Lecture Notes in Computer Science 675

Edited by G. Goos and J. Hartmanis

Anne Mulkers

Live Data Structures in Logic Programs

Derivation by Means of Abstract Interpretation

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
W-7500 Karlsruhe, FRG

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Author

Anne Mulkers
Department of Computer Science, K.U. Leuven
Celestijnenlaan 200 A, B-3001 Heverlee, Belgium

CR Subject Classification (1991): F.3.1, D.3.4, I.2.2-3

ISBN 3-540-56694-5 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-56694-5 Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera ready by author/editor
45/3140-543210 - Printed on acid-free paper

Preface

Abstract interpretation is a general approach for program analysis to discover at compile time properties of the run-time behavior of programs, as a basis to perform sophisticated compiler optimizations. Several frameworks of abstract interpretation for logic programs have been presented [11, 25, 27, 43, 48, 49, 51, 55, 57, 65, 81, 82]. A framework is a parameterized construction for the static analysis of programs, together with theorems that ensure the soundness and termination of the analysis. To complete the construction, an application specific domain and primitive operations satisfying certain safety conditions must be provided.

This book elaborates on an application for such a generic framework. The framework used [11] belongs to the class of top-down abstract interpretation methods and collects the information derived in an abstract AND-OR-graph that represents the set of concrete proof trees that can possibly occur when executing the source program. The starting point of the present work is the previously developed application of integrated type and mode analysis [38]. The purpose of that application was to guide the compiler, based on a characterization of the entry uses of the program, to generate code that is more specific for the calls that can occur at run time.

In an attempt to give further guidance to the compiler, we address the problem of compile-time garbage collection, the purpose of which is to (partially) shift run-time storage reclamation overhead to compile time. In applicative programming languages, the programmer has no direct control over storage utilization, and run-time garbage collection is necessary. Garbage collection involves a periodic disruption of the program execution, during which usually a marking and compaction algorithm is employed. Such schemes are expensive in time. Our research shows that at compile time useful and detailed information about the liveness of term substructures can be deduced which the compiler can use to improve the allocation of run-time structures. In fact, it provides a technique to automatically introduce destructive assignments into logic languages in a safe and transparent way, thereby reducing the rate at which garbage cells are created. The resulting system gets near to the methods of storage allocation used in imperative programming languages.

The global flow analysis to be performed on Prolog source programs in order to derive the liveness of data structures is constructed in three layers. The first layer, consisting of the *type* and *mode* analysis, basically supplies the logical terms to which variables can be bound. The two subsequent layers of the analysis heavily rely on these descriptions of term values. The *sharing analysis* derives how the representation of logical terms as structures in memory can be shared, and the *liveness analysis* uses the sharing information to determine when a term structure in memory can be live.

Acknowledgments

This book is based on my Ph.D. dissertation [59] conducted at the Department of Computer Science of the K.U.Leuven, Belgium. The research presented has been carried out as part of the RFO/AI/02 project of the *Diensten voor de programmatie van het wetenschapsbeleid*, which started in November 1987 and was aimed at the study of implementation aspects of logic programming: 'Logic as a basis for artificial intelligence: control and efficiency of deductive inferencing and parallelism'.

I am indebted to Professor Maurice Bruynooghe, my supervisor, for giving me the opportunity to work on the project and introducing me to the domain of abstract interpretation, for sharing his experience in logic programming, his invaluable insights and guidance. I wish to thank Will Winsborough for many helpful discussions, for his advice on the design of the abstract domain and safety proofs and his generous support; Gerda Janssens for her encouragement and support, and for allowing the use of the prototype for type analysis as the starting point for implementing the liveness analysis; Professors Yves Willems and Bart Demoen, for managing the RFO/AI/02 project and providing me with optimal working facilities; Professor Marc Gobin, my second supervisor, and Professors Baudouin Le Charlier and Danny De Schreye, for their interest and helpful comments, and for serving on my Ph.D. thesis committee. I also want to thank my family, friends and colleagues for their support and companionship.

Leuven, March 1993 Anne Mulkers

Contents

1 Introduction 1

2 Abstract Interpretation 5

2.1 Basic Concepts 5

2.2 Abstract Interpretation Framework 7

2.2.1 Overview of the Framework 8

2.2.2 Concrete and Abstract Domains of Substitutions 10

2.2.3 Primitive Operations 11

2.2.4 Abstract Interpretation Procedure 14

2.3 Example: Integrated Type and Mode Inference 16

2.3.1 Rigid and Integrated Type Graphs 16

2.3.2 Type-graph Environments 23

2.3.3 Primitive Operations for Type-graph Environments 25

3 Related Work 31

3.1 Aliasing and Pointer Analysis 31

3.2 Reference Counting and Liveness Analysis 38

3.3 Code Optimization 41

4 Sharing Analysis 47

4.1 Sharing Environments 47

4.1.1 Concrete Representation of Shared Structure 48

4.1.2 Abstract Representation of Shared Structure 55

4.1.3 The Concrete and Abstract Domains 62

4.1.4 Order Relation and Upperbound Operation 66

4.2 Primitive Operations 68

4.2.1 Unification 68

4.2.1.1 $X_i = X_j$ 69

4.2.1.2 $X_i = f(X_{i_1}, \ldots, X_{i_j})$ 85

4.2.2 Procedure Entry 93

4.2.3 Procedure Exit 98

4.3 Evaluation 110

4.3.1 Example: `insert/3` 111

4.3.2 Relevance of Sharing Edges 114

4.3.3 Imprecision in the Sharing Analysis 117
4.3.4 Efficiency of the Sharing Analysis 123

5 Liveness Analysis 127
5.1 Liveness Environments 127
5.1.1 Concrete Representation of Liveness Information 128
5.1.2 Abstract Representation of Liveness Information 133
5.1.3 The Concrete and Abstract Domains 141
5.1.4 Order Relation and Upperbound Operation 145
5.2 Primitive Operations 147
5.2.1 Unification 147
5.2.1.1 $X_i = X_j$ 147
5.2.1.2 $X_i = f(X_{i_1}, \ldots, X_{i_j})$ 153
5.2.2 Procedure Entry 154
5.2.3 Procedure Exit 163
5.3 Evaluation 165
5.3.1 Example: qsort/3 165
5.3.2 Precision of the Liveness Analysis 168
5.3.3 The Practical Usefulness of Liveness Information 171

6 Conclusion 179

Appendix: Detailed Examples 183
A.1 List of Types 184
A.2 append/3 185
A.3 nrev/2 188
A.4 buildtree/2 and insert/3 193
A.5 permutation/2 and select/3 196
A.6 split/3 199
A.7 qsort/2 and partition/4 202
A.8 sameleaves/2 and profile/2 205
A.9 sift/2 and remove/3 209

Bibliography 213