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

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

1. Overview 1

2. Program verification 2
   2.1. Writing correct programs 3
      2.1.1. Program design 3
      2.1.2. An example 5
      2.1.3. A word of warning 6
   2.2. Logics of programming 6
      2.2.1. Logic of computable functions (LCF) 6
      2.2.2. First order logic 7
      2.2.3. Hoare's logic 7
   2.3. The Stanford verifier 7
      2.3.1. Assertion language 8
      2.3.2. Theorem Prover 9

3. Formal definition of programming languages 10
   3.1. Syntax 10
      3.1.1. Micro syntax 10
      3.1.2. Phrase structure 11
      3.1.3. Tree Transformations. 12
   3.2. Semantics 12
      3.2.1. Operational semantics 12
      3.2.2. Denotational semantics 13
      3.2.3. Floyd - Hoare logic 13
      3.2.4. Algebraic semantics 14
      3.2.5. Others 15
   3.3. Machine languages 15
   3.4. Summary 16

4. Developing a verified compiler 16
   4.1. What we prove 17
   4.2. Representation 18
   4.3. Scanner 19
   4.4. Parser 20
   4.5. Semantic analysis 21
   4.6. Code generation 22
      4.6.1. Necessary theory 22

4.6.2. Implementation 23

5. Related work 24
5.1. Previous work on compiler verification 24
5.2. Relation to our work 25
5.3. Compiler generators 26

6. Organization of this thesis 27

Chapter II. Theoretical framework

1. Basic concepts 28
1.1. Functions 28
1.2. First order logic 29
1.2.1. Syntax 29
1.2.2. Semantics 29

2. Scott's logic of computable functions 30
2.1. Basic definitions 31
2.2. Operations on domains 31
2.3. Notations 33
2.3.1. Conditionals 33
2.3.2. Lists 34
2.4. Fixed points 34
2.4.1. Fixed point induction 34
2.4.2. Reasoning about fixed points 35

3. Denotational semantics 36
3.1. General concepts 36
3.2. Semantic concepts of Algol–like languages 37
3.3. Denotational definition of a machine language 38
3.4. Notational issues 38

4. Verification techniques 39
4.1. Pointers 40
4.1.1. Reference classes 40
4.1.2. Reasoning about pointers 40
4.1.3. Reasoning about extensions 41

4.2. Quantification 42
4.3. Computable functions and first order logic 44
4.3.1. Representations 44
4.3.2. Standard interpretations 46
4.3.3. Higher order functions 46
4.3.4. Least fixed points 47
4.3.5. Operationalization of fixed points 48

Chapter III. Source and target languages

1. The source language 51
1.1. Data structures 51
1.2. Program structures 53
1.3. Structure of the formal definition 54

2. Micro syntax 55
2.1. Definitional formalism 55
2.2. Micro syntax of *LS* 57

3. Syntax 57
3.1. Labeled context free grammars 57
3.1.1. The accepted language 58
3.1.2. Parse trees 58
3.1.3. The function defined by a labeled grammar 58
3.2. Syntax of *LS* 59

4. Tree transformations 59
4.1. Abstract syntax 59
4.1.1. A different notation 60
4.1.2. Syntactic domains of *LS* 60
4.2. Tree transformations for *LS* 62

5. Semantics of *LS* 62
5.1. Semantic concepts 63
5.1.1. Semantic domains 63
5.1.2. Types and modes 67
5.1.3. Auxiliary functions, static semantics 67

5.2. Static semantics 69
5.2.1. Declarations 69
5.2.2. Types 70
5.2.3. Labels and identifiers 70
5.2.4. Expressions 70
5.2.5. Statements 71
5.2.6. Commands 71
5.3. Dynamic semantics 71
5.3.1. Auxiliary functions 72
5.3.2. Memory allocation 73
5.3.3. Declarations 73
5.3.4. Expressions 74
5.3.5. Statements 75

6. The target language *LT* 76
6.1. A hypothetical machine 77
6.1.1. Design decisions 77
6.1.2. Architecture 77
6.1.3. Instructions 79
6.2. Formal definition of *LT* 80
6.2.1. Abstract syntax 80
6.2.2. Semantic domains 80
6.2.3. Semantic equations 81

Chapter IV. The compiler proof

1. Verifying a compiler 82
1.1. The compiler 82
1.1.1. Correctness statement 82
1.1.2. Structure of the compiler 82
1.2. The individual proofs 84

2. A scanner for *LS* 85
2.1. Underlying theory 85
2.1.1. A suitable definition 85
2.1.2. Axiomatization of concepts 86
2.2. Basic algorithm 87

2.3. Implementation details 89

3. A parser for *LS* 90
3.1. LR theory 90
3.1.1. LR–parsing tables 91
3.1.2. The LR–parsing algorithm 92
3.1.3. Axiomatization 93
3.2. Tree transformations 95
3.2.1. Building abstract syntax trees 95
3.3. Refinement 96
3.3.1. Development cycle 96
3.3.2. Representation 97
3.3.3. Reference classes and pointer operations 98

4. Static semantics 99
4.1. Recursive declarations 99
4.1.1. Operationalization 100
4.1.2. Revised definition of t and dt 101
4.1.3. Representation of $U_s \rightarrow U_s$ 103
4.1.4. Resolving undefined references 104
4.2. Development of the program 105
4.2.1. Computing recursive functions 105
4.2.2. Refinement 109
4.2.3. Representation 109
4.2.4. Auxilliary functions 112
4.2.5. The complete program 113

5. Code generation 113
5.1. Principle of code generation 114
5.2. Modified semantics definitions 116
5.2.1. A structured target language 116
5.2.2. A modified definition of *LS* 118
5.3. Relation between *LS* and *LT* 120
5.3.1. Compile time environments 121
5.3.2. Storage allocation 121
5.3.3. Storage maps 122
5.3.4. Relations between domains 123
5.3.5. Existence of recursive predicates 126
5.4. Implementation of the code generation 127
5.4.1. Specifying code generating procedures 127

5.4.2. Treatment of labels 131
5.4.3. Declarations 134
5.4.4. Procedures and functions 135
5.4.5. Blocks 137
5.4.6. Refinement 137

Chapter V. Conclusions

1. Summary 138

2. Extensions 139
2.1. Optimization 139
2.2. Register machines 140
2.3. New language features 140
2.4. A stronger correctness statement 141

3. Future research 142
3.1. Structuring a compiler 142
3.2. Improvements of verification systems 143
3.3. Better verification techniques 144
3.4. Program development systems 145

References 146

Appendix 1. Formal definition of *LS*

1. Micro Syntax of *LS* 156
1.1. Domains 156
1.2. Languages L_i 156
1.3. Auxiliary definitions 157
1.4. Semantic Functions 158

2. Syntax of *LS* 160

3. Abstract syntax 162
3.1. Syntactic Domains 162
3.2. Constructor functions 163

4. Tree transformations 164
4.1. Programs 164
4.2. Declarations 165
4.3. Expressions 166
4.4. Statements 167

5. Semantics of *LS* 167
5.1. Semantic Domains 167
5.2. Types and Modes 168

6. Static Semantics of *LS* 169
6.1. Auxilliary functions, static Semantics 169
6.2. Declarations 174
6.3. Expressions 176
6.4. Statements 177

7. Dynamic Semantics of *LS* 179
7.1. Auxiliary functions 179
7.2. Declarations 185
7.3. Expressions 187
7.4. Statements 189

Appendix 2. Formal definition of *LT*

1. Abstract syntax 191

2. Semantic Domains 191

3. Auxiliary Functions 192

4. Semantic Equations 192

Appendix 3. The Scanner

1. Logical basis 195
1.1. Definition of the micro syntax 195
1.2. Representation functions 197
1.3. Sequences 198

2. The program 199

3. Typical verification conditions 207

Appendix 4. The Parser

1. Logical basis 211
1.1. Representation functions 211
1.2. LR theory 211
1.3. Tree transformations 212
1.4. Extension operations 213

2. The program 214

Appendix 5. Static semantics

1. Logical basis 225
1.1. Rules for *e* 225
1.2. Recursive types 227
1.2.1. Types 227
1.2.2. Fixed points 229

2. The program 230
2.1. Declarations 230
2.1.1. Types 230
2.1.2. Abstract syntax 231
2.1.3. Auxiliary functions 234

2.2. Expressions 237
2.3. Types 240

Appendix 6. Code generation

1. Logical basis 245

2. The program 250
2.1. Declarations 250
2.2. Virtual procedures 250
2.3. Auxiliary functions 252
2.4. Abstract syntax, types and modes 253
2.5. Code generating functions 255
2.6. Expressions 261
2.7. Commands 265
2.8. Statements 267