that are interpreted precisely. The second dimension distinguishes between must- and may-constants; may-constants appear in two variations: single- and multiple-valued. In the third dimension we distinguish between programs with or without loops.

2. We showed that detection of copy constants in parallel programs is undecidable, PSPACE-complete, and NP-complete if we consider programs with procedures, without procedures, and without loops, respectively. These proofs rely on the standard assumption that base statements execute atomically. They reveal fundamental limits for precise analysis of parallel programs.

3. We then abandoned this atomic execution assumption. Surprisingly, this makes copy-constant detection decidable for programs with procedures although it remains intractable (co-NP-hard). Similar statements can be made for faint-code elimination. In order to show decidability we exhibited a precise abstract interpretation of sets of runs (program executions). The worst-case running time of this algorithm is exponential in the number of
global variables but polynomial in the parameters describing the program size.

From a practical perspective, our most interesting findings concern potential for the construction of algorithms. In the sequential case, we find that polynomial constants are decidable and that Presburger constants can even be detected in polynomial time. In the parallel case we could show that problems that are undecidable under the standard idealization of atomic execution are in the reach of algorithmic techniques if more realistic atomicity assumptions are adopted. This in particular holds for the fundamental problem of exact dependence analysis. While further work is necessary to construct algorithms that are efficient enough to be of practical use, our findings open up potential for interesting future work.

The worst-case running-time of the algorithms in Chapter 9 is exponential. We cannot even hope that they would perform well in practice because already the elementary operations are expensive, in particular the abstract interleaving operator. Nevertheless, we believe that refinements of the technique underlying dependence traces can lead to practically interesting algorithms with acceptable performance and superior precision. Let us discuss possible targets for improvements.

While the run-time of the algorithms is exponential in the number of program variables, it is polynomial in the program size; cf. Theorem 9.6. Hence, if the number of program variables is bounded, they are polynomial-time algorithms. For a practical algorithm it is thus essential to keep the number of the variables that are used in dependence trace construction small. In order to keep the technical treatment manageable, we do not distinguish between local and global variables of threads and procedures in the current exposition. All variables are global and all of them are visible to each thread. Therefore, we must include all variables into the precise interference analysis provided by dependence traces. In practice, however, most variables are local to threads and there are only a few global variables on which interference can happen. A practical algorithm should take advantage of the distinction between local and global variables. The idea is to devise a combined analysis that uses a cheap sequential technique for propagation via local variables and applies the expensive interference reasoning via dependence traces only to global variables. Analysis with respect to such a domain promises to be exponential only in the number of global variables, which is probably small in practice.

We should also strive for a compact representation of the values in the abstract domain AD. Each value comprises a set of variables T and an antichain of short dependence traces D. Set T may straightforwardly be represented by a bitvector. It is less clear, however, what is an adequate representation for D. Storing all the dependence traces in D, e.g., as a linked list, is probably not a good solution, because D can be large and there is much redundancy. The run \langle b := a, d := c, f := e \rangle, for instance, has the dependence trace
\[ \tau = (1, (a, b), (c, d), (e, f)), 1) \] but also the dependence traces \((1, ((a, b), (c, d)), 1), (1, ((a, b), 1)), \) and many others. In a certain sense the latter dependence traces are implied by \( \tau \) except of the transparency bits. We should use a representation that employs sharing to compactly represent all these dependence traces by a structure that is not much larger than \( \tau \) alone and that allows a cheaper computation of the composition operators.

In flow analysis of sequential programs we mostly propagate informative values through the program. In constant propagation, for instance, we use values of a lattice like the one in Figure 11.1. The dependence traces domain, however, is a rather poor and pure domain that treats interference in isolation. It is fitted to the computation of dependences only. Although it allows us to solve problems like copy-constant detection and faint-code elimination, the approach is indirect via bridging runs and involves even an iterated computation of dependence traces in the case of faint-code detection. It is interesting to invent and study more complex abstract domains that work with more informative values but rely on the idea of dependence traces to come to grips with interference. Ideally, such domains should be obtained by a modular extension of the dependence traces domain in order to isolate the interference-related reasoning from other semantic questions.

## 11.1 Future Research

Let us discuss some ideas for future research.

**Complete the hierarchy of constants.** An obvious target for future research are the two questions that remain open in the hierarchy of constants of Chapter 3: (1) we miss an upper bound for linear may-constants and (2) the upper and lower bound for polynomial must-constants do not coincide. Currently, we have decidability as an upper bound, as witnessed by the algorithm in Chapter 4, and PSPACE-hardness as a lower bound.
Investigate interprocedural hierarchy. It is interesting to study the hierarchy of constants in Chapter 3 also in sequential programs with procedures, i.e., the interprocedural problem. Particularly interesting are the questions whether Presburger constants can still be detected efficiently and whether polynomial constants are still decidable. In view of the negative results of Chapter 5 and 10 already for the weakest class of constants, copy constants, it is less interesting to generalize the results to parallel programs.

Implement Presburger and polynomial constant detection. On the practical side, we would like to implement the detection algorithms for Presburger and polynomial (must-)constants. In particular, it is interesting to evaluate how the algorithm for polynomial constants performs in practice.

Research towards more practical analysis algorithms. Concerning analysis of parallel programs the dependence traces domain proposed in this thesis is only a first step. We do not expect that the algorithms in Chapter 9 run satisfactorily in practice. We believe, however, that variants of the dependence traces techniques can well lead to algorithms with acceptable performance and superior precision. The next three points mention again the possible targets for improvements that have already been motivated and discussed in more detail above.

Take advantage of local variables. We would like to study algorithms that take advantage of the distinction between local and global variables. The expensive dependence traces technique should be applied only to global variables and local variables should be treated by much cheaper sequential techniques. The two propagation methods must be intertwined because both types of variables can contribute to propagate information to a certain point in the program. This may make the resulting algorithms rather complicated.

Represent antichains compactly. It is important to find a compact representation of antichains of dependence traces on which the abstract operations can be computed more efficiently than on an explicit representation.

Specialized domains. It is worth inventing domains that work with more informative values than dependence traces. With such domains it should be possible to perform, e.g., copy-constant detection by means of an abstraction of reaching runs rather than bridging runs. Thus it would reveal a closer connection to traditional analysis of sequential programs. Note, however, that in itself this does not imply a gain in efficiency (with respect to asymptotic run-time) because the constraint systems for reaching runs and bridging runs both have $O(|\text{Proc}| \cdot |N|)$ constraint variables and $O(|N| \cdot |E|)$ constraints.
More realistic programming languages. We should also consider application of the dependence traces technique to more realistic programming languages. In this thesis we studied the prototypic scenario of non-deterministic parallel flow graphs. Generalization to practical languages may lead to additional interesting problems.

Weak memory consistency models. Many modern implementations of multi-threaded programs provide only a weak memory consistency model that allows the implementation to change the order in which writes from one thread are observed in other threads [1, 63, 69]. The reason is that weaker assumptions about the memory enable a multitude of software and hardware optimizations. A weak memory consistency model is another reason besides non-atomicity, why the idealistic atomicity assumptions adopted in classic program verification and in our reductions in Chapter 5 are unrealistic. We conjecture that the dependence traces abstraction is sound and complete also under most if not all weak memory consistency models. This would emphasize the importance of dependence traces.
Bibliography


Appendix A

A Primer on Constraint-Based Program Analysis

Constraint-based program analysis provides a framework for developing analyses and arguing about their correctness and completeness. In this chapter we describe the idea underlying constraint-based program analysis. As a running example we use forward dataflow analysis in (non-procedural, sequential) flow graphs and consider constant propagation in particular.

Definition A.1 A flow graph is a structure \( G = (N, E, A, s, e) \) with node set \( N \), edge set \( E \subseteq N \times N \), a unique start node \( s \in N \), and a unique end node \( e \in N \). The mapping \( A : E \rightarrow \text{Asg} \cup \{\text{skip}\} \) associates each edge with an assignment statement \( x := e \in \text{Asg} \) or with the statement \( \text{skip} \). Edges represent the branching structure and the statements of a program, while nodes represent program points.

Program analysis problems are concerned with answering questions about certain sets of runs. A run is a sequence of atomic action; in a sequential context we can think of an action simply as an edge of the flow graph. A forward dataflow analysis, for instance, is concerned with the runs that reach program points from the start point of the program.

Definition A.2 Suppose \( G = (N, E, A, s, e) \) is a flow graph and \( w \in N \) is a program point. A run reaching \( w \) is a sequence of edges \( \langle e_1, \ldots, e_k \rangle \) with \( e_i = (u_i, v_i) \in E \) such that \( u_1 = s \), \( v_k = w \), and \( v_i = u_{i+1} \) for \( 1 \leq i < k \). In addition \( e \), the empty sequence, is a run reaching \( s \), the start node. We write \( R[u] \) for the set of runs reaching \( u \).

In constraint-based program analysis, we first set up a system of subset constraints that characterize the run sets of interest. Each constraint takes the form

\[
X_i \supseteq E(X_1, \ldots, X_k),
\]

181
where the variables \(X_i\) represent the run sets of interest plus, perhaps, some additional auxiliary run sets, and \(E(X_1, \ldots, X_k)\) is a term in these variables that denotes a monotonic mapping on run sets. We can have more than one constraint per variable. It follows from the Knaster-Tarski fixpoint theorem [76] that such a constraint system always has a smallest solution. We choose the constraints such that their smallest solution comprises just the run sets of interest. This is meant by saying that the constraint system characterizes the run sets. Throughout this thesis, we obey the following convention: run sets of interest are denoted by letters in sans serif font and the corresponding variables in constraint systems by the same letter in italic font.

In the constraint system for the reaching runs in flow graphs, for example, we have one variable \(R[u]\) for each program point \(u \in N\) that represents \(R[u]\) and no auxiliary variables. The characterizing constraint system for reaching runs has a special constraint for the start node

\[
[1] \quad R[s] \supseteq \{ \varepsilon \}
\]

and one constraint for each edge \(e = (u, v) \in E:\)

\[
[2] \quad R[v] \supseteq R[u] \cdot \{(u, v)\}.
\]

It is easy to see that the family \((R[u])_{u \in N}\) of sets of reaching runs satisfies all these constraints. It is moreover not hard to prove by induction on the length of runs that if \((F_u)_{u \in N}\) is a family of run sets that solves this constraint system, then any run that reaches \(u\) must be contained in \(F_u\). Together this implies that the smallest solution of this family of inequations over run sets is indeed the family of reaching run sets.

On the right hand side of constraints, certain run sets and operations on run sets are used. We may conceive the constraint system abstractly as a system over a certain signature \(\mathcal{Sig} = (C, O)\) consisting of a set of constants \(C\) and a set of operator \(O\), where each operator \(o\) has an associated arity \(\text{ar}(o) \in \mathbb{N}\).

In the constraint system for reaching runs, for instance, the signature consists of one constant \(c_e\) and a unary operator \(o_e\); the constraint system is this:\footnote{The reader may consider it more natural to read the second constraint as}

\[
[1] \quad R[s] \supseteq c_e
\]

\[
[2] \quad R[v] \supseteq o_e(R[u]), \quad \text{if } (u, v) \in E
\]

An interpretation \(I\) of the signature comprises a complete lattice \((\mathbb{D}, \sqsubseteq)\) and an assignment of a value \(I(c) \in \mathbb{D}\) to each constant \(c\) and an \((n\text{-ary})\) operations \(I(o): \mathbb{D}^n \to \mathbb{D}\) to each \(n\text{-ary}\) operator \(o\).
In the standard or concrete interpretation $I$, $\mathbb{D}$ is the power set of the set of runs, $\mathbb{D} = 2^{\text{Runs}}$, the order is subset inclusion, $\sqsubseteq = \subseteq$, and the interpretation of the constants and operators is as in the concrete constraint system. Thus, the least solution of the constraint-system comprises the run sets of interest. The concrete interpretation for the signature underlying the reaching runs in flow graphs, for instance, is this: $I(c_e) = \{e\} \in 2^{\text{Runs}}$ for constant $c_e$ and $I(o_e) = (\lambda R : R \cdot \{e\}) \in (2^{\text{Runs}} \to 2^{\text{Runs}})$ for unary operator $o_e$.

In constraint-based program analysis we obtain the analysis result by solving the constraint system over an abstract lattice ($\mathbb{D}^\#, \sqsubseteq^\#$) and a non-standard or abstract interpretation $I^\#$ of the constants and operators over ($\mathbb{D}^\#, \sqsubseteq^\#$). Typically, $\mathbb{D}^\#$ is a finite-height lattice such that the constraint system can be effectively solved by standard fixpoint iteration.

In order to specify a forward dataflow analysis, we choose a finite-height lattice ($\mathbb{D}^\#, \sqsubseteq^\#$) of dataflow facts and a value $d_0 \in \mathbb{D}^\#$ that represents the fact valid at the start of the program, and associate with each flow graph edge $e$ a monotonic transfer function $[e]^\# : \mathbb{D}^\# \rightarrow \mathbb{D}^\#$ that describes the effect of execution of edge $e$ on dataflow facts. Often the latter is given via the annotation of edges by statements. The members of $\mathbb{D}^\#$ represent, depending on the specific analysis, potential run-time properties of program points. The order, $\sqsubseteq^\#$, captures information contents: smaller values represents more accurate (more precise) information. In particular, the top value, $\top_{\mathbb{D}^\#}$, represents absence of information. Note that our interpretation of the order is dual to the traditional one.

**Example A.3 (Simple constant propagation)** Let us discuss so-called simple constant propagation. Here the lattice is $\mathbb{D}_\text{uc} = (\text{Var} \rightarrow (\text{Val} \cup \{\top\})) \cup \{\perp\}$, where $\text{Var}$ is the set of variables occurring in the program and $\text{Val}$ is the set from which variables draw their value at run-time. $\perp$ is an artificial bottom element that is added in order to make $\mathbb{D}_\text{uc}$ a complete lattice. The other values are abstract states $d : \text{Var} \rightarrow (\text{Val} \cup \{\top\})$. An abstract state assigns to each variable $x \in \text{Var}$ either a value $c \in \text{Val}$—in this case $x$ is guaranteed to be a constant of value $c$—or the special value $\top$—in this case $x$'s value at run-time is unknown.

The order on $\mathbb{D}_\text{uc}$ is defined as follows: $\perp \sqsubseteq d$ for all $d \in \mathbb{D}_\text{uc}$ and, for abstract states $d, d, d \sqsubseteq d'$ iff for all $x \in \text{Var}$, $d(x) = \perp$ or $d(x) = d'(x)$. That is, the order is the lift of the co-flat order on $\text{Val} \cup \{\top\}$ extended by $\perp$ as a bottom element.

---

1 For simplicity, we assume that all variables have the same type.
The co-flat order on \( \text{Val} \cup \{ \top \} \) is illustrated by the Hasse diagram in Fig. A.1 for \( \text{Val} = \mathbb{Z} \).

The initial value is \( d_0 = (\lambda x : \top) - \) at the start of the program we have no knowledge about the value of the variables.

The transfer functions are induced by the statements: \( [e]^\# = [A(e)]_{sc} \), where \( [\text{skip}]_{sc}(d) = d \), i.e., \( [\text{skip}] \) is the identity on \( \mathbb{D}_{sc} \), and \( [x := e]_{sc} \) is defined by \( [x := e]_{sc}(\bot) = \bot \) and \( [x := e]_{sc}(d) = d[x \mapsto e^d] \) for abstract states \( d \). The standard way of defining \( e^d \), the value of expression \( d \) in abstract state \( d \), is by extending the standard interpretation of operators from \( \text{Val} \) to \( \text{Val} \cup \{ \top \} \) in a strict way, i.e., such that each operation yields \( \top \) if any of its arguments is \( \top \).

The entities that specify a forward dataflow analysis induce a non-standard interpretation of the signature underlying the constraint system for reaching runs: the interpretation works on the lattice \( (\mathbb{D}^\#, \sqsubseteq^\#) \) of dataflow facts; constant \( c_e \) is interpreted by \( I^\#(c_e) = d_0 \), and the operator \( o_e \) by the transfer function associated with edge \( e: I^\#(o_e) = [e]^\# \). The smallest solution of the constraint system for reaching runs over this non-standard interpretation can effectivley be computed by fixpoint iteration. It is called the MFP-solution in dataflow analysis parlance. We denote the value computed for variable \( R[v] \) by \( \text{MFP}[v] \) for each \( v \in \mathbb{N} \).

Example A.4 (Simple Constant Propagation) If, for the simple constant propagation framework, \( \text{MFP}[v] \neq \bot \) and \( \text{MFP}[v](x) = c \in \text{Val} \) then \( x \) is called a simple constant of value \( c \) at program point \( v \).

The theory of abstract interpretation allows us to argue that the non-standard interpretation gives us the desired analysis result. For this purpose, we define first an abstraction function \( \alpha : \mathbb{D} \rightarrow \mathbb{D}^\# \) that describes the intended relationship between concrete interpretation \( I \) and abstract interpretation \( I^\# \). In the standard setting this amounts to a relationship between run sets and analysis results.

We call \( \alpha \) a weak homomorphism of the two interpretations \( I \) and \( I^\# \) if

1. \( \alpha(I(c)) \sqsubseteq^\# I^\#(c) \) for any constant \( c \in C \) and

2. \( \alpha(I(o))(d_1, \ldots, d_k) \sqsubseteq^\# I^\#(o)(\alpha(d_1), \ldots, \alpha(d_k)) \) for any \( k \)-ary operator \( o \in O \) and values \( d_1, \ldots, d_k \in \mathbb{D} \).

Alternatively, we say in this case that the abstract operators and constants are correct abstractions of the concrete ones. Intuitively, \( \alpha \) is a weak homomorphism if computation on abstractions yields sound but in general less accurate abstractions computation on concrete values.

---

\(^3\)For some operators, we could use a non-strict interpretation, if other arguments determine the value of the operation uniquely. For example, we could define that \( 0 \cdot \top = 0 \).
We call $\alpha$ a strong homomorphism if 1. and 2. hold with $\subseteq$ in place of $\sqsubseteq$. Alternatively, we say that the abstract operators and constants are precise abstractions of the concrete ones. Intuitively, $\alpha$ is a strong homomorphism if we get the same information by computing on abstractions.

A function $f : L \rightarrow L'$ between complete lattices $L$ and $L'$ is called distributive (universally disjunctive) if it distributes over arbitrary joins, i.e., if $f(\bigvee S) = \bigvee \{ f(l) \mid l \in S \}$ for all $S \subseteq L$.

For any variable $X$ used in a given constraint system, let $X_c \in D$ be the value assigned to variable $X$ in the smallest solution over concrete interpretation $I$ and $X_a \in D^\#$ be the value assigned to $X$ in the smallest solution over abstract interpretation $I^\#$. Then the crucial theorem can be formulated as follows:

**Theorem A.5** Suppose $\alpha$ is distributive.

1. If $\alpha$ is a weak homomorphism then $\alpha(X_c) \sqsubseteq^# X_a$.

2. If $\alpha$ is a strong homomorphism then $\alpha(X_c) = X_a$.

In forward dataflow analysis the relationship between the standard and the abstract interpretation is given by the MOP-abstraction. MOP stands for “Meet Over all Paths”. As our interpretation of the order is dual to the traditional one we define it here as a “join over all paths”. Nevertheless, we use the term MOP that is very well-established in the literature.

In order to define the MOP-abstraction, the local interpretation $[e] : D \rightarrow D$ of flow-graph edges is extended to runs by the natural definition

$$[\langle e_1, \ldots, e_k \rangle] \overset{\text{def}}{=} [e_k] \circ \cdots \circ [e_1].$$

In particular, $[\lambda] = (\lambda d \in D^\# : d)$, the identity on $D^\#$. Obviously, the information valid after execution of a particular run $r$ is given by $[r](d_0)$. The MOP-abstraction is now defined as $\alpha_{\text{MOP}} : D \rightarrow D^\#$:

$$\alpha_{\text{MOP}}(R) \overset{\text{def}}{=} \bigsqcup \{ [r](d_0) \mid r \in R \}.$$

With this definition, we clearly have

$$\alpha_{\text{MOP}}(R[v]) = \text{MOP}[v] \overset{\text{def}}{=} \bigsqcup \{ [r](d_0) \mid r \in R[v] \}.$$

That is, the set of reaching runs to $v$ is abstracted to what is commonly called the MOP-solution in dataflow analysis, where it is used as the specification of what the analysis tries to compute or approximate. The intuition is that $\text{MOP}[v]$ is the most precise abstract information we can guarantee whenever execution reaches program point $v$: we must be prepared to see any of the runs $r \in R[v]$; the best we can say after a specific run $r$ is $[r](d_0)$; and the most precise value consistent with all these values is their join. Therefore, a sound analysis must compute for program point $v$ a fact $f$ with $\text{MOP}[v] \subseteq^# f$, preferably $f = \text{MOP}[v]$. 


Example A.6 (Simple constant propagation) For the simple constant propagation framework, \( \text{MOP}[v] \neq \perp \) for each reachable program point \( v \in N \). Let us assume that \( v \) is indeed reachable. If \( x \) is a constant of value \( c \in \text{Val} \) at program point \( v \), i.e., holds \( c \) whenever execution reaches \( v \), \( \text{MOP}[v](x) = c \). Otherwise, \( \text{MOP}[v](x) = \top \). Therefore, the MOP-solution of the simple constant propagation framework is a perfect reference point for judging soundness of constant propagation algorithms.

It is not hard to prove that the MOP-abstraction is distributive. Furthermore, if all transfer functions \( [e] \) are monotonic, a very natural assumption we have made above, \( \alpha_{\text{MOP}} \) is a weak homomorphism. By Theorem A.5 this means that the constraint-based analysis delivers sound results, a classic theorem by Kam and Ullman [34].

**Theorem A.7 (Monotonic frameworks)** If all transfer functions \( [e] \), \( e \in E \), are monotonic then \( \text{MOP}[v] \leq \# \text{MFP}[v] \) for all \( v \in N \).

Example A.8 Theorem A.7 implies, in particular, that simple constant propagation yields sound results. If \( \text{MFP}[v](x) = c \in \text{Val} \) for a program point \( v \in N \) and a variable \( x \in \text{Var} \), we can infer \( \text{MOP}[v](x) = c \) because \( \text{MOP}[v] \subseteq \text{MFP}[v] \). Therefore, \( x \) is indeed a constant of value \( c \) in this case. However, if \( \text{MFP}[v](x) = \top \) we cannot infer anything.

Ideally, we would like that MOP- and MFP-solution coincide. Indeed, if we pose stronger requirements on the transfer functions we obtain such a result: it is not hard to show that \( \alpha \) is a strong homomorphism if all transfer functions \( [e] \) are universally disjunctive (distributive) and by Theorem A.5 this implies that the constraint-based analysis computes exactly the MOP-solution in this case. Thus, we obtain the classic theorem of Kildall [36] ensuring soundness and completeness of the MFP-solution for distributive frameworks.

**Theorem A.9 (Distributive frameworks)** If all transfer functions \( [e] \), \( e \in E \), are distributive then \( \text{MOP}[v] = \text{MFP}[v] \) for all \( v \in N \).

The transfer functions in simple constant propagation are not distributive as illustrated by the program in Fig. A.2: while the MOP-solution assigns the value \( 5 \) to \( z \) at node 7, the MFP-solution loosen precision at node 6 by assigning \( \top \) to both \( x \) and \( y \) at node 6. Hence, the MFP-solution assigns the sound but imprecise value \( \top \) to \( z \) at node 7. The reason is that \( \lfloor z := x + y \rfloor_{\text{sc}} \), the transfer function assigned to edge \((6,7)\), is non-distributive. Let us write \( [a, b, c] \) with \( a, b, c \in \text{Val} \cup \{\top\} \) for the abstract state that assigns \( a \) to \( x \), \( b \) to \( y \), and \( c \) to \( z \). Then

\[
[z := x + y]_{\text{sc}}([2, 3, \top] \cup [3, 2, \top]) = [z := x + y]_{\text{sc}}([\top, \top, \top]) = [\top, \top, \top]
\]
Figure A.2: Non-distributivity of simple constant propagation.

but

\[
[z := x + y]_{sc}(\{2, 3, \top\}) \cup [z := x + y]_{sc}(\{3, 2, \top\}) = [\top, \top, 5] \neq [\top, \top, \top].
\]

It is possible to define distributive frameworks for constant propagation. A well-known example is copy-constant propagation in which composite expressions are not interpreted at all.

**Example A.10 (Copy-constant propagation)** In copy-constant propagation we use the same lattice as in simple constant propagation, the same order, and the same initial value. We modify, however, the transfer functions: composite expressions are no longer interpreted. Specifically, we define for composite expressions \( e \), \([x := e]_{sc}\) by \([x := e]_{cc}() = \bot\) and \([x := e]_{cc}(d) = d[x \mapsto \top]\). For all other base statements \( s \), \([s]_{cc} = [s]_{sc}\). Note that besides of \textbf{skip}, only constant and copying assignments \( x := v \), where \( v \) is a constant or variable, are interpreted, hence the name copy-constant propagation.

It is not hard to prove that the transfer functions of the copy-constant framework are universally disjunctive. Therefore, the MFP-solution of the copy constant propagation framework coincides with the MOP-solution. Of course we pay a price for this coincidence. The MOP-solution of the copy constant propagation framework no longer captures constancy at run-time precisely; unlike the MOP-solution of simple constant propagation framework, it is itself a conservative approximation only.

There is no deep fundamental difference between the classic approach to dataflow analysis, which relies on equations, and the constraint-based approach that relies on inequations. However, the constraint-based approach enables a more modular specification, as in any single inequation we can concentrate on
one particular phenomenon, why a certain dataflow information must be weakened, while in an equational specification we must consider all of them at the same time. This often results in a more transparent specification, in particular if we consider more complex scenarios than intraprocedural analysis of sequential flow graphs like analysis of parallel flow graphs.