Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

124

Wolfgang Polak

Compiler Specification and Verification

Springer-Verlag
Berlin Heidelberg New York

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

124

Wolfgang Polak

Compiler Specification and Verification

Springer-Verlag
Berlin Heidelberg New York 1981

Author

Wolfgang Polak
Computer Systems Laboratory, Stanford University
Stanford, CA 94305, USA

AMS Subject Classifications (1979): 68B10
CR Subject Classifications (1981): 4.12, 5.24

ISBN 3-540-10886-6 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-10886-6 Springer-Verlag New York Heidelberg Berlin

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

Lecture Notes in Computer Science

Vol. 23: Programming Methodology. 4th Informatik Symposium, IBM Germany Wildbad, September 25–27, 1974. Edited by C. E. Hackl. VI, 501 pages. 1975.

Vol. 24: Parallel Processing. Proceedings 1974. Edited by T. Feng. VI, 433 pages. 1975.

Vol. 25: Category Theory Applied to Computation and Control. Proceedings 1974. Edited by E. G. Manes. X, 245 pages. 1975.

Vol. 26: GI-4. Jahrestagung, Berlin, 9.–12. Oktober 1974. Herausgegeben im Auftrag der GI von D. Siefkes. IX, 748 Seiten. 1975.

Vol. 27: Optimization Techniques. IFIP Technical Conference. Novosibirsk, July 1–7, 1974. (Series: I.F.I.P. TC7 Optimization Conferences.) Edited by G. I. Marchuk. VIII, 507 pages. 1975.

Vol. 28: Mathematical Foundations of Computer Science. 3rd Symposium at Jadwisin near Warsaw. June 17–22, 1974. Edited by A. Blikle. VII, 484 pages. 1975.

Vol. 29: Interval Mathematics. Procedings 1975. Edited by K. Nickel. VI, 331 pages. 1975.

Vol. 30: Software Engineering. An Advanced Course. Edited by F. L. Bauer. (Formerly published 1973 as Lecture Notes in Economics and Mathematical Systems, Vol. 81) XII, 545 pages. 1975.

Vol. 31: S. H. Fuller, Analysis of Drum and Disk Storage Units. IX, 283 pages. 1975.

Vol. 32: Mathematical Foundations of Computer Science 1975. Proceedings 1975. Edited by J. Bečvář. X, 476 pages. 1975.

Vol. 33: Automata Theory and Formal Languages, Kaiserslautern, May 20–23, 1975. Edited by H. Brakhage on behalf of GI. VIII, 292 Seiten. 1975.

Vol. 34: GI – 5. Jahrestagung, Dortmund 8.–10. Oktober 1975. Herausgegeben im Auftrag der GI von J. Mühlbacher. X, 755 Seiten. 1975.

Vol. 35: W. Everling, Exercises in Computer Systems Analysis. (Formerly published 1972 as Lecture Notes in Economics and Mathematical Systems, Vol. 65) VIII, 184 pages. 1975.

Vol. 36: S. A. Greibach, Theory of Program Structures: Schemes, Semantics, Verification. XV, 364 pages. 1975.

Vol. 37: C. Böhm, λ-Calculus and Computer Science Theory. Proceedings 1975. XII, 370 pages. 1975.

Vol. 38: P. Branquart, J.-P. Cardinael, J. Lewi, J.-P. Delescaille, M. Vanbegin. An Optimized Translation Process and Its Application to ALGOL 68. IX, 334 pages. 1976.

Vol. 39: Data Base Systems. Proceedings 1975. Edited by H. Hasselmeier and W. Spruth. VI, 386 pages. 1976.

Vol. 40: Optimization Techniques. Modeling and Optimization in the Service of Man. Part 1. Proceedings 1975. Edited by J. Cea. XIV, 854 pages. 1976.

Vol. 41: Optimization Techniques. Modeling and Optimization in the Service of Man. Part 2. Proceedings 1975. Edited by J. Cea. XIII, 852 pages. 1976.

Vol. 42: James E. Donahue, Complementary Definitions of Programming Language Semantics. VII, 172 pages. 1976.

Vol. 43: E. Specker und V. Strassen, Komplexität von Entscheidungsproblemen. Ein Seminar. V, 217 Seiten. 1976.

Vol. 44: ECI Conference 1976. Proceedings 1976. Edited by K. Samelson. VIII, 322 pages. 1976.

Vol. 45: Mathematical Foundations of Computer Science 1976. Proceedings 1976. Edited by A. Mazurkiewicz. XI, 601 pages. 1976.

Vol. 46: Language Hierarchies and Interfaces. Edited by F. L. Bauer and K. Samelson. X, 428 pages. 1976.

Vol. 47: Methods of Algorithmic Language Implementation. Edited by A. Ershov and C. H. A. Koster. VIII, 351 pages. 1977.

Vol. 48: Theoretical Computer Science, Darmstadt, March 1977. Edited by H. Tzschach, H. Waldschmidt and H. K.-G. Walter on behalf of GI. VIII, 418 pages. 1977.

Vol. 49: Interactive Systems. Proceedings 1976. Edited by A. Blaser and C. Hackl. VI, 380 pages. 1976.

Vol. 50: A. C. Hartmann, A Concurrent Pascal Compiler for Minicomputers. VI, 119 pages. 1977.

Vol. 51: B. S. Garbow, Matrix Eigensystem Routines – Eispack Guide Extension. VIII, 343 pages. 1977.

Vol. 52: Automata, Languages and Programming. Fourth Colloquium, University of Turku, July 1977. Edited by A. Salomaa and M. Steinby. X, 569 pages. 1977.

Vol. 53: Mathematical Foundations of Computer Science. Proceedings 1977. Edited by J. Gruska. XII, 608 pages. 1977.

Vol. 54: Design and Implementation of Programming Languages. Proceedings 1976. Edited by J. H. Williams and D. A. Fisher. X, 496 pages. 1977.

Vol. 55: A. Gerbier, Mes premières constructions de programmes. XII, 256 pages. 1977.

Vol. 56: Fundamentals of Computation Theory. Proceedings 1977. Edited by M. Karpiński. XII, 542 pages. 1977.

Vol. 57: Portability of Numerical Software. Proceedings 1976. Edited by W. Cowell. VIII, 539 pages. 1977.

Vol. 58: M. J. O'Donnell, Computing in Systems Described by Equations. XIV, 111 pages. 1977.

Vol. 59: E. Hill, Jr., A Comparative Study of Very Large Data Bases. X, 140 pages. 1978.

Vol. 60: Operating Systems, An Advanced Course. Edited by R. Bayer, R. M. Graham, and G. Seegmüller. X, 593 pages. 1978.

Vol. 61: The Vienna Development Method: The Meta-Language. Edited by D. Bjørner and C. B. Jones. XVIII, 382 pages. 1978.

Vol. 62: Automata, Languages and Programming. Proceedings 1978. Edited by G. Ausiello and C. Böhm. VIII, 508 pages. 1978.

Vol. 63: Natural Language Communication with Computers. Edited by Leonard Bolc. VI, 292 pages. 1978.

Vol. 64: Mathematical Foundations of Computer Science. Proceedings 1978. Edited by J. Winkowski. X, 551 pages. 1978.

Vol. 65: Information Systems Methodology, Proceedings, 1978. Edited by G. Bracchi and P. C. Lockemann. XII, 696 pages. 1978.

Vol. 66: N. D. Jones and S. S. Muchnick, TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Languages. IX, 118 pages. 1978.

Vol. 67: Theoretical Computer Science, 4th GI Conference, Aachen, March 1979. Edited by K. Weihrauch. VII, 324 pages. 1979.

Vol. 68: D. Harel, First-Order Dynamic Logic. X, 133 pages. 1979.

Vol. 69: Program Construction. International Summer School. Edited by F. L. Bauer and M. Broy. VII, 651 pages. 1979.

Vol. 70: Semantics of Concurrent Computation. Proceedings 1979. Edited by G. Kahn. VI, 368 pages. 1979.

Vol. 71: Automata, Languages and Programming. Proceedings 1979. Edited by H. A. Maurer. IX, 684 pages. 1979.

Vol. 72: Symbolic and Algebraic Computation. Proceedings 1979. Edited by E. W. Ng. XV, 557 pages. 1979.

Vol. 73: Graph-Grammars and Their Application to Computer Science and Biology. Proceedings 1978. Edited by V. Claus, H. Ehrig and G. Rozenberg. VII, 477 pages. 1979.

Vol. 74: Mathematical Foundations of Computer Science. Proceedings 1979. Edited by J. Bečvář. IX, 580 pages. 1979.

Vol. 75: Mathematical Studies of Information Processing. Proceedings 1978. Edited by E. K. Blum, M. Paul and S. Takasu. VIII, 629 pages. 1979.

Vol. 76: Codes for Boundary-Value Problems in Ordinary Differential Equations. Proceedings 1978. Edited by B. Childs et al. VIII, 388 pages. 1979.

Preface

About four years ago David Luckham hinted to me the possibility of verifying a "real" compiler. At that time the idea seemed unrealistic, even absurd. After looking closer at the problem and getting more familiar with the possibilities of the Stanford verifier a verified compiler appeared not so impossible after all. In fact, I was fascinated by the prospect of creating a large, correct piece of software; so this subject became my thesis topic. I am very grateful to David Luckham for suggesting this topic and for his continued advice.

The research has drastically changed my view of verification and programming in general. Analysis and design of programs (even large ones) can be subject to rigorous mathematical treatment – the art of programming may become a science after all. Naturally, the reader will be skeptical, still, I hope to be able to convey some of my fascination.

This text is a revised version of my Ph.D. thesis. The research was done at Stanford University and was supported by the Advanced Research Projects Agency of the Department of Defense, by the National Science Foundation, and by the Studienstiftung des deutschen Volkes.

This work would have been impossible without the use of the Stanford verifier. I have to thank all members of the Stanford verification group for providing this excellent tool. Don Knuth's text processing system TEX was a most valuable asset for typesetting a manuscript that must be any typist's nightmare.

I would like to thank my thesis committee David Luckham, Zohar Manna, and Susan Owicki for their valuable time, for their careful reading, and for their helpful advice. Friedrich von Henke contributed through numerous discussions and careful perusal of initial drafts of my writing. Bob Boyer, J Moore, Bob Tennent, and Brian Wichman provided valuable comments on the original thesis which have improved the present text. Last but not least I thank my wife Gudrun for her patience and support.

Table of Contents

Chapter I. Introduction

Chapter II. Theoretical framework

Chapter III. Source and target languages

Chapter IV. The compiler proof

Chapter V. Conclusions

Appendix 1. Formal definition of *LS*

Appendix 2. Formal definition of *LT*

Appendix 3. The Scanner

Appendix 4. The Parser

Appendix 5. Static semantics

Appendix 6. Code generation

Chapter I. Introduction

"The ultimate goals (somewhat utopian) include error-free compilers, ... ".

S. Greibach

1. Overview

In this thesis we describe the design, implementation, and verification of a compiler for a Pascal–like language. While previous work on compiler verification has focused largely on proofs of abstract "code generating algorithms" we are interested in a "real" compiler translating a string of characters into machine code efficiently. We consider all components of such a compiler including scanning, parsing, type checking and code generation.

Our interest is twofold. First, we are concerned with the formal mathematical treatment of programming languages and compilers and the development of formal definitions suitable as input for automated program verification systems. Second, we are interested in the more general problem of verifying large software systems.

There are several reasons for verifying a compiler. Compilers are among the most frequently used programs. Consequently, if we invest in program proofs it is reasonable to do this in an area where we may expect the highest payoff. Verification techniques are in general applied to programs written in high level languages. If we ever want to use verified programs we have to be able to correctly translate them into executable machine languages; another reason why correct compilers are a necessity.

The implemented language, *LS*, contains all features of Pascal [JW76] that are of interest to compiler constructors. The language contains most control structures of Pascal, recursive procedures and functions, and jumps. It provides user defined data types including arrays, records, and pointers. A simple facility for input – output is included; each program operates with one input and one output file.

The target language, *LT*, assumes a stack machine including a display mechanism [RR64, Or73] to handle procedure and function calls. This language simplifies our task somewhat as it avoids the issue of register allocation and similar irrelevant

details. But at the same time the target language is realistic in that similar machine architectures exist, notably the B6700.

The compiler itself is written in Pascal Plus, a dialect of Pascal accepted by the Stanford verifier.[1] The Stanford verifier [SV79] is used to give a complete formal machine checked verification of the compiler.

We review existing methods for the formal definition of programming languages. We use the most appropriate definitional methods to formally define source and target language.

We further investigate how the correctness of a compiler can be specified in a form suitable for mechanical verification. Here, we have to deal with several technical issues such as fixed points and reasoning about pointers.

We show that an efficient program can be developed systematically from such specifications.

During this research verification has proven to be a most useful tool to support program development; verification should be an integral part of the development process and plays a role comparable to that of type ckecking in strongly typed languages. This methodology of verification supported programming is not limited to compilers, rather it is applicable to arbitrary problem domains.

The results of this thesis are encouraging and let us hope that verification will soon become a widely accepted software engineering tool. But also this work reveals many of the trouble spots still existing in today's verification technology. We point to several promising research areas that will make verification more accessible. The need for better human engineering and integrated software development systems is particularly urgent.

2. Program verification

Verification has provoked several controversial statements by opponents and proponents recently [DL79]. Therefore it is appropriate at this point to clarify what verification is, what it can do, and, most importantly, what it cannot do.

To verify a program means to prove that the program is consistent with its specifications. Subsequently, the term "verification" is used as a technical term referring to the process of proving consistency with specifications. A program is "verified" if a consistency proof has been established.

Formal specifications for a program can express different requirements. For example, a specification can be "the program terminates for each set of input data".

1.) Notable difference to standard Pascal is that formal documentation is an integral part of the language, for more details see 2.3.

Or, even more trivially, one can specify that "the program satisfies all type and declaration constraints"; every compiler verifies this property. But of course, we consider more interesting properties of our compiler. What exactly its specifications are is discussed in the following sections and in more detail in chapter IV.

Since specifications can be weak and need not (and generally do not) express all requirements for a program verification should not be confused with correctness in the intuitive sense (i.e. the program does what one expects it to do). Verification is not a substitute for other software engineering techniques such as systematic program development, testing, walk-through and so on; rather verification augments these techniques. It gives us an additional level of confidence that our program has the property expressed in its specifications.

Depending on the application of a program certain errors may be mere annoyances while other may have disasterous effects. We can classify errors as "cheap" and "expensive". For example, in terms of our compiler cheap properties are the reliablility of error recovery and termination. An expensive error would be if the compiler would translate an input program without reporting an error but would generate wrong code.

As long as verification is expensive we can concentrate our efforts on the most costly requirements of a program. These requirements can be formalized and taken as specifications for the program. Verification can be used to guarantee these expensive requirements. Other cheaper properties can be validated by conventional testing methods. Redundant code can easily be added to a verified program to increase its reliability or establish additional properties without effecting verified parts of the program. For example, in our compiler a sophisticated error recovery could be added without invalidating program proofs.

2.1. Writing correct programs

The non–technical use of the expression "to verify a program" suggests that there is an existing program which we subject to a verification. This is possible in principle but most certainly not practical for large software systems; it will only work for "toy" programs.

Instead, verification should be an integral part of program development: a program and its proof should be developed simultaneously from their specifications.

2.1.1. Program design

The process of designing a verified program can be visualized by figure 1.
The starting point for the program design is a formal specification of the problem. In general, this specification alone is insufficient to prove the correctness of an