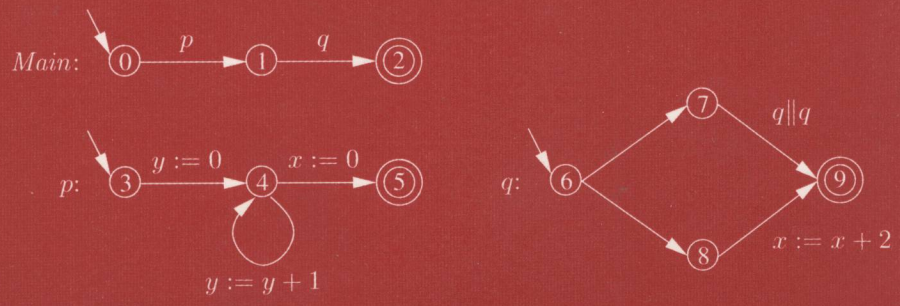
Markus Müller-Olm

LNCS 3800

Variations on Constants

Flow Analysis of Sequential and Parallel Programs



Springer

Markus Müller-Olm

Variations on Constants

Flow Analysis of Sequential and Parallel Programs

Springer

Author

Markus Müller-Olm
Westfälische Wilhelms-Universität Münster
Institut für Informatik, FB 10
Einsteinstraße 62, 48149 Münster, Germany
E-mail: mmo@denethor.uni-muenster.de

Library of Congress Control Number: 2006933227

CR Subject Classification (1998): D.2.4, D.2, D.3, F.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-45385-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-45385-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 11871743 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 3800

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Lecture Notes in Computer Science

For information about Vols. 1–4142

please contact your bookseller or Springer

Vol. 4248: S. Staab, V. Svátek (Eds.), Engineering Knowledge in the Age of the Semantic Web. XIV, 400 pages. 2006. (Sublibrary LNAI).

Vol. 4241: R. Beichel, M. Sonka (Eds.), Computer Vision Approaches to Medical Image Analysis. XI, 262 pages. 2006.

Vol. 4239: H.Y. Youn, M. Kim, H. Morikawa (Eds.), Ubiquitous Computing systems. XVI, 548 pages. 2006.

Vol. 4238: Y.-T. Kim, M. Takano (Eds.), Management of Convergence Networks and Services. XVIII, 604 pages. 2006.

Vol. 4229: E. Najm, J.F. Pradat-Peyre, V.V. Donzeau-Gouge (Eds.), Formal Techniques for Networked and Distributed Systems - FORTE 2006. X, 486 pages. 2006.

Vol. 4228: D.E. Lightfoot, C.A. Szyperski (Eds.), Modular Programming Languages. X, 415 pages. 2006.

Vol. 4227: W. Nejdl, K. Tochtermann (Eds.), Innovative Approaches for Learning and Knowledge Sharing. XVII, 721 pages. 2006.

Vol. 4224: E. Corchado, H. Yin, V. Botti, C. Fyfe (Eds.), Intelligent Data Engineering and Automated Learning – IDEAL 2006. XXVII, 1447 pages. 2006.

Vol. 4223: L. Wang, L. Jiao, G. Shi, X. Li, J. Liu (Eds.), Fuzzy Systems and Knowledge Discovery. XXVIII, 1335 pages. 2006. (Sublibrary LNAI).

Vol. 4222: L. Jiao, L. Wang, X. Gao, J. Liu, F. Wu (Eds.), Advances in Natural Computation, Part II. XLII, 998 pages. 2006.

Vol. 4221: L. Jiao, L. Wang, X. Gao, J. Liu, F. Wu (Eds.), Advances in Natural Computation, Part I. XLI, 992 pages. 2006.

Vol. 4219: D. Zamboni, C. Kruegel (Eds.), Recent Advances in Intrusion Detection. XII, 331 pages. 2006.

Vol. 4217: P. Cuenca, L. Orozco-Barbosa (Eds.), Personal Wireless Communications. XV, 532 pages. 2006.

Vol. 4216: M.R. Berthold, R. Glen, I. Fischer (Eds.), Computational Life Sciences. XIII, 269 pages. 2006. (Sublibrary LNBI).

Vol. 4213: J. Fürnkranz, T. Scheffer, M. Spiliopoulou (Eds.), Knowledge Discovery in Databases: PKDD 2006. XXII, 660 pages. 2006. (Sublibrary LNAI).

Vol. 4212: J. Fürnkranz, T. Scheffer, M. Spiliopoulou (Eds.), Machine Learning: ECML 2006. XXIII, 851 pages. 2006. (Sublibrary LNAI).

Vol. 4211: P. Vogt, Y. Sugita, E. Tuci, C. Nehaniv (Eds.), Symbol Grounding and Beyond. VIII, 237 pages. 2006. (Sublibrary LNAI).

Vol. 4209: F. Crestani, P. Ferragina, M. Sanderson (Eds.), String Processing and Information Retrieval. XIV, 367 pages. 2006.

Vol. 4208: M. Gerndt, D. Kranzlmüller (Eds.), High Performance Computing and Communications. XXII, 938 pages. 2006.

Vol. 4207: Z. Ésik (Ed.), Computer Science Logic. XII, 627 pages. 2006.

Vol. 4206: P. Dourish, A. Friday (Eds.), UbiComp 2006: Ubiquitous Computing. XIX, 526 pages. 2006.

Vol. 4205: G. Bourque, N. El-Mabrouk (Eds.), Comparative Genomics. X, 231 pages. 2006. (Sublibrary LNBI).

Vol. 4203: F. Esposito, Z.W. Ras, D. Malerba, G. Semeraro (Eds.), Foundations of Intelligent Systems. XVIII, 767 pages. 2006. (Sublibrary LNAI).

Vol. 4202: E. Asarin, P. Bouyer (Eds.), Formal Modeling and Analysis of Timed Systems. XI, 369 pages. 2006.

Vol. 4201: Y. Sakakibara, S. Kobayashi, K. Sato, T. Nishino, E. Tomita (Eds.), Grammatical Inference: Algorithms and Applications. XII, 359 pages. 2006. (Sublibrary LNAI).

Vol. 4199: O. Nierstrasz, J. Whittle, D. Harel, G. Reggio (Eds.), Model Driven Engineering Languages and Systems. XVI, 798 pages. 2006.

Vol. 4197: M. Raubal, H.J. Miller, A.U. Frank, M.F. Goodchild (Eds.), Geographic, Information Science. XIII, 419 pages. 2006.

Vol. 4196: K. Fischer, I.J. Timm, E. André, N. Zhong (Eds.), Multiagent System Technologies. X, 185 pages. 2006. (Sublibrary LNAI).

Vol. 4195: D. Gaiti, G. Pujolle, E. Al-Shaer, K. Calvert, S. Dobson, G. Leduc, O. Martikainen (Eds.), Autonomic Networking. IX, 316 pages. 2006.

Vol. 4194: V.G. Ganzha, E.W. Mayr, E.V. Vorozhtsov (Eds.), Computer Algebra in Scientific Computing. XI, 313 pages. 2006.

Vol. 4193: T.P. Runarsson, H.-G. Beyer, E. Burke, J.J. Merelo-Guervós, L. D. Whitley, X. Yao (Eds.), Parallel Problem Solving from Nature - PPSN IX. XIX, 1061 pages. 2006.

Vol. 4192: B. Mohr, J.L. Träff, J. Worringen, J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface. XVI, 414 pages. 2006.

Vol. 4191: R. Larsen, M. Nielsen, J. Sporring (Eds.), Medical Image Computing and Computer-Assisted Intervention – MICCAI 2006, Part II. XXXVIII, 981 pages. 2006.

Vol. 4190: R. Larsen, M. Nielsen, J. Sporring (Eds.), Medical Image Computing and Computer-Assisted Intervention – MICCAI 2006, Part I. XXXVVIII, 949 pages. 2006.

Vol. 4189: D. Gollmann, J. Meier, A. Sabelfeld (Eds.), Computer Security – ESORICS 2006. XI, 548 pages. 2006.

Vol. 4188: P. Sojka, I. Kopeček, K. Pala (Eds.), Text, Speech and Dialogue. XIV, 721 pages. 2006. (Sublibrary LNAI).

Vol. 4187: J.J. Alferes, J. Bailey, W. May, U. Schwertel (Eds.), Principles and Practice of Semantic Web Reasoning. XI, 277 pages. 2006.

Vol. 4186: C. Jesshope, C. Egan (Eds.), Advances in Computer Systems Architecture. XIV, 605 pages. 2006.

Vol. 4185: R. Mizoguchi, Z. Shi, F. Giunchiglia (Eds.), The Semantic Web – ASWC 2006. XX, 778 pages. 2006.

Vol. 4184: M. Bravetti, M. Núñez, G. Zavattaro (Eds.), Web Services and Formal Methods. X, 289 pages. 2006.

Vol. 4183: J. Euzenat, J. Domingue (Eds.), Artificial Intelligence: Methodology, Systems, and Applications. XIII, 291 pages. 2006. (Sublibrary LNAI).

Vol. 4182: H.T. Ng, M.-K. Leong, M.-Y. Kan, D. Ji (Eds.), Information Retrieval Technology. XVI, 684 pages. 2006.

Vol. 4180: M. Kohlhase, OMDoc – An Open Markup Format for Mathematical Documents [version 1.2]. XIX, 428 pages. 2006. (Sublibrary LNAI).

Vol. 4179: J. Blanc-Talon, W. Philips, D. Popescu, P. Scheunders (Eds.), Advanced Concepts for Intelligent Vision Systems. XXIV, 1224 pages. 2006.

Vol. 4178: A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg (Eds.), Graph Transformations. XII, 473 pages. 2006.

Vol. 4177: R. Marín, E. Onaindía, A. Bugarín, J. Santos (Eds.), Current Topics in Aritficial Intelligence. XIII, 621 pages. 2006. (Sublibrary LNAI).

Vol. 4176: S.K. Katsikas, J. Lopez, M. Backes, S. Gritzalis, B. Preneel (Eds.), Information Security. XIV, 548 pages. 2006.

Vol. 4175: P. Bücher, B.M.E. Moret (Eds.), Algorithms in Bioinformatics. XII, 402 pages. 2006. (Sublibrary LNBI).

Vol. 4174: K. Franke, K.-R. Müller, B. Nickolay, R. Schäfer (Eds.), Pattern Recognition. XX, 773 pages. 2006.

Vol. 4173: S. El Yacoubi, B. Chopard, S. Bandini (Eds.), Cellular Automata. XV, 734 pages. 2006.

Vol. 4172: J. Gonzalo, C. Thanos, M. F. Verdejo, R.C. Carrasco (Eds.), Research and Advanced Technology for Digital Libraries. XVII, 569 pages. 2006.

Vol. 4169: H.L. Bodlaender, M.A. Langston (Eds.), Parameterized and Exact Computation. XI, 279 pages. 2006.

Vol. 4168: Y. Azar, T. Erlebach (Eds.), Algorithms – ESA 2006. XVIII, 843 pages. 2006.

Vol. 4167: S. Dolev (Ed.), Distributed Computing. XV, 576 pages. 2006.

Vol. 4166: J. Górski (Ed.), Computer Safety, Reliability, and Security. XIV, 440 pages. 2006.

Vol. 4165: W. Jonker, M. Petković (Eds.), Secure, Data Management. X, 185 pages. 2006.

Vol. 4163: H. Bersini, J. Carneiro (Eds.), Artificial Immune Systems. XII, 460 pages. 2006.

Vol. 4162: R. Královič, P. Urzyczyn (Eds.), Mathematical Foundations of Computer Science 2006. XV, 814 pages. 2006.

Vol. 4161: R. Harper, M. Rauterberg, M. Combetto (Eds.), Entertainment Computing - ICEC 2006. XXVII, 417 pages. 2006.

Vol. 4160: M. Fisher, W.v.d. Hoek, B. Konev, A. Lisitsa (Eds.), Logics in Artificial Intelligence. XII, 516 pages. 2006. (Sublibrary LNAI).

Vol. 4159: J. Ma, H. Jin, L.T. Yang, J.J.-P. Tsai (Eds.), Ubiquitous Intelligence and Computing. XXII, 1190 pages. 2006.

Vol. 4158: L.T. Yang, H. Jin, J. Ma, T. Ungerer (Eds.), Autonomic and Trusted Computing. XIV, 613 pages. 2006.

Vol. 4156: S. Amer-Yahia, Z. Bellahsène, E. Hunt, R. Unland, J.X. Yu (Eds.), Database and XML Technologies. IX, 123 pages. 2006.

Vol. 4155: O. Stock, M. Schaerf (Eds.), Reasoning, Action and Interaction in AI Theories and Systems. XVIII, 343 pages. 2006. (Sublibrary LNAI).

Vol. 4154: Y.A. Dimitriadis, I. Zigurs, E. Gómez-Sánchez (Eds.), Groupware: Design, Implementation, and Use. XIV, 438 pages. 2006.

Vol. 4153: N. Zheng, X. Jiang, X. Lan (Eds.), Advances in Machine Vision, Image Processing, and Pattern Analysis. XIII, 506 pages. 2006.

Vol. 4152: Y. Manolopoulos, J. Pokorný, T. Sellis (Eds.), Advances in Databases and Information Systems. XV, 448 pages. 2006.

Vol. 4151: A. Iglesias, N. Takayama (Eds.), Mathematical Software - ICMS 2006. XVII, 452 pages. 2006.

Vol. 4150: M. Dorigo, L.M. Gambardella, M. Birattari, A. Martinoli, R. Poli, T. Stützle (Eds.), Ant Colony Optimization and Swarm Intelligence. XVI, 526 pages. 2006.

Vol. 4149: M. Klusch, M. Rovatsos, T.R. Payne (Eds.), Cooperative Information Agents X. XII, 477 pages. 2006. (Sublibrary LNAI).

Vol. 4148: J. Vounckx, N. Azemard, P. Maurine (Eds.), Integrated Circuit and System Design. XVI, 677 pages. 2006.

Vol. 4147: M. Broy, I.H. Krüger, M. Meisinger (Eds.), Automotive Software – Connected Services in Mobile Networks. XIV, 155 pages. 2006.

Vol. 4146: J.C. Rajapakse, L. Wong, R. Acharya (Eds.), Pattern Recognition in Bioinformatics. XIV, 186 pages. 2006. (Sublibrary LNBI).

Vol. 4144: T. Ball, R.B. Jones (Eds.), Computer Aided Verification. XV, 564 pages. 2006.

Vol. 4143: R. Lämmel, J. Saraiva, J. Visser (Eds.), Generative and Transformational Techniques in Software Engineering. X, 471 pages. 2006.

¥359.00元

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

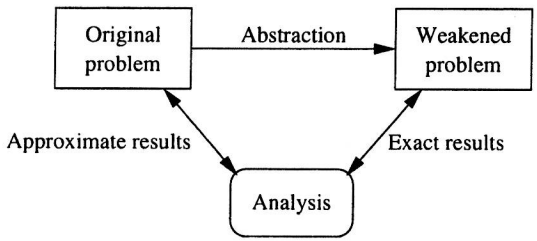


Fig. 0.1. Using an abstraction to assess the precision of an approximate analysis.

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-*

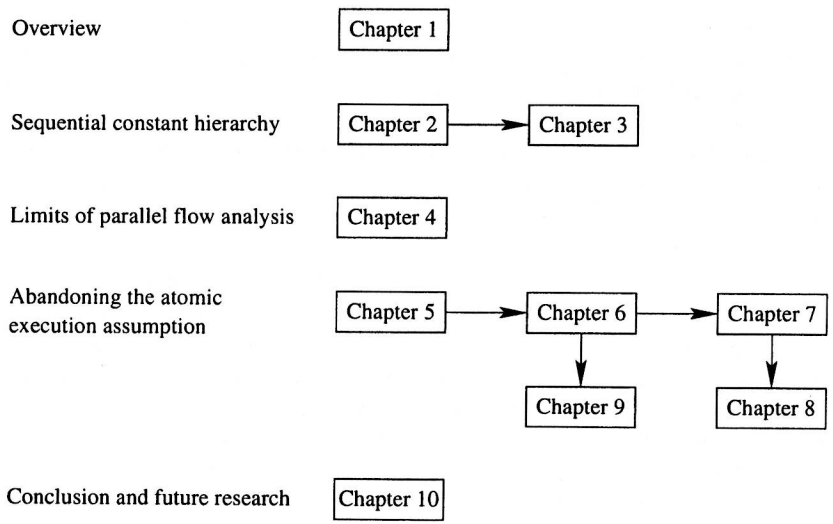


Fig. 0.2. Dependence between the chapters.

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

Table of Contents

1. Introduction 1

2. A Hierarchy of Constants 13
- 2.1 A Taxonomy of Constants 16
  - 2.1.1 Flow Graphs 16
  - 2.1.2 May- and Must-Constants 17
  - 2.1.3 Weakened Constant Detection Problems 19
  - 2.1.4 Classes of Integer Constants 21
- 2.2 Known Results 22
- 2.3 New Undecidability Results 24
- 2.4 New Intractability Results 25
- 2.5 Summary 29

3. Deciding Constants by Effective Weakest Preconditions ... 31
- 3.1 Presburger and Polynomial Constants 32
- 3.2 Presburger-Constant Detection at a Glance 33
- 3.3 A Generic Algorithm 37
- 3.4 Detection of Presburger Constants 40
- 3.5 A Primer on Computable Ideal Theory 43
- 3.6 More About $\mathbb{Z}[x_1, \ldots, x_n]$ 45
  - 3.6.1 $\mathbb{Z}[x_1, \ldots, x_n]$ as a Complete Lattice 45
  - 3.6.2 Zeros 45
  - 3.6.3 Substitution 46
  - 3.6.4 Projection 46
- 3.7 Detection of Polynomial Constants 47
- 3.8 Conclusion 49

4. Limits of Parallel Flow Analysis 53
- 4.1 A Motivating Example 55
- 4.2 Parallel Programs 56
- 4.3 Interprocedural Copy-Constant Detection 57
  - 4.3.1 Two-Counter Machines 58
  - 4.3.2 Constructing a Program 59
  - 4.3.3 Correctness of the Reduction 62

4.4 Intraprocedural Copy-Constant Detection 62
4.5 Copy-Constant Detection in Loop-Free Programs 66
4.6 Beyond Fork/Join Parallelism 67
4.7 Owicki/Gries-Style Program Proofs 67
4.8 Correctness of the Reduction in Section 4.3 68
4.8.1 Enriching the Program 68
4.8.2 The Proof Outlines 69
4.8.3 Interference Freedom 72
4.9 Correctness of the Reduction in Section 4.4 73
4.9.1 Enriching the Program 73
4.9.2 An Auxiliary Predicate 73
4.9.3 Proof Outline for π_0 74
4.9.4 Proof Outline for $\pi_i(r)$ 75
4.9.5 Proof Outline for *Main* 76
4.9.6 Interference Freedom 77
4.10 Conclusion 78

5. Parallel Flow Graphs 81
5.1 Parallel Flow Graphs 82
5.2 Operational Semantics 84
5.3 Atomic Runs 86
5.4 The Run Sets of Ultimate Interest 87
5.5 The Constraint Systems 88
5.5.1 Same-Level Runs 88
5.5.2 Inverse Same-Level Runs 90
5.5.3 Two Assumptions and a Simple Analysis 91
5.5.4 Reaching Runs 92
5.5.5 Terminating Runs 94
5.5.6 Bridging Runs 94
5.5.7 The General Case 96
5.6 Discussion 98

6. Non-atomic Execution 101
6.1 Modeling Non-atomic Execution by Virtual Variables 103
6.2 A Motivating Example 105
6.3 The Domain of Non-atomic Run Sets 106
6.3.1 Base Statements 107
6.3.2 Sequential Composition 108
6.3.3 Interleaving Operator 108
6.3.4 Pre-operator 109
6.3.5 Post-operator 109
6.4 Conclusion 109

7. Dependence Traces 111
7.1 Transparency and Dependences 113
7.2 Dependence Traces 114
7.3 Implication Order 116
7.4 Subsumption Order 117
7.5 A Lattice of Antichains 118
7.6 Short Dependence Traces 121
7.7 The Abstract Domain 124
7.8 Pre-operator 126
7.9 Post-operator 128
7.10 Sequential Composition 128
7.11 Interleaving 130
7.11.1 Complementary Dependence Traces 131
7.11.2 Interleaving Operator 132
7.11.3 Soundness Lemmas 132
7.11.4 Completeness Lemmas 136
7.11.5 Proof of Theorem 7.11.1 139
7.12 Base Edges 140
7.13 Running Time 141
7.14 Discussion 142

8. Detecting Copy Constants and Eliminating Faint Code ... 145
8.1 Copy-Constant Detection 146
8.2 Faint-Code Elimination 148
8.3 Running Time 150
8.4 Conclusion 152

9. Complexity in the Non-atomic Scenario 153
9.1 The SAT-reduction 154
9.2 Towards Stronger Lower Bounds 156
9.2.1 Assignment Statements That Propagate Twice 157
9.2.2 Propagating Runs of Exponential Length 159
9.3 Summary 160

10. Conclusion 161
10.1 Future Research 163

A. A Primer on Constraint-Based Program Analysis 165

References 173

1. Introduction

Constant propagation is one of the most widely used optimizations in practice (cf. [2, 30, 56]). 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:

$$\textbf{read}(\texttt{new})\,;P\,;\textbf{write}(\texttt{new})\,.$$

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,

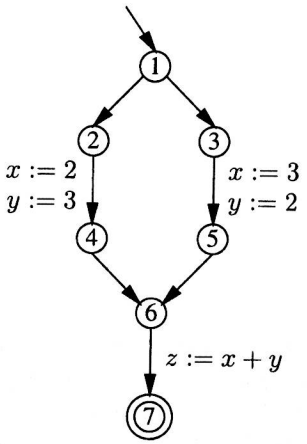


Fig. 1.1. A constant not detected by standard constant propagation.

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. 1.1; 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 some 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 monograph) 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 monograph. Non-deterministic programs represent deterministic programs in which guarded branching has been abstracted to non-deterministic branching.