Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

217

Programs as Data Objects

Proceedings of a Workshop
Copenhagen, Denmark, October 17–19, 1985

Edited by H. Ganzinger and N.D. Jones

Springer-Verlag
Berlin Heidelberg New York Tokyo

Editors

Harald Ganzinger
Fachbereich Informatik, Universität Dortmund
Postfach 500500, D-4600 Dortmund 50

Neil D. Jones
DIKU
Universitets Parken 1, DK-2100 Copenhagen Ø

The Workshop was organized by Neil D. Jones

CR Subject Classifications (1985): D.3.1, D.3.4, F.3, I.2.2

ISBN 3-540-16446-4 Springer-Verlag Berlin Heidelberg New York Tokyo
ISBN 0-387-16446-4 Springer-Verlag New York Heidelberg Berlin Tokyo

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

PREFACE

There has been a nearly explosive growth in the usage of computers over the past twenty years, especially in the last decade due to the advent of microcomputers. On the other hand, the increase in the number of qualified computer scientists has been much less. The result has been the now well-known "software crisis".

An obvious solution, but one which is still somewhat in advance of the state of the art, is to use the computer itself as a tool, to produce and maintain software on a large scale with minimal human intervention. In particular, we need to be able to treat programs *en masse* - as members of collections rather than as individuals (similar changes of viewpoint had to occur with the introduction of both agriculture and mass production).

Unfortunately, programming as now practiced is at best a well-developed handcraft. In the last decade we have learned some techniques which allow faster development of more reliable and user-friendly programs than could be done before, but program development is still far from an assembly line process. Aside from compilers, text editors and file systems, there are in fact very few computer-based tools to facilitate program construction and maintenance.

What is needed is a wide range of flexible tools which can be used to manipulate programs - tools which treat programs as *data objects,* which can be executed, analyzed, synthesized, transformed, verified and optimized. Treatment of programs as data is far from trivial, though, since programs are objects which carry *meanings.* Program meanings are rather complex, and can only be discussed in relation to the *semantics* of the programming language in which they are written.

Consequently, development of truly powerful program manipulation tools will require a deeper understanding of semantics and its relation to possible principles for systematic and automatic program construction. This new understanding is analogous to the new insights in physics and engineering that had to be developed before the first automobile factories based on the assembly line and mass production could be built.

The theme of this workshop was "Programs as Data Objects", and it will be seen that all the papers in this proceedings are concerned with the problems just mentioned. Some address the development of methods for automatic construction, transformation and analysis of programs, others introduce new programming constructs which it is hoped will aid the construction and manipulation of large-scale, high-level programs, and others formulate and solve mathematical questions which are directly relevant to treating programs as data.

Most of the papers are concerned with functional programming languages. Only a few years ago a book with the same title would have exclusively been concerned with compiler-writing systems oriented around more traditional imperative languages and compiler-writing systems (based on, for example, parser generators and attribute grammar evaluators). It has become clear in practice, though, that one needs more than tools for syntax analysis and computation and management of the values used during compilation (although these are indispensible, and their successful automation has been a very big step forward).

One needs as well a precise understanding of the semantics of the programming languages involved in order to construct program manipulation systems such as compilers, verifiers, intelligent editors and program synthesizers. It is here that functional languages have significant advantages, for they have shown themselves to be more tractable and in general easier to transform and analyze, due to their relatively simple semantics in comparison with the more traditional imperative languages.

The papers in this collection fall into several natural groups. Two papers, those by Bellegarde and Wadler, are concerned with program transformation with the aim of more efficient implementation. Bellegarde addresses the problem of transforming programs written in John Backus' FP language into equivalent ones which use less intermediate storage. She shows that this may be done by using algorithms from the theory and practice of term rewriting systems to rewrite programs. Wadler showed (in an earlier article) how to transform programs written in a lazy functional language into much more efficient *listless* ones resembling finite automata, using Turchin's "driving" algorithm. In the current paper he shows that a large class of listless programs may be constructed from basic ones by symbolic composition, and shows how to integrate execution of listless programs with the usual graph reduction implementation techniques.

The papers by Ganzinger and Giegerich address the question of how one may develop formal specifications of compilers and interpreters by means of small, relatively independent and loosely coupled modules (in contrast with the large, monolithic, unstructured language specifications all too often seen in practice). Further, they show how logic programming may be used to develop prototype implementations directly from the specifications. The papers differ in their choice of problem areas, and in the prototyping techniques they use.

Christiansen describes parsing and compilation techniques for "generative languages", a class of extensible languages with powerful abstraction mechanisms and a wide range of binding times which generalize some aspects of denotational semantics and attribute grammars.

Nordström shows that the multilevel arrays which have been used in the VDL and VDM projects may be generalized to allow *multilevel functions*, and that this rather general class may be formalized within the framework of Martin-Löf's highly constructive type theory. He then shows that a number of familiar data structures and algorithms may be naturally expressed in terms of multilevel functions.

Program flow analysis is a well-established technique for program analysis, which has its roots in methods developed for highly optimizing compilers for imperative languages. It is largely based on *abstract interpretation, i.e.* the execution of programs over nonstandard domains of data values which describe the sets of data on which the program may be run. Nielson's paper is concerned with making a rather general framework he developed earlier more usable for practical applications. He shows that "safe" data flow analysis may be done by using any of several *expected forms;* these are more natural and easily computable than the more precise "induced" flow analyses which come out of his general framework.

Burn, Hankin and Abramsky develop a mathematically elegant and very general framework for the flow analysis of the typed lambda calculus, using categories and power domains, and show that strictness analysis (discussed below) is in fact a very natural abstract interpretation over finite lattice structures. One outcome is a surprisingly simple and efficient algorithm for strictness analysis.

Mycroft and Jones show that a wide variety of flow analyses of the lambda calculus (not necessarily typed) may be naturally regarded as *interpretations* within a common semantic framework, and show how one may use relations to compare one interpretation with another. As an application, they show that soundness of the polymorphic type system of Milner, Damas and others may be established by showing that it abstracts the standard call-by-name interpretation.

In all, four papers are concerned with the problem of determining whether a given argument of a given defined function is *strict*, meaning that undefinedness of the argument will imply undefinedness of the function's value. One reason for doing strictness analysis is for efficient implementation of functional languages: a strict argument may be evaluated prior to calling the given function, thus avoiding the often significant overhead involved in sending arguments "by name" or "by need". Further, the results obtained in these papers have wider implications than may be apparent from the problem description. An important reason is that strictness analysis is the simplest natural nontrivial flow analysis of lazy higher order functional languages (a class whose popularity is growing rapidly). As such, it points the way to further, more sophisticated flow analyses which could aid materially in implementing these powerful and concise languages with high efficiency.

The paper by Maurer extends earlier results by Mycroft. It presents a method using abstract interpretation of the lambda calculus over a domain of "need expressions", and gives algorithms for constructing safe approximations to a lambda expression's strictness properties. Burn, Hankin and Abramsky develop (as mentioned above) a framework for strictness analysis of the typed lambda calculus.

Hughes extends the strictness question to structured data, and derives *evaluation contexts* from a first-order program; expressions which describe those substructures of a structured data object which actually must be available in order to evaluate a given program. This is particularly relevant to programs that manipulate conceptually infinite data structures, *e.g.* communication streams within an operating system.

Finally, Abramsky shows a very elegant result: that nearly any program property of computational interest is *polymorphically invariant.* One consequence is that many familiar program analysis algorithms which have been developed to handle the monomorphic case can be carried over to polymorphic programs. In particular, the techniques of Burns, Hankin and Abramsky can be applied, essentially without change, to the polymorphic lambda calculus.

The last group of papers can be described as steps towards semantics-directed compiler generation. Nielson and Nielson show how machine code may be generated for semantic definitions written using the two-level metalanguage they developed in previous papers. This indicates the possibility of using data flow information in automatic compiler generation from denotational semantics.

Sestoft describes the techniques used by MIX, a *partial evaluation* system which can been used for compilation. MIX has also been self-applied to achieve compiler generation and even compiler generator generation (the last was done by using MIX to partially evaluate the MIX program, when run on itself as known input!).

Turchin's work centers about the concept of *metasystem transition,* in which one program performs an analysis of the behavior of another program written in the same language. This leads to the possibility of self-application. Turchin was the first to understand that triple self-application can, in principle, yield a compiler generator, and MIX was its first realization on the computer. The paper here is concerned with *supercompilation,* a powerful program transformation technique capable of partial evaluation, compilation, data structure analysis and other operations on programs.

Schmidt outlines a strategy for converting a "direct" denotational semantics into a compiler from the language it defines into stack code. Particular attention is paid to the question of serializing the store, to attain reasonable efficiency on traditional Von Neuman architectures. Wand addresses similar problems using a continuation-based semantics, and describes a systematic way to transform a language definition in the form of an interpreter written in an applicative language into a compiler producing efficient code using the familiar stack-offset addressing of data.

The workshop "Programs as Data Objects" was held October 17-19, 1985. It was made possible by two Danish grants, one from the Natural Science Research Council and one from the the Ministry of Education's "Pool for International Contacts in Computer Science" (Statens naturvidenskabelige Forskningsråd og DVUs "pulje for internationale kontakter i datalogi"). The workshop was conceived in a conversation with Harald Ganzinger; it was planned and organized by the undersigned, and editing of the papers was done jointly with Harald Ganzinger. The University of Dortmund printed preliminary conference proceedings for the participants, and DIKU (the department of Computer Science at the University of Copenhagen) provided reproduction facilities and secretarial help. Local arrangements were handled by Gunvor Howard (thanks for an outstanding job, Gunvor!). The contributions of a number of individuals are warmly acknowledged, including: Nils Andersen, Klaus Grue, Peter Sestoft and Harald Søndergaard.

Neil D. Jones
November 1985
Copenhagen, Denmark

Contents