Foreword

By its nature, automatic program analysis is the art of finding adequate compromises. Originally, in the 1970s, program analysis aimed at deriving preconditions for typically obviously correct optimizing program transformations. Heuristics for loop optimizations were popular, which in particular concerned the treatment of multi-dimensional arrays. The limits of these heuristics-based approaches became apparent when looking at the combined effects of optimizations – in particular in the context of concurrency. Since then, the loss of confidence in optimizing compilers has been fought by semantics-based methods that come with explicitly stated power and limitations.

A particularly natural and illustrative class of program analyses aims at detecting program constants, i.e. occurrences of program expressions which are guaranteed to evaluate to the same value in every run. This problem is essentially as hard as program verification in its full generality, though there are interesting subclasses which can be solved effectively or even efficiently.

Markus Müller-Olm investigates particularly interesting variations of such classes which are characterized by varying strengths of interpretation and by increasingly complex data and control structures. In particular, he considers in detail three main classes of problems:

- The purely sequential situation, where his ideal theoretic treatment of polynomial constants is really outstanding. It is a delight to follow the elegant algebraic development!
- The treatment of copy constants for fork-join parallel programs. This turns out to be very hard already in restricted settings like acyclic programs, and becomes undecidable in the context of procedures.
- A variation of the second class, where he waives the usual atomicity properties during execution. At first sight it is really surprising that this drastically simplifies the analysis problem. However, a closer look reveals that the decrease in algorithmic complexity goes hand in hand with a decrease in quality – as the waived atomicity is vital for a decent control of parallel computation.

Markus Müller-Olm succeeds in significantly improving the known results for the scenarios considered. However, what makes the book very special is the impressive firework of elaborate methods and powerful techniques.

Everybody working in the field will profit from passing from scenario to scenario and experiencing Markus Müller-Olm's mastership of choosing the adequate means for each of the considered analysis problems: one leaves with a deep understanding of the inherent underlying differences and in particular of the complexity of modern programming concepts in terms of the hardness of the implied analysis problem.

July 2006 Bernhard Steffen

Preface

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 [79, 31], every non-trivial semantic question about programs in a universal programming language is undecidable. 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.

An interesting problem is to assess the *precision* of an approximate analysis. One approach is to consider an abstraction of programs or program behavior that gives rise to weaker but sound information and to prove that the analysis yields exact results with respect to this abstraction (cf. Fig. 0.1). 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 [40, 86, 43, 87, 88, 24].

The scheme of Fig. 0.1 allows us to make meaningful statements on approximate analysis problems independently of specific algorithms: by devising abstractions of programs, we obtain well-defined weakened analysis problems

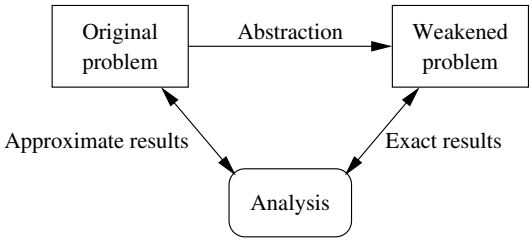

Fig. 0.1. Using an abstraction to assess the precision of an approximate analysis.

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 gain 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 monograph 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 and program slicing.

In this monograph we combine techniques from different areas such as 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 monograph:

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 restricts the set of interpreted integer expressions. The second di-

mension distinguishes between *must-* and *may-constants*. May-constants appear in two variations: single- and multiple-valued. May-constants are closely related to reachability. In the third dimension we distinguish between programs with and without loops. We succeed in classifying the complexity of the problems almost completely (Chapter 2). Moreover, we develop (must-)constant-propagation algorithms that interpret completely all integer operators except for the division operators by using results from linear algebra and computational ring theory (Chapter 3).

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 copy-constant propagation is undecidable, PSPACE-complete, and NP-complete if we consider programs with procedures, without procedures, and without loops, respectively (Chapter 4). 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 5 to 9). Surprisingly, it turns out that this makes copy-constant detection, faint-code elimination and, more generally, analysis of transitive dependences decidable for programs with procedures (Chapter 8) although it remains intractable (NP-hard) (Chapter 9). In order to show decidability we develop a precise abstract interpretation of sets of runs (program executions) (Chapter 7). 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 essentially self-contained parts that can be read independently of each other. Figure 0.2 shows the assignment of the chapters to these parts and indicates dependences between the chapters. For clarity, transitive relationships are omitted.

Throughout this monograph we assume that the reader is familiar with the basic techniques and results from the theory of computational complexity [72, 36], program analysis [70, 2, 30, 56], and abstract interpretation [14, 15]. A brief introduction to constraint-based program analysis is provided in Appendix A.

Acknowledgments

This monograph is a revised version of my habilitation thesis (*Habilitationsschrift*), which was submitted to the Faculty of Computer Science (*Fach-*

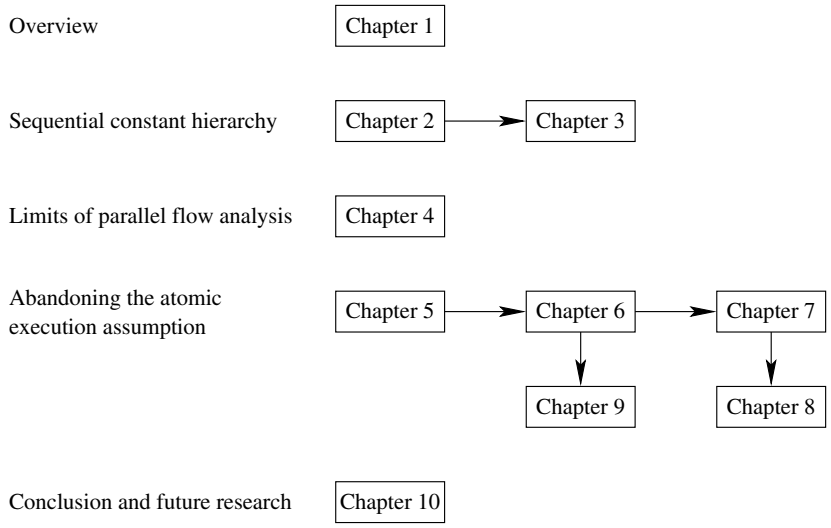

Fig. 0.2. Dependence between the chapters.

bereich Informatik) of Dortmund University in August 2002 and accepted in February 2003. I would like to thank Bernhard Steffen, head of the research group on Programming Systems and Compiler Construction at Dortmund University, 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 and Jens Knoop for many helpful discussions and Hardi Hungar for insightful comments on a draft version. I thank the referees of my habilitation thesis, Javier Esparza, Neil Jones, and Bernhard Steffen, for their time and enthusiasm.

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

Dortmund, June 2005 Markus Müller-Olm