Programs with Recursively Defined Data Structures Using Pointers

Organizers:

M. Sagiv (Tel Aviv), M. Schwartzbach (Aarhus)
K. Weihe (Konstanz), K. Mehlhorn (Saarbrücken)

April 20–24, 1998

1 Introduction

The theme of this seminar was to study programs that manipulate dynamically allocated data. The attendees were researchers from three different areas:

- Design and implementation of combinatorial (e.g., graph) algorithms with a heavy usage of dynamically allocated memory. Many of these algorithms are currently implemented in C++, e.g., see [MN95].
- Verification of partial correctness of such implementations, e.g., proving that a program only refers to allocated memory cells. Proving that a program does not create memory leaks. By convention, we call these *cleanness* checks since they must hold for every reasonable program as opposed to correctness, which is program specific. The hope is that some of these tests can be carried out by future compilers.
- Compiler optimization to speed up the execution time of such programs. For example prefetching of linked data structures can improve performance by 45%, see [LM96].

Three panels of open problems were held (summarized in Section 3). Few of the open problems already initiated research, e.g., in the area of improving locality of programs that manipulate dynamically allocated memory and in the area of automatically eliminating "checking code" that validates that certain invariants are maintained after a library operation, e.g., that the insertion of a new edge into a directed graph, maintains the Euler equation.

33 talks were given presenting the state of art techniques in these areas (the schedule is given in Section 2 and the abstract is summarized in Section 4). The talks were very interesting and accessible to all groups. Some of the talks presented solutions to open problems. One of the most interesting subject studied is the treatment of memory hierarchy in general and cache in particular. Finally, one talk by John L. Ross presented a solution to a problem posed by Luddy Harrison at the Dagstuhl seminar No. 9535, held in August 1995.

The schedule of the workshop was not fixed on the usual daily basis. Instead, we did an experiment on Monday afternoon: the schedule was fixed in an additional "problem session."

In this session speakers from different fields were encouraged to discover relations between their works (the preceding "five minutes madness" session helped a lot) and to build a session. Moreover, we included time for questions at the end of these and other sessions to review all talks in summary. In our opinion, some of the most interesting insights came about this way.

Preliminary conclusions of the seminar are summarized in Section 5.

Acknowledgments

Thanks to Reinhard Wilhelm for inviting us to organize this Dagstuhl seminar for many insights that contribute to the success of this workshop; to Angelika Müller and Annette Bayer in the Dagstuhl office in Saarbrücken and the staff at Dagstuhl for ensuring that everything ran perfectly; to the participants for making the seminar lively, fruitful and of very high quality; and to Markus Mohnen for coordinating the production of this report.

2 Schedule

Monday, April 20

9:00– 9:05 Reinhard Willhelm: *Welcome and Opening*

9:05– 9:45 Karsten Weihe: *Challenges from the Algorithms and Data Structures Frontier*

9:45–10:40 Michael I. Schwartzbach: *A Brief Survey on Pointer Verification*

10:40–11:00 Coffee Break

11:00–11:40 Hanne Riiis Nelson: *An Overview of Program Analysis Techniques*

11:40–12:15 Mooly Sagiv: *An Overview on Pointer Analysis*

12:15–14:00 Lunch

14:00–15:00 "Five Minutes Madness": Everybody presents her-/himself.

15:00–16:00 Session on the Workshop Schedule for the rest of the week: *Chair*: Karsten Weihe

16:00–17:30 First Open Problem Session: *Chair*: Michael I. Schwartzbach

17:30–18:00 Break

18:00–20:00 Dinner

20:00–21:00 Software Demos:

- Florian Martin: *Shape Analysis in PAG*
- Uwe Assman: *PRISMA, an Interprocedural Chase/Wegman/Zadeck Heap Analyzer*

Tuesday, April 21

9:00– 9:45 Amer Diwan: *Type-Based Alias Analysis*

9:45–10:30 Mooly Sagiv: *Detecting Memory Errors via Static Analysis*

10:30–10:50 Coffee Break

10:50–11:35 Michael I. Schwartzbach: *Automatic Verification of Pointer Programs using Monadic Second-Order Logic*

11:35–12:10 Andreas Podelski: *Abstract Debugging of Programs with Tree-like Data Structures is Model Checking of Pushdown Systems*

12:10–13:50 Lunch

13:50–14:35 Shai Rubin: *Virtual Cache Line: A New Technique to Improve Cache Exploitation for Recursive Data Structures*

14:35–15:35 Trishul Chilimbi: *Cache-Conscious Data Structures*

15:35–16:20 Claudia Leopold: *Arranging Statements and Data of Program Instances for Locality*

16:20–16:40 Coffee Break

16:40–17:10 Reinhard Willhelm: *Predicting Cache Behavior Fast and Efficiently*

17:10–17:25 Rudolf Fleischer: *Caching — A Theoretician's View*

17:25–17:55 David Bernstein: *FDPR - A Postpass Code Locality Optimization Tool*

17:55–20:00 Dinner

20:00–21:00 Problem Session on Locality

Wednesday, April 22

8:45– 9:40 Andreas Crauser, Ulrich Meyer, Michael Seel: *LEDA - Library of Efficient Data Types and Algorithms*

9:40–10:10 Lutz Kettner: *Designing a Polyhedral Surface Data-Structure in C++*

10:10–10:20 Discussion of the Last Two Talks

10:20–10:40 Coffee Break

10:40–11:20 Michael Hind: *Empirically Comparing Interprocedural Pointer Analysis*

11:20–12:00 Laurie Hendren: *Putting Heap Analysis to Work*

12:00–12:10 Discussion of the Last Two Talks

12:10–13:15 Lunch

13:15–13:30 Preparing for Hike and Excursion

13:30–18:00 Hike and Excursion

18:00–19:30 Dinner

19:30–20:30 Session on Algorithm and Data Structures Problems or
What the "Client Group" Can Do for the "Server Groups": *Chair*: Karsten Weihe

Thursday, April 23

8:45–10:15 Anne Rogers and Laurie Hendren: *Supporting Dynamic Data Structures on Distributed Memory Machines*

10:15–10:25 Discussion of the Last Two Talks

10:25–10:45 Coffee Break

10:45–11:35 Thomas Reps: *Parametric Shape Analysis via 3-Valued Logic*

11:35–12:10 Martin Trapp: *Heap–SSA*

12:10–12:20 Discussion of the Last Two Talks

12:20–14:00 Lunch

14:00–14:40 John Ross: *Building a Bridge Between Pointer Aliases and Program Dependences*

14:40–15:10 Hanne Riis Nielson: *Store Models Challenge the Semanticists Toolbox*

15:10–15:20 Discussion of the Last Two Talks

15:20–15:40 Trishul Chilimbi (incl. Discussion): *An Answer to Open Question 18 for a specific model*

15:40–16:10 Coffee Break

16:10–16:50 Markus Mohnen: *The Expressive Power of Escape Analysis*

16:50–17:20 Helmut Seidl: *Compile-Time Garbage Collection for Object-Oriented Languages*

17:20–17:30 Discussion of the Last Two Talks

17:30–17:50 Claudia Leopold: *Cache-Conscious Sorting*

18:00–19:30 Dinner

19:30–20:30 Second General Open Problem Session: *Chair*: David Bernstein

Friday, April 24

8:45– 8:50 General Announcements

8:50– 9:25 Karsten Weihe: *Statically Type-Safe, Run-Time Dynamic Interfaces to Tabular Data Structures*

9:25– 9:55 Alex Bijlsma: *Construction of Pointer Programs*

9:55–10:15 Coffee Break

10:15–11:05 Bernhard Möller: *Calculating with Pointer Structures*

11:05–11:15 Discussion of the Last Two Talks

11:15–12:05 Thomas Reps: *Program Analysis via Graph Reachability*

12:05–12:10 Discussion

12:10–12:15 Good-Bye

12:15–13:30 Lunch

3 Open Problems

We spent three evenings in understanding the open problems in this field. There are number of reasons why dynamically allocated memory will be more important in the research community and in the industry in the near future:

- Stack and statically allocated storage cannot be used for many of the combinatorial algorithms.
- It is well known that programs that manipulate dynamically allocated memory are hard to debug, prove correct, and optimize.

- Many of the pragmatic open compiler research problems related to stack and statically allocated memory were already solved. For example, Michael Hind presented in the seminar an empirical study which indicates that in many cases, flow insensitive analysis of stack allocated pointers provides information that is as precise as the flow sensitive approaches.
- There are many opportunities to speed up the execution of programs that manipulate dynamically allocated memory both on existing and on future architectures.
- Programming languages like Java create many challenges by opening many of the research ideas such as safe pointer dereferences and garbage collection to a wide audience.

Below we summarize the open problems that we have identified. Each of this problem is characterized as follows:

C A conceptual problem

P A pragmatic problem

A An algorithmic problem

3.1 List of Open Problems

1. (**C**) Can (explicit) pointers (sometimes) be avoided by introducing stronger language concepts?
   (a) Eliminate pointer arithmetic and explicit addressing ($C \rightarrow Java$)
   (b) Eliminate Java style pointers (references)
2. (**C+P**) Can verification of cleanness or even correctness of programs with pointers be feasible?
   (a) Automatically
   (b) With user information + how much?
      i. Interactively
      ii. Annotations as comments (a-la-LCLint [Eva96a, Eva96b])
      iii. Program checking, e.g., using ANSI C assert — can the compiler optimize a way or reorder these checks?
      iv. Programming language styles, e.g., isolate pointer stuff in a separate function and use smart pointers in the rest of the code — can the compiler eliminate the cost via optimizations?
3. (**C**) How to do run-time prediction., e.g., of cache models?
4. (**C**) Can templates be used to verify libraries in a "generic way" without investigating the application programs?
5. (**C**) Can program analysis be used to verify that reasonable graph operations preserve Euler equations?

6. (**P**) How much can pointer analysis buy for classical machine (in)dependent optimizations? (Amer Diwan answered that for a particular optimization of redundant loads)
   (a) Which pointer analysis is best for a given application?
7. (**C**) Is there a way to improve the process of testing using information obtained by program analysis/verification?
8. (**A+P**) How can the memory efficiency of programs with pointers be improved?
9. (**C+A+P**) How can locality of references be expressed, proved or analyzed?
10. (**P**) Can flow sensitive algorithms for analyzing dynamic allocated data structure scale up for large programs?
11. (**P**) Is there a benchmark that is preferable for pointer verification and/or analysis (Spec [SPE92], Olden [RCRH95], LEDA, Todd Austin's pointer intensive benchmark)?
12. (**C**) User interface to allow tuning the garbage collection performance (w/o changing program semantics).
13. (**P**) Use cheap analysis (e.g., linear time point-to) in order to improve the running time of a more expensive, e.g., flow sensitive analysis?
14. (**C**) Can code idioms that identify special data structure manipulations, e.g., can insertions into a single linked list be identified via program analysis (in order to be replaced by a more efficient implementation)?
15. (**A**) Demand-driven pointer analysis.
16. (**C**) Pointer analysis in "open" programming environments (languages), e.g., Java dynamic loading.
17. Can domain specific information be used to improve efficiency of the memory subsystem performance (by a user or compiler)
18. Is there a formula that predicts the number of cache misses for a given program. Trishul Chilimbi partially answered this question.

References

[Eva96a] D. Evans. *LCLint User's Guide*, 1996. Available at http://larch-www.lcs.mit.edu:8001/larch/lclint/guide/.

[Eva96b] D. Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Languages Design and Implementation*, 1996. Available at http://larch-www.lcs.mit.edu:8001/∼evs/pldi96-abstract.html.

[LM96] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

[MN95] K. Melhorn and S. Nähr. LDEA: A platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, 1995.

[RCRH95] Anne Rogers, Martin C. Carlisle, John Reppy, and Laurie Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995. Article and benchmark suit available at http://www.cs.princeton.edu/∼mcc/olden.html.

[SPE92] SPEC Component CPU Integer Release 2/1992 (Cint92). Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1992.

4 Abstracts of Talks

Challenges from the Algorithms and Data Structures Frontier

Karsten Weihe (Universität Konstanz)

In this seminar, the participants doing research on efficient implementations of sophisticated algorithms and data structures are the "clients" who pose problems, and the other two groups (hopefully) offer solutions, which means they are the "servers."

In this talk, I tried to formulate a few "challenges" to the other groups, that is open problems whose solutions are possibly found in their research areas.

A Brief Survey on Pointer Verification

Michael I. Schwartzbach (University of Aarhus)

Pointer programming is difficult and risky, but may become safer through static analysis that seeks detect pointer errors at compile-time. This survey describes the fundamental problems in such analyses and discusses a number of tools that offer practical solutions for at least a subset of those. The conclusion is that such tools appear to be feasible, but must be carefully designed to strike a compromise between precision and efficiency.

An Overview of Program Analysis Techniques

Hanne Riis Nielson (Aarhus University)

Program analysis offers static techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically during the execution of programs. The analysis problem clearly depends on the application one has in mind and it is here important to clarify what demands it put to the analysis and thereby to the technique used to specify it:

- Generality: to which class of programming languages and program analysis problems can the technique be applied.
- Precision: does the technique inherently limit the precision of the specified analyses and can it be used to guarantee a certain level of precision for a given class of programs.

- Efficiency: does the technique inherently limit the efficiency of the specified analyses and does the technique scale up for large programs.
- Correctness: does the technique offer results that lighten the burden of establishing correctness results for the analyses.

Over the years a number of program analysis techniques have been developed with different strengths and weaknesses with respect to the above criteria. In this talk I give an overview of some of the main techniques. This includes the traditional intra- and inter-procedural data flow approaches, the constraint-based approaches to control flow analysis and the main ingredients of abstract interpretation: Galois connections, a catalogue of techniques for combining Galois connections, the approximation technique of widening and the technique of inducing one analysis from another. Approaches based on denotational semantics, type and effect systems and more general logical approaches are briefly mentioned.

An Overview of Pointer Analysis

Mooly Sagiv (Tel-Aviv University)

We describe different variants of the problem of statically analyzing pointer references. These problems have many classifications including:

May-alias analysis: locating pointer access paths that always denote different locations, e.g., proving that two pointer variables x and y denote different locations at every execution path leading to a given program point.

Shape-analysis: locating the shape of the data structures manipulated by the program, e.g., discriminating between a list and a cyclic list.

We sketch some of the many potential applications of pointer analysis for compiler optimizations and program understanding.

A full version of the talk can be retrieved from: http://www.math.tau.ac.il/~sagiv/DAG98/overview.ps

Type-Based Alias Analysis

Amer Diwan (Stanford University)

My talk (presenting work by Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss) evaluated three alias analyses based on programming language types. The first analysis uses type compatibility to determine aliases. The second extends the first by using additional high-level information such as field names. The third extends the second with a flow-insensitive analysis. Although other researchers suggests using types to disambiguate memory references, none evaluates its effectiveness. We present results for both static and dynamic evaluations of type-based alias analyses for Modula-3, a statically-typed type-safe language. The static analysis reveals that type compatibility alone yields a very imprecise alias analysis, but the other two analyses significantly improve alias precision. We use redundant load elimination (Redundant load elimination) to demonstrate the effectiveness of the three alias algorithms in terms of the opportunities for optimization, the impact on simulated execution times, and to compute an upper bound on what a perfect alias analysis would yield. We show modest dynamic improvements for (Redundant load elimination), and more surprisingly, that on average our alias analysis is within 2.5% of a perfect alias analysis with respect to Redundant load elimination on 8 Modula-3 programs. These results illustrate that to explore thoroughly the effectiveness of alias analyses, researchers need static, dynamic, and upper-bound analysis. In addition, we show that for type-safe languages like Modula-3 and Java, a fast and simple alias analysis may be sufficient for many applications.

Detecting Memory Errors via Static Analysis

Mooly Sagiv (Tel Aviv University)

Programs which manipulate pointers are hard to debug. Pointer analysis algorithms (originally aimed at optimizing compilers) may provide some remedy by identifying potential errors such as dereferencing `NULL` pointers by statically analyzing the behavior of programs on all their input data.

Our goal is to identify the "core program analysis techniques" that can be used when developing realistic tools which do not generate too many false alarms. It is an open question if exists a conservative technique that will yield only a modest number of false alarms, and if so, if it will scale for large programs. Our preliminary experience indicates that the following techniques are necessary: (i) finding aliases between pointers, (ii) flow sensitive techniques that account for the program control flow constructs, (iii) partial interpretation of conditional statements, (iv) analysis of the relationships between pointers, and sometimes (v) analysis of the underlying data structures manipulated by the C program.

We show that a combination of these techniques yields better results than those achieved by state of art tools.

This is a joint work with Nurit Dor and Micky Rodeh which will appear in PASTE'98 http://www.math.tau.ac.il/~sagiv/paste98.ps.

Automatic Verification of Pointer Programs using Monadic Second-Order Logic

Michael I. Schwartzbach (University of Aarhus)

Joint work with Jakob L. Jensen and Michael E. Jørgensen.

We present a technique for automatic verification of pointer programs based on a decision procedure for the monadic second-order logic on finite strings.

We are concerned with a while-fragment of Pascal, which includes recursively-defined pointer structures but excludes pointer arithmetic.

We define a logic of stores with interesting basic predicates such as pointer equality, tests for nil pointers, and garbage cells, as well as reachability along pointers.

We present a complete decision procedure for Hoare triples based on this logic over loop-free code. Combined with explicit loop invariants, the decision procedure allows us to answer surprisingly detailed questions about small but non-trivial programs. If a program fails to satisfy a certain property, then we can automatically supply an initial store that provides a counterexample.

Our technique has been fully and efficiently implemented for linear linked lists, and extends in principle to tree structures. The resulting system can be used to verify extensive properties of smaller pointer programs and could be particularly useful in a teaching environment.

Abstract Debugging of Programs with Tree-like Data Structures is Model Checking of Pushdown Systems

Andreas Podelski (Max-Planck-Institut für Informatik)

Abstract debugging (a term coined by Francois Bourdoncle) stands for the inference of necessary conditions for the correctness of programs wrt. invariant or intermittent assertions (initial values not satisfying the conditions are bugs). In the system considered by Bourdoncle, assertions could be used to described interval ranges of program variables; the necessary conditions were inferred using static analysis based on abstract interpretation techniques. In a first step towards extending this system, we analyze that abstract debugging is abstract model checking. That is, a conservative approximation of the initial states satisfying certain temporal logic properties is computed. We then ask how we can extend the framework to programs over pointers; i.e., how can we formulate assertions over "intervals" of pointers, and what are the corresponding abstract interpretation techniques. We restrict this question to tree-like data structures as they were considered in the first shape-analysis papers by Neil Jones and others. We establish a connection between that work on shape-analysis and model checking of pushdown systems. An extension of pushdown systems is used as an abstraction of while programs over trees. In summary, we derive that *abstract debugging of while programs over trees is model checking of extended pushdown systems.*

Virtual Cache Line: A New Technique to Improve Cache Exploitation for Recursive Data Structures

Shai Rubin (The Technion)

Joint work with David Bernstein (IBM Research Lab) and Michael Rodeh (The Technion and IBM Research Lab).

Recursive data structures (lists, trees, graphs, etc.) are used throughout scientific and commercial software. The common approach is to allocate storage to the individual nodes of such structures dynamically, maintaining the logical connection between them via pointers. Once such a data structure goes through a sequence of updates (inserts and deletes), it may get scattered all over memory yielding poor spatial locality, which in turn introduces many cache misses. Furthermore, such updates invoke the memory manager, quite often degrading performance even further. In this paper we present the new concept of Virtual Cache Lines (VCLs). This machine independent concept is designed to support reducing the cache miss ratio and improving memory utilization when recursive data types are used. Basically, the mechanism keeps groups of consecutive nodes in close proximity, forming virtual cache lines, while allowing the groups to be stored arbitrarily far away from each other. Virtual cache lines increase the spatial locality of the given data-structure resulting in better locality of reference. Furthermore, since the spatial locality is improved, software prefetching becomes much more attractive. Indeed, we also present two software prefetching algorithms that can be used when dealing with VCLs resulting in even higher data cache performance. Our results show that the average performance of linked list operations, like scan, insert, and delete can be improved by more than 200% even in architectures that do not support prefetching, like the Intel Pentium. Moreover, when using prefetching (e.g. on the IBM PowerPC) one can gain additional 200% improvement.

We believe that given a program which manipulates certain recursive data structures, compilers will be able to generate VCL-based code. This observation builds on recent results in shape analysis. Also, until this vision becomes true, VCLs can be used to build more efficient user libraries, operating-systems and applications programs.

Cache-Conscious Data Structures

Trishul Chilimbi (University of Wisconsin-Madison)

Joint work with James R. Larus and Mark D. Hill.

Processor and memory technology trends show a continual increase in the cost of accessing main memory. Machine designers have tried to mitigate the effect of this trend through hardware and software prefetching, multiple levels of cache, non-blocking caches, dynamic instruction scheduling, speculative execution, etc.

These techniques unfortunately, have only been partially successful for pointer-manipulating programs. This talk explores the complementary approach of redesigning and reorganizing data structures to improve cache locality. Pointer-based structures allow data to be placed in arbitrary locations in memory, and consequently in a cache. This freedom enables a programmer to improve performance by applying techniques such as clustering, compression, and coloring.

To reduce the cost and complexity of applying these techniques, this talk also presents two semi-automatic techniques for implementing cache-conscious data structures with minimal programmer effort. The first reorganizes tree-like data structures to improve locality. The second is a cache-conscious heap allocator. Our evaluations – with a tree microbenchmark, four Olden benchmarks, and two large applications – show that cache-conscious data structures, including those implemented by semi-automatic techniques, can produce large performance improvements and outperform hardware and software prefetching.

Arranging Statements and Data of Program Instances for Locality

Claudia Leopold (Universität Jena)

In memory hierarchies, programs can be speeded up by increasing their degree of locality. The talk suggested a semi–automatic method for locality optimization that is based on the human approach of considering several small program instances, optimizing the instances for locality, and generalizing the structure of the solutions to the program.

Emphasis was given to a local search algorithm that automatically optimizes the locality of program instances. It takes as input a set of statement instances where each statement instance is characterized by a sequence of data accesses. The algorithm orders the statement instances, thereby respecting data dependencies, and groups the data into blocks of memory. It uses a novel objective function that reflects the 'the–closer–the–better' principle of the intuitive notion of locality. Advantages of this function over a function based on communication costs were shown.

Finally, I gave experimental results that compare so–optimized instances with corresponding instances derived from compiler–optimized programs. The results indicate that the optimization algorithm produces high–quality output. I also illustrated the semi–automatic method with examples and showed that the method can speed up programs.

For further information see my homepage: http://www.minet.uni-jena.de/www/fakultaet/ips/claudia.html

Predicting Cache Behavior Fast and Efficiently

Reinhard Wilhelm (Universität des Saarlandes)

Joitn work with Christian Ferdinand and Florian Martin (Universität des Saarlandes).

There is a tremendous gap between the cycle times of modern microprocessors and the access times of main memory. Caches are used to overcome this gap in virtually all performance-oriented processors (including high-performance microcontrollers and DSPs).

Hard real-time systems have specified deadlines for their tasks. It is the duty of the developer to guarantee that the tasks making up the system will always meet the deadlines specified. When it comes to processors with caches, computing sharp upper limits on the worst-case execution time is of critical importance.

The widely used classical methods of predicting execution times are not generally applicable. Software monitoring or the dual loop benchmark changes the code, in the process influencing cache behavior. Hardware simulation, emulation or direct measurement with logic analyzers can only determine the execution time for one input. This can generally not be used to infer the cache behavior for all possible inputs.

We have developed static analyses of programs which predict the program's behavior on the cache for a given architecture. In other words, these analyses classify most of the memory references as cache hits or misses. The analysis is generic, i.e. it can be easily instantiated using another cache architecture, and with moderate effort it can also be adapted to a new instruction set.

Experiments on a set of relevant programs, djpeg, fft, ndes, ... have shown that the best/worst-case cache behavior interval is very small, i.e. it corresponds to a relatively small portion of the actual execution time. By contrast, making the safe, yet for the most part unrealistic assumption that all memory references result in cache misses results in the execution time being overestimated by several hundred percent.

The analyses developed by us are

provably correct owing to our systematic approach based on abstract interpretation theory,

extremely precise due to a representation of the sets of possible cache content that preserves relevant information; this allows for exact predictions of cache behavior,

fast due to an efficient representation of the cache content and elaborate iteration algorithms,

generic i.e. easily adaptable to many cache architectures,

automatically generated from concise specifications using the Program Analyzer Generator PAG.

Caching — A Theoretician's View

Rudolf Fleischer (Universität Trier and Max-Planck-Institut für Informatik)

In this talk, I give a short overview of the theoretical questions arising in the study of caching. There are two main directions of research : Probabilistic analysis of caching strategies, supported by experimental data, and competitive analysis as a measure of worst-case performance. Whereas the former method faces the problem that reality does rarely behave according to a simple probability distribution, the latter method is often overly pessimistic. To get closer to reality, it has been tried to refine competitive analysis by assuming certain access patterns as given by the control flow graph of a program, for example.

FDPR - A Postpass Code Locality Optimization Tool

David Bernstein (IBM Haifa Research Lab)

Recent superscalar and VLIW processors feature multiple functional units which reduce the execution time of computational programs quite significantly. It turns out, however, that larger and larger fractions of execution time programs spend in accessing the memory hierarchy, specifically the instruction and data caches and the main memory. This becomes even more pronounced, as the increase in memory speed does not keep pace with superfast clock rates of recently announced microprocessors.

The focus of this work is to demonstrate how the code locality of application programs can be improved via profiling-based optimization. We describe a stand-alone optimization tool, called FDPR for Feedback Directed Program Restructuring, which can globally reorder application programs after they passed the compilation and linkage process. Currently, FDPR is operational on IBM's UNIX (AIX) and it can optimize XCOFF binary programs compiled with IBM's XL compilers.

Improved code locality resultant from FDPR optimization leads to less instruction cache and instruction TLB misses, as well as to a smaller number of page faults and reduced branch penalty. The performance improvements achieved by FDPR for SPECint92 benchmarks are in the range of 5% on average for the IBM PowerPC and POWER2 processors. For big programs, like the DB2/6000 database applications, the achieved performance gains are well beyond 20%.

LEDA - Library of Efficient Data Types and Algorithms

Andreas Crauser (MPI - Saarbrücken)

Joint work with Ulrich Meyer and Michael Seel (MPI - Saarbrücken).

We give an overview of the LEDA platform for combinatorial and geometric computing and an account of its development. The talk is an introduction to the functionality and applicability of LEDA as a programming toolbox. Moreover some programming concepts are presented which are of interest to the communities of this workshop.

For more information about LEDA please visit: http://www.mpi-sb.mpg.de/LEDA.

Computing in Secondary Memory - A Library Prototype

Andreas Crauser (MPI - Saarbrücken)

Data to be processed is growing so fast that secondary storage (disks, tapes) must be used for computation. In this talk we present the classical two-level memory model, introduced by Aggarwal and Vitter. In this model, our computer consists of a small and fast internal memory of size M and a large and slow secondary memory. data is exchanged between the two memories in blocks of size B, this is called I/O. Algorithmic performance in this model is measured by counting a) the number of I/Os, b) the CPU-time and c) the number of occupied blocks in secondary memory. We show that many internal memory data structures and algorithms fail if used in secondary memory. This is often the reason if they access secondary memory in an unstructured way by the use of pointers. To circumvent this problem, new algorithms and data structures must be introduced. We present our library prototype LEDA-SM which provides a collection of data structures and algorithms explicitely designed for secondary memory usage. This library is an extension of the LEDA library.

Single Source Shortest Path - The Quest for an I/O Efficient Algorithm and its Spinoffs

Ulrich Meyer (MPI - Saarbrücken)

After the introduction of LEDA-SM we present a case study for the Single Source Shortest Path (SSSP) problem in External Memory. The previously best solution [KS96] needs $\Theta(n) + O(m/(DB)\log_2 m/(DB))$ I/O for graphs having n nodes and m edges. It is an adaption of Dijkstra's Algorithm using I/O efficient data structures. We show how to divide Dijkstra's Algorithm into r phases such that the outgoing edges of all nodes removed from the queue during a single phase can be treated in parallel. This observation can be used to reduce $\Theta(n)$ I/O for accessing the adjacency lists to $O(n/D)$ I/O where D is restricted to $\min n/(r\log n), M/B$. For random graphs with random edge weights we prove that $r = O(n^{1/3})$ with high probability. A more aggressive variant of Dijkstra's Algorithm even achieves $r = O(\log^2 n)$ phases at the cost of $O(n)$ reinsertions. Thus, we are able to derive an algorithm that shows even better I/O performance although it scans the whole graph representation during each phase: $O(r(n+m)/(DB))$ I/O on random graphs with random edge weights with high probability, $D \leq \min n/(r\log n), M/B$. Our techniques also yield improvements for other models of computation: Sequentially, it is possible to compute SSSP in $O(n+m)$ time on the average (random graphs, random edge weights) using an appropriate bucket structure. Extending the ideas of the sequential solution we derive a PRAM algorithm which solves the SSSP in poly-logarithmic time and $O(n+m)$ work on the average. Both, sequential and parallel implementations show the practicality of our approaches. Part of these results will be included in the forthcoming proceedings of ESA '98.

Designing a Polyhedral Surface Data-Structure in C++

Lutz Kettner (ETH Zurich)

An overview of CGAL, the Computational Geometry Algorithms Library, was given. Design solutions for a program library were presented for combinatorial data structures, such as planar maps and polyhedral surfaces. Particular issues considered are flexibility, correctness, time and space efficiency, and ease-of-use with the focus on topological aspects of polyhedral surfaces. The design follows the generic programming paradigm known from the Standard Template Library (STL) for C++. The design consists of three layers: at the bottom the classes for vertices, halfedges and facets, in the middle the halfedge data-structure, and at the top the high-level, easy-to-use interface for polyhedral surfaces, which maintains combinatorial integrity by means of Euler-operators.

Library development poses the question to program verification, how to cope with separate compilation and the unkown application the library will be used for. The use of templates aggravates the situation, since library users could exchange internal parts of library components with the choice of template arguments. For compiler optimization arises interesting data-structures with pointers, beginning with highly regular Delaunay triangulations, continuing over planar maps and trapezoidal decompositions up to polyhedral surfaces. Typical algorithms would be ray shooting along a horizontal direction, along an arbitrary direction, or the inspection of the neighborhood for one or multiple (random) points.

Further references and the CGAL library can be found at http://www.cs.ruu.nl/CGAL/. A comprehensive directory of available source code in computational geometry is presented at http://www.geom.umn.edu/software/cglist/.

Empirically Comparing Interprocedural Pointer Alias Analyses

Michael Hind (SUNY at New Paltz and IBM Research)

Joint work with Anthony Pioli, Michael Burke, Paul Carini, and Jong-Deok Choi.

To enhance program optimization, a recent trend in program analysis has been to analyze whole programs, rather than each procedure in isolation. Although this approach can increase precision, it may also increase compilation time. In order for these interprocedural analysis techniques to make the transition from research efforts to production compilers, the cost/precision tradeoffs must be clearly understood.

This talk describes an empirical comparison of four context-insensitive pointer alias analysis algorithms that use varying degrees of flow-sensitivity: a flow-insensitive algorithm that tracks variables whose addresses were taken and stored, a flow-insensitive algorithm that computes a solution for each function, a variant of this algorithm that uses precomputed kill information, and a flow-sensitive algorithm. In addition to contrasting the precision and efficiency of these analyses, we describe implementation techniques and quantify their analysis-time speed-up.

Further details can be found in publications located at www.mcs.newpaltz.edu/~hind/papers

Putting Pointer Analysis to Work

Laurie Hendren (McGill University)

There has been a lot of work on pointer analysis, but relatively little research into the effectiveness of pointer analysis - how useful is it?

This talk presented the three major pointer analyses that have been implemented in the McCAT C compiler. Points-to analysis estimates pointer relationships for stack locations. Connection analysis estimates connection properties of heap-directed pointers. Shape analysis estimates the shape (TREE, DAG, CYCLIC) for heap structures accessible from each stack pointer.

Given the information computed by these pointer analyses, techniques were presented for computing read/write sets, extended SSA numbers and dependence testing. Based on this, techniques for optimizing memory references via loop-invariant removal, common sub-expression elimination and location invariant detection were presented.

A collection of pointer-intensive benchmarks showed speedups of 1-10speedup due to these memory optimizations.

Other applications discussed included improved array dependence testing, summarizing the effect of procedure calls, and McWeb - a web-based tool for browsing programs to display the results of pointer analyses.

This work was done as part of the McCAT compiler project at McGill University. This talk presented work done by Rakesh Ghiya, Maryam Emami and Christopher Lapkowski.

Supporting Dynamic Data Structures on Distributed Memory Machines

Laurie Hendren (McGill University) and Anne Rogers (AT&T Labs)

Compiling for distributed-memory machines has been a very active research area in recent years. Much of this work has concentrated on programs that use arrays as their primary data structures. To date, little work has been done to address the problem of supporting programs that use dynamic data structures. The techniques developed for supporting SPMD execution of array-based programs rely on the fact that arrays are statically defined and directly addressable. Recursive data structures do not have these properties, so new techniques must be developed. In this talk, we described two approaches— Olden and McCat Earth-C —for supporting programs that use pointer-based dynamic data structures.

Olden provides a pair of complementary mechanisms for managing remote data, software caching and computation migration, and introduces parallelism using a technique based on futures and lazy task creation. It also includes a compile-time heuristic for choosing between software caching and computation migration automatically. The talk discussed these mechanisms and included a report on experiments with ten benchmarks.

McCat Earth-C is targeted for a distributed-memory machine that has multi-threaded processors. In addition to managing remote references and synchronization, the Earth-C compiler must also generate static threads for the processors. The compiler uses stack and heap pointer analysis to improve: dependence testing, reduce pointer dereferences via common subexpression elimination and loop invariant removal, infer locality, and move and block communication. Overall these optimizations are designed to reduce communication and increase thread length. The talk included a report of experimental results on five benchmarks and a discussion of the relative importance of the optimizations.

Parametric Shape Analysis via 3-Valued Logic

Thomas Reps (University of Wisconsin)

We present a family of algorithms that are sometimes capable of determining "shape invariants" of programs that perform destructive updating on heap-allocated storage. The approach described is parametric: it provides the basis for generating a family of shape-analysis algorithms. This is achieved by applying the following principle of abstraction to stores:

Memory locations are partitioned into equivalence classes according to their sets of "abstraction–property" values. Every store is then represented (conservatively) by a condensed store in which each element of the condensed store represents an equivalence class.

By varying the parameter that controls abstraction — namely, the set of properties used for abstraction — one can specify different shape-analysis algorithms.

Condensation does not preserve (in)equality, and hence a single element in a condensed store may represents multiple memory locations of the concrete store. Because of this aspect of abstraction, the logic that is most natural to the problem is a three-valued logic (with a semantics due to Kleene).

(Joint work with Mooly Sagiv (Tel Aviv Univ.) and Reinhard Wilhelm (Univ. des Saarlandes).)

Heap–SSA

Martin Trapp (University of Karlsruhe)

Static Single assignment (SSA) form is a graph based representation for program code. The main advantages for program optimization is that def-use-chains are made explicit and commonalities are factored by so-called ϕ-nodes, which allow for a compact representation of dependencies. SSA-Form for programs with pointers requires considerable exact alias or points-to information. Currently, most approaches to pointer analysis use the power-set of a set of abstract objects as the domain of their data flow analysis. This modeling leads to a non-distributive data flow analysis which involves a loss of information at CFG joins which is unacceptable for our purpose. More exact analysis like the shape analysis of Reps, Sagiv, and Wilhelm avoid to merge information at join points as far as possible: A program point is annotated with a set of storage graphs, where the number of graphs may be exponential to the number of local variables.

The main point of this talk is to show, how ϕ-functions can profitably be applied to storage graph representation to achieve a more compact representation, higher accuracy and even better runtime of the analysis. Moreover, these techniques can be generalized to a framework for arbitrary non-distributive data flow problems. The key observation is that a ϕ-function delivers its result depending on some control condition. With DAGs of ϕ-functions we are able to distinguish values that reach a given program point over different control flow paths. Adding ϕ-terms to the domain of a data flow analysis allows for the encoding of control flow history into the data flow values. Thus we gain the effect of distinguished values for different control flow paths and only need a single representation for all possible alternatives at a given program point. The ϕ-functions manage to factor the common parts of the alternatives. Equivalently, we can view a ϕ-function of two data flow values as a symbolic join operations with defered evaluation. Of course this evaluation cannot be indefinitely defered, since a unbound number of different control flow paths may exist. We can at any time abstract from different control flow leading to a program point by widening, i.e. replacing the ϕ-function by the join of its arguments. ϕ-functions act like multiplexers where same control conditions mean same switching behavior. Thus they can correlate alternatives at different points reached under the same control conditions.

Based on these ideas, we define a data flow lattice for pointer analysis and give formal transfer functions for object allocation and read/write access to heap cells. We show how to make strong updates even in the presence of non-unique pointers and that we can represent information in linear space for cases where approaches with separate graphs would suffer from exponential blow up. Moreover the encoding of control flow history into data flow values with help of ϕ-functions can also be exploited to distinguish different call contexts in the interprocedural analysis. This gives the effect of inlining without code replication and is still more compact and accurate than the standard approach of distinguishing contexts by k-suffixes of call strings.

Building a Bridge Between Pointer Aliases and Program Dependences

John Ross (University of Chicago)

In this talk we present a surprisingly simple reduction of the program dependence problem to the may-alias problem. While both problems are undecidable, providing a bridge between them has great practical importance. Program dependence information is used extensively in compiler optimizations, automatic program parallelizations, code scheduling in super-scalar machines, and in software engineering tools such as code slicers. When working with languages that support pointers and references, these systems are forced to make very conservative assumptions. This leads to many superfluous program dependences and limits compiler performance and the usability of software engineering tools. Fortunately, there are many algorithms for computing conservative approximations to the may-alias problem. The reduction has the important property of always computing conservative program dependences when used with a conservative may-alias algorithm. We believe that the simplicity of the reduction and the fact that it takes linear time may make it practical for realistic applications.

Store Model Challenge the Semanticists Toolbox

Hanne Riis Nielson (Aarhus University)

Abadi and Cardelli have introduced the imperative object calculus, a untyped but statically scoped calculus for studying the creation of objects, the dynamic updating of their methods and the invocation of their methods; additionally, the calculus also contain constructs for cloning objects and for local definitions.

The operational semantics of the imperative object calculus relies on a store and we study two different organisations of the store: one that associates locations with the individual methods of the object and one that associates locations with the objects themselves. While this might seem to be a fairly innocent difference it turns out that there is quite some difference in the kind of the mathematical techniques needed for proving the correctness of even simple program analyses.

The program analysis we consider will for each expression determine which objects, abstracted by the list of their method names, it might evaluate to and hence which methods might be invoced and updated at the various points in the program. We prove the correctness of the analysis with respect to the two semantics. In the case where the store model associates locations with the methods, our proof relies on Kripke logical relations as well as coinduction.

In the case where locations are associated with the objects themselves it suffices to introduce a notion of heap signature and except for ordinary induction no special proof techniques are required.

A Pointer Data Structure-Centric Cache Model

Trishul Chilimbi (University of Wisconsin-Madison)

Joint work with James R. Larus and Mark D. Hill.

This talk presents an analytic framework for evaluating and quantifying the performance benefits of cache-conscious pointer-based structures. A key part part of this framework is a data structure-centric cache model of a series of accesses that traverse a pointer-based structure. The model has good predictive power, underestimating the actual performance improvement by not more than 15% and accurately predicting the shape of speedup curves. In addition, the model indicates that list structures are inherently more suited to caches than tree structures.

The Expressive Power of Escape Analysis

Markus Mohnen (RWTH Aachen)

The aim of escape analysis is to extract information about the storage behaviour of functional programs, needed for optimisations like compile–time garbage collection. We consider two approaches to escape analysis, both based on abstract interpretation: The analysis by Goldberg & Park and our analysis.

In this talk, we compare the relative expressive power of these analyses. For the first–order case both analyses have the same expressive power. Goldberg & Park's analysis is incomparable with ours in the higher–order case. However, we show that if we restrict the higher–order case to the functional escape behaviour then both analyses have the same expressive power even for the higher–order case. Moreover, our analysis is more precise for extensions of the language with product types and inductive types.

Compile-Time Garbage Collection for Object-Oriented Languages

Helmut Seidl (Universität Trier)

We suggest a heap analysis for JAVA-like languages that has the following advantages:

- It analyzes lifetimes of objects and allows destructive updates of objects.
- It is based on a concrete operational semantics; thus, correctness can be proved rigorously. This aspect, seems especially demanding for security critical applications.

Our analysis algorithms are taylored for a modified heap management scheme. Abstract locations of the analysis correspond to fractions of collectible objects. For each such fraction we provide a distinct slice of heap storage into which its objects reside. Our analyses allow to (partially) empty these slices again. Two strategies are supported. The first one resets a slice completely but only at program points where all objects of that fraction are definitely garbage. The second more flexible strategy maintains each slice in a stack-like fashion. It may de-allocate already that portion of a slice that contains just objects created since invocation of the current method and have found to be garbage. Not only for efficiency reasons but also for this second strategy we rely onto the concept of *local heaps*.

A full version of the talk can be retrieved from: http://www.informatik.uni-trier.de/~seidl/papers/OO.ps.gz

Cache-Conscious Sorting

Claudia Leopold (Universität Jena)

The talk gave an overview on the following two papers:

- A. LaMarca and R. Ladner: The Influence of Caches on the Performance of Sorting. in: Proceedings ACM-SIAM Symposium on Discrete Algorithms, 1997, pp. 370–379
- A. LaMarca and R. Ladner: The Influence of Caches on the Performance of Heaps. Technical Report TR-96-02-03, University of Washington, 1996

The papers describe some techniques for a cache–conscious algorithm design and give impressive experimental results indicating that cache–consciousness can have a high impact on performance. Furthermore, they introduce a technique for analyzing cache misses, called collective analysis.

Statically Type-Safe, Run-Time Dynamic Interfaces to Tabular Data Structures

Karsten Weihe (Universität Konstanz)

An attributed data type may be viewed as a data type whose items define the rows of a table (with the attributes constituting the individual columns of this table). If an algorithm shall be generic in the sense that it shall not assume a specific constellation of attributes, it must apply some sort of run-time type information, which destroys static type safety.

This talk presents a design concept for interfaces to tables, which allows one to implement statically type-safe algorithms, although the set of attributes may be given only at run time. The key idea is to enable the algorithm to integrate all type-dependent stuff into the interface.

Construction of Pointer Programs

Alex Bijlsma (Eindhoven University of Technology)

Program construction is the activity of transforming, using precise mathematical rules, specifications into efficient, correct programs. Guided by heuristics, both general and problem-specific, this transformation is intended to be performed by hand. A calculus is proposed where transformation steps based on logic or set theory are interspersed with program statements.

Pointers form a special problem because, applied naively to pointers, the general construction method would lead to unfeasibly large formulas. It is a challenge to come up with concepts and notations that are sufficiently expressive to specify the desired properties of dynamic data structures, and at the same time satisfy extremely simple proof rules. The talk contains a proposal for a family of such concepts.

The slides are available as Word97 document at http://www.win.tue.nl/inf/staf/secties/st/pm/lexb/ab64s.doc.

Calculating With Pointer Structures

Bernhard Möller (Universität Augsburg)

In calculational program design one derives implementations from specifications using semantics preserving deduction rules. The aim of modern algebraic approaches is to make both specification and calculation more concise and perspicuous by compacting logic into algebra as much as possible. The essential aims are:

- To package frequently occurring shapes of formulae, notably larger aggregations of quantifiers, into algebraic operators and to prove strong equational or inequational laws for them. Such a law usually compacts a series of inference steps in pure predicate calculus into a single one.
- To make the formulae involved more concise and less repetitious. Then writing them will be less error-prone, in particular, since copying mistakes are reduced, and reading and understanding them will be easier and quicker.
- To raise the level of discourse in formal specification and derivation close to that of informal reasoning, to achieve formality and understandability at the same time.

We present such an algebraic approach to the calculation of programs with pointer structures. It is based on the algebra of relations and partial maps. We investigate sufficient criteria for preservation of substructures under selective updating. The approach is illustrated with some simple examples such as list concatenation and reversal, tree rotation and search tree insertion and deletion. The approach covers also cyclic structures like cyclic lists or threaded trees.

Program Analysis via Graph Reachability

Thomas Reps (University of Wisconsin)

This talk describes how a wide variety of program-analysis problems can be solved by transforming them to "context-free-language reachability problems". Let L be a context-free language over alphabet A, and let G be a graph whose edges are labeled with members of A. Each path in G defines a word over A, namely, the word obtained by concatenating, in order, the labels of the edges on the path. A path in G is an L-path if its word is a member of L. Context-free-language reachability involves determining which pairs of vertices are connectd by L-paths.

Some of the program-analysis problems that are amenable to this approach include:

- Interprocedural program slicing
- Interprocedural dataflow analysis

- Flow-insensitive points-to analysis
- Analysis of dependences transmitted via manipulations of structured data (for languages that permit the use of heap-allocated storage but do not permit destructive updating of fields).

The talk is based in part on joint work with Susan Horwitz, Mooly Sagiv, Genevieve Rosay, and David Melski. Various papers concerning the material covered are available over the World Wide Web at URL http://www.cs.wisc.edu/~reps/.

5 Conclusions

There are number of interesting conclusions that we concluded from the seminar:

- Despite of the difference in background and interest the different groups interact very well.
- At the organizational level, it may be better to present research in groups to save time and make discussions more interesting. Also, talks can be shorter and not everybody needs to give a talk.
- Memory hierarchy will continue to be a challenging issue both to algorithm designers and compiler implementors.
- Cleanness checking is a very important and difficult problem.
- Many of the pointer analysis algorithms and in particular the flow insesitive ones are mature enough to be implemented in industrial compilers.

6 Participants

Uwe Assmann Universität Karlsruhe Inst. für Programmstrukturen und Datenorganisation Am Zirkel 2, Postfach 6980 D-76128 Karlsruhe, D phone: +49-721-608-6088 fax: +49-721-30047 assmann@informatik.uni-karlsruhe.de i44www.info.uni-karlsruhe.de/ ~assmann/	David Bernstein IBM - Haifa IBM Haifa Research Lab 34995 Haifa IL phone: +972-4-829-6268 fax: +972-4-829-6114 Bernstn@haifa.vnet.ibm.com	Alex Bijlsma Eindhoven University of Technology Dept.of Mathematics and Computing Science Den Dolech 2 P.O. Box 513 NL-5600 MB Eindhoven, NL phone: +31-40-247-4317 fax: +31-40-245-1733 lexb@win.tue.nl
Andreas Crauser MPI - Saarbrücken MPI für Informatik Im Stadtwald D-66123 Saarbrücken, D phone: +49-681-9325-504 fax: +49-681-9325-199 crauser@mpi-sb.mpg.de	Amer Diwan Stanford University Dept. of Computer Science CA 94305-9020 Stanford, USA phone: +1-650-723-4013 fax: +1-650-725-6949 diwan@cs.stanford.edu suif.stanford.edu/~diwan	Rudolf Fleischer MPI - Saarbrücken MPI für Informatik Im Stadtwald D-66123 Saarbrücken, D phone: +49-681-9325-119 fax: +49-681-9325-199 rudolf@mpi-sb.mpg.de

Laurie Hendren McGill University School of Computer Science Mc Connell Bldg. - Room 318 3480 University Street QC-H3A 2A7 Montreal, CDN phone: +1-514-398-7391 fax: +1-514-398-3883 hendren@cs.mcgill.ca www-acaps.cs.mcgill.ca/~hendren	Michael Hind 136 North Chestnut Street App. 16B NY 12561 New Paltz USA phone: 914-257-3567 fax: 914-257-3571 hind@camelot.mcs.newpaltz.edu	Lutz Kettner ETH Zurich Institut für Theoretische Informatik IFW B 46.2 CH-8092 Zurich CH phone: +41-1-632-7339 fax: +41-1-632-1172 kettner@inf.ethz.ch
Claudia Leopold Universität Jena Institut für Informatik Ernst-Abbe-Platz 1-4 D-07743 Jena, D phone: +49-3641-94 63 34 fax: +49-3641-94 63 03 claudia@inf.uni-jena.de www.minet.uni-jena.de/www/ fakultaet/ips/claudia.ht	Florian Martin Universität des Saarlandes FB 14 - Informatik PF 15 11 50 D-66041 Saarbrücken D	Ulrich Meyer MPI - Saarbrücken MPI für Informatik Im Stadtwald D-66123 Saarbrücken D phone: +49-681-9325-506 fax: +49-681-9325-199 umeyer@mpi-sb.mpg.de
Markus Mohnen RWTH Aachen Lehrstuhl für Informatik II D-52056 Aachen, D phone: +49-241-80-21240 fax: +49-241-8888-217 mohnen@informatik.rwth-aachen.de www-i2.informatik.rwth-aachen.de/ ~mohnen/	Bernhard Moller Universität Augsburg Institut für Informatik Universitätsstr. 14 D-86135 Augsburg D phone: +49-821-598-2164 fax: +49-821-598-2274 moeller@uni-augsburg.de	Andreas Podelski MPI für Informatik Im Stadtwald D-66123 Saarbrücken, D phone: +49-681-9325-204 fax: +49-681-9325-299 podelski@mpi-sb.mpg.de www.mpi-sb.mpg.de/guide/staff/ podelski/pode
Thomas Reps University of Wisconsin-Madison Computer Sciences Dept. 1210 W. Dayton St. WI 53706 Madison, USA phone: +1-608-262-20 91 fax: +1-608-262-97 77 reps@cs.wisc.edu	Hanne Riis Nielson Aarhus University Dept. of Computer Science Ny Munkegade DK-8000 Aarhus, DK phone: +45-89 42 32 76 fax: +45-89 42 32 55 hrn@daimi.aau.dk	Anne Rogers AT&T Labs-Research 180 Park Avenue, P.O. Box 971 NJ 07932-0971 Florham Park USA phone: +1-973-360-8668 fax: +1-973-360-8077 amr@research.att.com
John Ross Apt. 25E, 5201 S. Cornel IL 60615 Chicago, USA phone: +1-773-752-6389 fax: +1-773-702-8487 johnross@uchicago.edu www.cs.uchicago.edu/~johnross	Shai Rubin IBM - Haifa IBM Haifa Research Lab 3200 Haifa - Matam, IL phone: +972-4-829-6145 fax: +972-4-829-6115 rubin@haifa.vnet.ibm.com	Mooly Sagiv Tel Aviv University Dept. of Computer Science Ramat Aviv 69978 Tel Aviv, IL fax: +972-3-640-93 57 sagiv@math.tau.ac.il

Michael Schwartzbach BRICS, University of Aarhus Dept. of Computer Science Ny Munkegade DK-8000 Aarhus, DK phone: +45-89 42 3374 fax: +45-89 42 3255 mis@brics.dk	Michael Seel MPI für Informatik AG 1, Raum 329 Im Stadtwald D-66123 Saarbrücken, D phone: +49-681-9325-129 fax: +49-681-9325-199 seel@mpi-sb.mpy.de	Helmut Seidl Universität Trier FB IV - Mathematik/Informatik Universitätsring 15 D-54286 Trier, D phone: +49-651-201-2835 fax: +49-651-201-3822 seidl@uni-trier.de
Kurt Sieber Universität des Saarlandes FB 14 - Informatik Raum 432, Postfach 15 11 50 D-66041 Saarbrücken, D phone: +49-681-302-3235 fax: +49-681-302-2414 sieber@cs.uni-sb.de www.cs.uni-sb.de/∼sieber/	Martin Trapp Universität Karlsruhe Inst. für Programmstrukturen und Datenorganisation Am Zirkel 2, Postfach 6980 D-76128 Karlsruhe, D phone: +49-721-7400 fax: +49-721-30047 trapp@ipd.info.uni-karlsruhe.de	Trishul Chilimbi University of Wisconsin-Madison Computer Sciences Dept. 1210 W. Dayton St. WI 53706 Madison, USA phone: +1-608-262-4196 fax: +1-608-262-9777 chilimbi@cs.wisc.edu
Karsten Weihe Universität Konstanz Fak. f. Mathematik u. Informatik Fach D 188 D-78457 Konstanz, D phone: +49-7531-88-43 75 fax: +49-7531-88-35 77 karsten.weihe@uni-konstanz.de	Reinhard Wilhelm Universität des Saarlandes FB 14 - Informatik PF 15 11 50 D-66041 Saarbrücken, D phone: +49-681-302 3434 fax: +49-681-302 3065 wilhelm@cs.uni-sb.de	Mark Ziegelmann MPI für Informatik AG 1, Raum 408 Im Stadtwald D-66123 Saarbrücken, D phone: +49-681-9325-508 fax: +49-681-9325-199 mark@mpi-sb.mpg.de