Compiler Design

Helmut Seidl · Reinhard Wilhelm
Sebastian Hack

Compiler Design

Analysis and Transformation

Springer

Helmut Seidl
Fakultät für Informatik
Technische Universität München
Garching, Germany

Sebastian Hack
Programming Group
Universität des Saarlandes
Saarbrücken, Germany

Reinhard Wilhelm
Compiler Research Group
Universität des Saarlandes
Saarbrücken, Germany

ISBN 978-3-642-17547-3 ISBN 978-3-642-17548-0 (eBook)
DOI 10.1007/978-3-642-17548-0
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2012940955

ACM Codes: D.1, D.3, D.2

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Compilers for programming languages should translate source-language programs correctly into target-language programs, often programs of a machine language. But not only that; they should often generate target-machine *code* that is as efficient as possible. This book deals with this problem, namely the methods to improve the efficiency of target programs by a compiler.

The history of this particular subarea of compilation dates back to the early days of computer science. In the 1950s, a team at IBM led by John Backus implemented a first compiler for the programming language FORTRAN. The target machine was the IBM 704, which was, according to today's standards, an incredibly small and incredibly slow machine. This motivated the team to think about a translation that would efficiently exploit the very modest machine resources. This was the birth of "optimizing compilers".

FORTRAN is an imperative programming language designed for numerical computations. It offers arrays as data structures to store mathematical objects such as vectors and matrices, and it offers loops to formulate iterative algorithms on these objects. Arrays in FORTRAN, as well as in ALGOL 60, are very close to the mathematical objects that are to be stored in them.

The descriptional comfort enjoyed by the numerical analyst was at odds with the requirement of run-time efficiency of generated target programs. Several sources for this clash were recognized, and methods to deal with them were discovered. Elements of a multidimensional array are selected through sequences of integer-valued expressions, which may lead to complex and expensive computations. Some numerical computations use the same or similar index expressions at different places in the program. Translating them naively may lead to repeatedly computing the same values. Loops often step through arrays with a constant increment or decrement. This may allow us to improve the efficiency by computing the next address using the address used in the last step instead of computing the address anew. By now, it should be clear that arrays and loops represent many challenges if the compiler is to improve a program's efficiency compared to a straightforward translation.

Already the first FORTRAN compiler implemented several efficiency improving program transformations, called *optimizing transformations*. They should, however, be carefully applied. Otherwise, they would change the semantics of the program. Most such transformations have *applicability conditions*, which when satisfied guarantee the preservation of the semantics. These conditions, in general, depend on nonlocal properties of the program, which have to be determined by a *static analysis* of the program performed by the compiler.

This led to the development of *data-flow analysis*. This name was probably chosen to express that it determines the flow of properties of program variables through programs. The underlying theory was developed in the 1970s when the semantics of programming languages had been put on a solid mathematical basis. Two doctoral dissertations had the greatest impact on this field; they were written by Gary A. Kildall (1972) and by Patrick Cousot (1978). Kildall clarified the lattice-theoretic foundations of data-flow analysis. Cousot established the relation between the semantics of a programming language and static analyses of programs written in this language. He therefore called such a semantics-based program analysis *abstract interpretation*. This relation to the language semantics allows for a correctness proof of static analyses and even for the design of analyses that are correct by construction. Static program analysis in this book always means *sound static analysis*. This means that the results of such a static analysis can be trusted. A property of a program determined by a static analysis holds for all executions of the program.

The origins of data-flow analysis and abstract interpretation thus lie in the area of compilation. However, static analysis has emancipated itself from its origins and has become an important *verification method*. Static analyses are routinely used in industry to prove *safety properties* of programs such as the absence of run-time errors. Soundness of the analyses is mandatory here as well. If a sound static analysis determines that a certain run-time error will never occur at a program point, this holds for all executions of the program. However, it may be that a certain run-time error can never happen at a program point, but the analysis is unable to determine this fact. Such analyses thus are *sound*, but may be *incomplete*. This is in contrast with bug-chasing static analysis, which may fail to detect some errors and may warn about errors that will never occur. These analyses may be *unsound* and *incomplete*.

Static analyses are also used to prove partial correctness of programs and to check synchronization properties of concurrent programs. Finally, they are used to determine *execution-time bounds* for embedded real-time systems. Static analyses have become an indispensable tool for the development of reliable software.

This book treats the compilation phase that attempts to improve the efficiency of programs by semantics-preserving transformations. It introduces the necessary theory of static program analysis and describes in a precise way both particular static analyses and program transformations. The basis for both is a simple programming language, for which an operational semantics is presented.

The volume *Wilhelm and Seidl: Compiler Design: Virtual Machines* treats several programming paradigms. This volume, therefore, describes analyses and

transformations for imperative and functional programs. Functional languages are based on the λ-calculus and are equipped with a highly developed theory of program transformation.

Several colleagues and students contributed to the improvement of this book. We would particularly like to mention Jörg Herter and Iskren Chernev, who carefully read a draft of this translation and pointed out quite a number of problems.

We wish the reader an enjoyable and profitable reading.

München and Saarbrücken, November 2011

Helmut Seidl
Reinhard Wilhelm
Sebastian Hack

General literature

The list of monographs that give an overview of static program analysis and abstract interpretation is surprisingly short. The book by Matthew S. Hecht [Hec77], summarizing the classical knowledge about data-flow analysis is still worth reading. The anthology edited by Steven S. Muchnick and Neil D. Jones [MJ81], which was published only a few years later, contains many original and influential articles about the foundations of static program analysis and, in particular, the static analysis of recursive procedures and dynamically allocated data structures. A similar collection of articles about the static analysis of declarative programs was edited by Samson Abramsky and Chris Hankin [AH87]. A comprehensive and modern introduction is offered by Flemming Nielson, Hanne Riis Nielson and Chris Hankin [NNH99].

Several comprehensive treatments of compilation contain chapters about static analysis [AG04, CT04, ALSU07]. Steven S. Muchnick's monograph "Advanced Compiler Design and Implementation" [Muc97] contains an extensive treatment.The *Compiler Design Handbook*, edited by Y.N. Srikant and Priti Shankar [SS03], offers a chapter about shape analysis and about techniques to analyze object-oriented programs.

Ongoing attempts to prove compiler correctness [Ler09, TL09] have led to an increased interest in the correctness proofs of optimizing program transformations. Techniques for the systematic derivation of correct program transformations are described by Patrick and Radhia Cousot [CC02]. Automated correctness proofs of optimizing program transformations are described by Sorin Lerner [LMC03, LMRC05, KTL09].

Contents