Dagstuhl Seminar 98231

Programs: Improvements, Complexity, and Meanings

Schloß Dagstuhl, 8–12 June 1998

Organizers:

Neil D. Jones, University of Copenhagen
Oege de Moor, Oxford University
James S. Royer, Syracuse University

Introduction

The general topic of this seminar was the interface between Programming Languages and Complexity Theory. A model question for this area is:

> Are functional or logic programs necessarily less efficient than equivalent imperative programs?

This has been the subject of endless hallway debates which have generated "much heat, but little light." Such questions need to be made more precise in order to be answered and solution can be tricky indeed. A model answer to the above question was given by Nick Pippenger (POPL 1996):

> "Pure LISP" is *provably* less efficient, by a logarithic time factor, than "impure LISP" with list-modifying assignments, when applied to solve a problem involving a series of online permutations.

This result together with a result relating the efficiency of "pointers versus addresses" (Ben-Amram and Galil) and a proof that "constant time factors do matter" (Jones) have inspired considerable interest in this topic. However, the area is still in a nascent stage.

The intent of this workshop was bring together:

Semanticists interested in operational models of programming languages, expressiveness, intensional properties, etc.

Program transformers and analysts interested in program execution speed, limits to implementation efficiency, etc.

Complexity theorists interested in precisely stated problems about computation time and space, language expressiveness, etc.

The goal was to promote fruitful interactions, leading to formulation and perhaps addressing new questions of both theoretical and practical interest.

As one can see from the following abstracts, there was a great breadth to the topics discussed and areas represented. A considerable amount of bridge building took place, but there were clearly some chasms no one yet knows how to cross (e.g., how to deal with all the issues involved in rigorous complexity analyses of nontrivial lazy programs). The discussions were lively, wide ranging, and stimulating. The organizers wish to thank all the participants, as well has the Dagstuhl staff, for making this a successful meeting.

The Interface Between Complexity and Programming Languages

Neil D. Jones
DIKU, University of Copenhagen

Comparisons are made between research in complexity theory and research in programming languages, with an eye to gaining the good features of both on subjects of common interest.

Complexity is an extensional theory, focusing on *what* is to be computed rather than how. High points include the existence of complexity classes PTIME, etc., that are *robust* with respect to machine model and choice of problem representation; and notions of problem equivalence via reductions, leading to *complete problems*. Low points: a tendency to ignore constants and to concentrate only on asymptotic factors; messy and sometimes cryptic algorithm descriptions.

Language research is intensional, focusing on algorithms more than problems. High points include elegance, generality, giving a basis for correctness proofs and program synthesis and transformation. Low points: imprecise cost measures, lack of robustness, few common assumptions.

A discussion was made of the lack of pen problems in programming languages—the tendency to set up new frameworks rather than work on already-existing problems.

The talk ended with an overview of the contributions by the speaker to the interface. Results include complexity bounds on program analysis, characterizations of LOGSPACE and PTIME by simple programming languages, a theorem that for a natural language I and a constant b, TIME(abn) $\not\subseteq$ TIME(an) for all $a \geq 1$, and an analysis of Levin's Theorem, that r.e. search problems *all* have near-optimal search programs. Both of the last two results depend critically on the existence of "efficient" self-interpreters, whose overhead is *independent* of the program being interpreted.

Some Remarks on Higher-Type Computational Complexity

James S. Royer
Syracuse University

We sketch some of the key ideas of current work on the computational complexity of higher-type functionals and operators. In particular, we discuss the Kapron-Cook machine model for "type-2 polynomial time" and the difficulties that occur when one tries to lift this model to type-3.

Some Remarks on Pippenger's Separation of Pure and Impure LISP

Amir M. Ben-Amram
Academic College, Tel-Aviv

A question which intrigued programmers for a long time was: Is LISP with side-effects actually stronger than the pure-functional language? Pippenger gave, in '96, the first proof of such a separation. The talk consists of a presentation of his work followed by several comments on possible extensions and remaining (or ensuing) open problems.

More Haste, Less Speed: Lazy vs. Eager Evaluation

Oege de Moor
Oxford University
(This is joint work with Richard Bird and Geraint Jones.)

We prove that there exists no complexity-preserving transformation from lazy functional programming to eager functional programming. The proof consists of a Haskell program for Pippenger's problem.

Abstract State Machines and Linear Time Hierarchies

Yuri Gurevich
University of Michigan
(This is joint work with Andreas Blass.)

We generalize Neil Jones' *Linear Time Hierarchy Theorem* to Random Access Machines and (which is harder) to Abstract State Machines.

Normalization by Evaluation

Helmut Schwichtenberg
Universität München
(This is joint work with Ulrick Berger and Matthias Eberl.)

We extend normalization by evaluation (first presented in LICS '91) from pure, typed λ-calculus to general higher-type rewriting systems.

Improvement Theory and Its Applications

David Sands
Chalmers University

Improvement theory is a variant of the standard theories of observational approximation (or equivalence) in which the basic observations made of a program's execution include some intensional information about, for example, the program's computational cost. One program is an improvement of another if its execution is no less costly in any program context. In this talk we give an overview of our work on the improvement theory and its applications. Applications include reasoning about time properties of functional programs, and proving the correctness of program transformation methods.

Computer Programming as Mathematics

Paul J. Voda
Comenius University

This is basically the philosophy of the programming language and proof system CL (pairing, Clausal Language, Proof system) that has a precise mathematical characterization of what the functions definable in CL (unary primitive recursive functions) are and the kind of theorems about them that can be proved in the formal system, i.e., what the strength of the formal system is. This turns out to be the $I\Sigma_1$-fragment of Peano arithmetic. The language and proof system CL is extremely simple to justify. On the other hand, the characterization is quite involved and difficult because there is an inherent trade-off between computer programming and proving properties of programs. For efficient programming one needs quite a fine way of controlling the execution speed of programs (i.e., functions) by means of rich recursive and other control instructions. It turns out that the more efficient the function is, the harder it is to prove its properties. This is because the proofs call for quite a rich set of induction and other proof rules. To show that all such rules are admissible in the $I\Sigma_1$-fragment of Peano arithmetic requires advanced proof and recursion-theoretical techniques. Basically, this involves the theorem of R. Peter on simple (i.e., 1-fold) nested recursive definitions and the theorem of Parsons on admissibility of Π_2-rules in $I\Sigma_1$ arithmetic.

Ramified Recurrence and Alternation

Daniel Leivant
Indiana University

(No abstract submitted.)

A Survey of Efficiency-Increasing Transformation Techniques

Bernhard Möller
Universität Augsburg

Program transformations serve a dual purpose:

1. Constructing first correct implementations from formal specifications
2. Improving the efficiency of such implementations concerning
   - asymptotic time complexity (A)
   - constant factors (C)
   - space complexity (S)

In this talk we sketch the most important strategies for the latter aspect using some simple examples. The most important techniques, with an indication of their main improvement potentials, are:

I. Control fusion, both parallel (tupling (C)) and sequentially (deforestation (C,S))

II. Hornering and strength reduction (formal differentiation, Delta-optimization) (A,C)

III. Memoization and pre-computation (A)

IV. Change of data structure (A,S)

V. Re-use of variables (S)

Complexity Analysis of Logic Programs Based on Ordered Resolution

David Basin
Universität Freiburg

We provide a new method for classifying the time complexity of decision problems that are presented as sets of clauses.

We define order locality to be a property of clauses relative to a term ordering. This property is a kind of generalization of the subformula property for proofs where terms arising in proofs are bounded, under the given ordering, by terms appearing in the goal clause. We show that when a clause set is order local, then the complexity of its ground entailment problem is a function of its structure (e.g., full versus Horn clauses), and the ordering used. We prove that, in many cases, order locality is equivalent to a clause set being saturated under ordered resolution. This provides a means of using standard resolution theorem provers for testing order locality and transforming non-local clause sets into local ones. We have used the Saturate system to automatically establish complexity bounds for a number of nontrivial entailment problems relative to complexity classes which include polynomial and exponential time and co-NP.

Semantics, Complexity, and Optimization in Query Languages

Val Tannen
University of Pennsylvania

I surveyed work by several people in the design and analysis of query languages for complex values. The approach is based on the type structure of complex values. The base language, the NRA (Nested Relational Algebra, *not* National Rifle Association!) is a well-behaved conservative extension of first-order logic. Extended with forms of structural recursion over sets, it captures the classes PTIME and NC over ordered sets. I also survey some negative results, in particular some EXPSPACE lower-bounds that follow from natural conditions on the operational semantics of the languages.

Collaborators: P. Buneman, A. Deutsch, K. Lellehi, S. Nequi, L. Popa, D. Sucia, L. Wong

Soft Branching and Soft Correctness for Gains in Efficiency

Arnold Schönhage
Universität Bonn

Examples from my implementation work on fast numerical algorithms illustrate the usefulness of "soft branching"; in branching, a program may continue this (A) or that (B) path of execution on condition of fulfilling α, or β, respectively, with uncertain outcome in cases with both α and β being satisfied.

"Soft correctness" is about one attempt to formalize time savings obtained by relaxing the correctness requirements for subroutine calls.

Expressiveness of DATALOG Circuits (DAC)

Irène Guessarian
Université de Paris VII
(This is joint work with Foto Afrati and Michel de Rougemont.)

We define a new logic query language called DAC, extending DATALOG. We show that there are queries which are DAC-expressible, but not DATALOG-expressible. We infer various strict hierarchies of DAC program classes obtained by allowing more rapidly growing functions in the bound parameters that define these classes.

Operational Models for Compiler Verification

Egon Börger
Universita di Pisa

We show how the use of operational models, enhanced by appropriate abstraction and refinement mechanisms, helps to bridge the gap between program semantics as seen by the programmer and its implementation on (virtual) machines. The approach allows us to break the complexity of compiler construction by decomposition into horizontal and vertical components and to support the correctness of the translation by rigorous (mathematical and machine checkable) reasoning. Our method uses Gurevich's ASMs (Abstract State Machines) and is illustrated through our work on proven-to-be-correct compilation schemes for PROLOG programs to WAM code and the extension to PROTOS-L and CLP(R) programs and their compilation on the PAM and the CLAM, respectively, of OCCAM programs to TRANSUTER code, and of JAVA programs to Java VM code.

Computer Experiments with Levin's Search Theorem

Niels H. Christensen
University of Copenhagen

We outline the central construction in the proof of Levin's Search Theorem. This theorem asserts, for a wide set of search problems, the existence of a solving algorithm that is time-optimal up to a constant factor. We also consider new variants of the construction. In touching upon a practical implementation of Levin's algorithm, we discuss the behavior of a program that enumerates the language generated by a given context-free grammar in constant time per expression. Finally, we discuss observations on the behavior of the implementation of Levin's algorithm.

Type-2 NC Functionals

Peter Clote
Universität München

Using the SIMD (single instruction, multiple data stream) model of parallel computation, the class NC is defined as the collection of functions computable in polylogarithmic time $\bigcup_k (\log n)^k$ with polynomially many $\bigcup_k n^k$ active processors. We define an oracle extension of the parallel random access machine to model type-2 parallel computation, and define second-order polynomials, and so type-2 NC. A function algebra for type-2 functionals is introduced, and it is proved that $\lambda n, f.F(f,x)$ is in NC if and only if $\lambda n, f.F(f,x)$ belongs to the function algebra.

Refinement Type Inference Using Sequential Algorithms

Denis Dancanet
Morgan Stanley & Co.

We present a new application of Berry and Curien's intensional semantics of sequential algorithms on concrete data structures and its related programming language, CDS0. We define a typing system based on concrete data structures and featuring recursive types, subtyping, intersection types, polymorphism, and overloading. Then we translate programs written in a high-level language to CDS0 and use the precise information on the dependence of pieces of output on pieces of input provided by sequential algorithms to derive very detailed types that are similar to Freeman and Pfenning's refinement types.

Safe Recursion with Higher Types, Lists, and Polymorphism

Martin Hofmann
University of Edinburgh

We introduce a functional programming language with the property that all definable functions are polynomial-time computable. This is achieved by imposing a modal-linear typing discipline inspired by Bellantoni and Cook's safe-recursion. Our programming language extents safe-recursion, à la Bellantoni and Cook, with data structures such as lists and trees, polymorphism, and higher-order result types. The proof that all definable functions are polynomial-time computable proceeds by constructing a model in which denotations are PTIME functions.

Resource-Bounded Notions of Continuity

Bruce Kapron
University of Victoria

We propose two definitions of poly-bound continuity for type-two (total) functionals, one based on bounding by second-order polynomials and one based on bounding by feasible functionals. These two definitions are *not* equivalent. We show that the latter class can be given a sequential characterisation (using decision trees), while the former contains functionals which cannot be computed by bounded-depth decision trees.

Characterising Polytime Through Higher-Type Recursion

Karl-Heinz Niggl
Universität München

(This is joint work with S. Bellantoni and H. Schwichtenberg.)

It is well known that by a single use of higher-type recursion on notation one can define the Kalmar elementary functions. This is due to a nested non-linear use of the "previous function" in the step-term of the recursion.

In this talk it is shown how to restrict recursion on notation in all finite types so as to characterise the polynomial-time computable functions. The restrictions are obtained by enriching the type structure with $!\sigma$ for any type σ, and by adding concepts of linearity to the lambda-calculus.

More precisely, a calculus is presented that supports recursion on notation in all finite types, and which is closed under reduction. Terms are strongly normalising with uniquely determined normal forms. In particular, every closed term of ground-type reduces to a numeral. In that way the system of "RNA-terms" can be considered a simple functional programming language. Moreover, it is shown that there is a polynomial-time evaluation strategy to compute normal forms.

CONS-Free Programs with Tree Input

Holger Petersen
Universität Stuttgart

(This is joint work with Amir M. Ben-Amram.)

We investigate programs operating on LISP-style input data that may not allocate additional storage. For programs without a test of pointer equality we obtain a strict hierarchy of sets accepted by deterministic, non-deterministic, and recursive programs. We also show that the classes accepted by programs with the ability to test pointers for equality are closely related to well-known complexity classes. For these classes, strictness of the hierarchy is an important open problem.

Complexity Spaces

Michel Schellekens
Universität Siegen

The theory of complexity spaces allows one to use classical semantic techniques, such as fixed-point analysis, to analyse the complexity of algorithms. We illustrate this for the class of divide-and-conquer algorithms. The theory is situated in the context of quantitative domain theory.

On Functions Preserving Levels of Approximation

Dieter Spreen
Universität Siegen

DI-domains are enriched by a family of projections that assign to each point a sequence of canonical approximations. The morphisms are stable maps that preserve the levels of approximation generated by the projections. For the computation of an approximation of a given level of the output of at most the same level of approximation, it is shown that the category of these domains and maps is Cartesian-closed. The set of morphisms between such domains is a domain of the same kind, but turns out not to be an exponent in the category.

Fishing for Speed

Barry Jay
University of Technology, Sydney

FISh is a new programming language that uses static *shape analysis* to determine the shape, i.e., the number of dimensions, size of each dimension, of every array appearing during execution. As well as detecting all shape errors statically, this analysis supports the construction of polymorphic, higher-order functions from simple imperative procedures, so that FISh programs execute at speeds comparable to those of C and Fortran.

Functional	=	Imperative	+	Shape
F		I		Sh

Resolutions, Towers, and Low Degree Type-2 Functionals

J. R. Otto

In our thesis we represented the linear space functions by a category initial among those with recursion compatible with a 2-comprehension. (There, also using V- and 3-comprehensions, we similarly characterized the multi-tape linear time, Ptime, Pspace, and Kalmar elementary functions. In particular, we provided a semantics for the tiers of S. Bellantoni and D. Leivant.)

More recently, we refined the construction of initial categories by collapsing bicategories of resolutions. These resolutions are in a logic programming sense.

In work in progress, we use N-comprehensions to study rank-2 linear space functionals. (For example, with **Set** the category of small sets, N the natural numbers, and $\mathbf{Set}^N$ the functor category, the partially ordered monoid of eventually fixed actions on N induces an N-comprehension on $\mathbf{Set}^N$.) Following J. Royer, R. Irwin, and B. Kapron, we stratify on the depth of rank 2 functionals. Not following them, we stratify inputs. (As we think of numbers as big sums of terminal objects, we would prefer to stratify inputs rather than outputs.) Thus, so that Y^X is below, rather than above, X and Y in a tower, we abstract by fragments of the local sections variant of local smallness. (For example, $\mathbf{Set}^N$ yields a tower of fibrations. Following T. Streicher, for a fibration to be a 2-comprehension is the global sections fragment of local smallness. Indeed, fibrations hide 2-comprehensions. However, the 2-comprehension we used to characterize the multi-tape linear time functions does not appear to be a fibration.)