Dagstuhl-Seminar

on Abstract Interpretation

Organized by:

Patrick Cousot (École Normale Supérieure, Paris)

Radhia Cousot (École Polytechnique, Palaiseau)

Alan Mycroft (Cambridge University)

28 August–1 September 1995

Overview

Abstract interpretation is a mathematical framework for specifying program analyzers which conservatively approximate program behaviours by abstraction of the language semantics. As can be seen from the abstracts of the talks, the seminar succeeded in bringing together researchers from different areas of application of abstract interpretation to focus discussion on various:

- program analysis methods;
- programming paradigms;
- program runtime properties;
- composable abstract domain designs;
- fixpoint computation methods leading to efficient implementations;
- designs of generic program analyzers and compilers using powerful analysis methods, four of them being demonstrated; and
- applications.

The numerous and lively discussions which occurred during the formal sessions, the discussion session on compositionality, and above all the informal meetings confirmed the need for such exchange between the different communities and demonstrated the benefits of this seminar.

The organizers September 1995

Acknowledgements

Thanks are due to Reinhard Wilhelm for inviting us to organize this Dagstuhl seminar on abstract interpretation; to Angelika Müller and Annette Bayer in the Dagstuhl office in Saarbrücken and the staff at Schloß Dagstuhl for ensuring that everything ran perfectly; to the participants for making the seminar lively, fruitful and of very high quality; and to Chris Colby for coordinating the production of this report.

Programme of the Workshop

Monday 28 August 1995

09h00–09h10	**R. Wilhelm**	Organizational remarks
09h10–09h45	**P. Cousot**	Introduction to Abstract Interpretation
09h45–10h30	**W.L. Harrison III**	Abstract Interpretation in a Production C/C++ Compiler
10h30–11h00	*Coffee break*	
11h00–12h00	**B. Ryder**	Flow Sensitive Data Flow Analyses
12h15–14h00	*Lunch*	
14h00–14h45	**F. Martin**	Program Analyzer Generator
14h45–15h15	**P. Cousot**	Compositional Design of Galois Connections
15h15–16h00	**C. Fecht**	A Tool for Generating Efficient Prolog Analyzers
16h00–16h30	*Coffee break*	
16h30–17h30	**G. Filé**	Cut, Complement and Paste with Abstract Domains
17h30–18h00	**P. Cousot**	Fixpoint Approximation with Widening/Narrowing
18h00–19h00	*Dinner*	
19h00–20h00	*U. Loebens*	*Opening of U. Loebens Art Exhibition*

Tuesday 29 August 1995

09h00–09h45	**G. Levi**	Abstract Debugging
09h45–10h45	**M. Codish**	Bottom-Up Abstract Interpretation of Logic Programs—from Theory to Practice
10h45–11h45	*Coffee break*	
11h45–12h00	**M. Hanus**	Abstract Interpretation of Functional Logic Languages
12h00–12h15	**M. Sagiv**	A Few Words on Interprocedural Analysis
12h15–14h00	*Lunch*	
14h00–14h30	**C. Hankin**	Analysis of Value-Passing Process Calculi
14h30–15h30	**F. Nielson**	Inference Based Analysis
15h30–16h15	**J. Palsberg**	A Type System Equivalent to Flow Analysis
16h15–16h45	*Coffee break*	
16h45–17h45	**M. Bruynooghe**	A Framework for Abstract Interpretation of Logic Programs Based on Preinterpretations
18h00–19h00	*Dinner*	
20h00–21h00	**F. Martin**	Demo: Program Analyzer Generator
	C. Fecht	Demo: Prolog Analyzer Generator

Wednesday 30 August 1995

09h00–09h45	**N. Jones**	Optimizing Run-Time Data and Operations
09h45–10h30	**F. Bourdoncle**	Implicit Higher-Order Polymorphism with Primitive Typing
10h30–11h00	*Coffee break*	
11h00–11h45	**D. Boulanger**	Decoding Models of Logic Programs
11h45–12h15	**P. Granger**	Local Iterations
12h15–14h30	*Lunch*	
14h30–18h00		*Visit to Trier/Hiking/Cycling*
18h00–19h00	*Dinner*	
20h00–21h00	**F. Bourdoncle**	Demo: SYNTOX: A Pascal Abstract Interpreter

Thursday 31 August 1995

09h00–09h45	**H. Søndergaard**	Dependency Analysis for Logic Programs
09h45–10h30	**R. Shyamasundar**	Termination Analysis of Logic Programs
10h30–11h00	*Coffee break*	
11h00–11h45	**M. Rosendahl**	On Fixpoint Iteration in Abstract Interpretation
11h45–12h15	**T. Jensen**	Flow Analysis in the Geometry of Interaction
12h15–14h30	*Lunch*	
14h30–15h15	**M. Sagiv**	Graph-based Pointer Analysis via Abstract Interpretation
15h15–16h00	**A. Deutsch**	Semantic Models of Dynamic Storage and their Abstractions
16h00–16h30	*Coffee break*	
16h30–17h15	**K. Marriott**	Practical Experience in Using Abstract Interpretation in a Highly Optimizing Compiler for CLP(R)
17h15–18h00	**H. Seidl**	Set Constraints to Stop Deforestation
18h00–19h00	*Dinner*	
19h00–19h30	**W.L. Harrison III**	Demo: C/C++ Abstract Interpretor
19h30–21h00	All participants	Discussion

Friday 1 September 1995

09h00–09h30	**U. Assmann**	Abstract Interpretation and Data Flow Analysis with Graph Rewrite Systems
09h30–10h00	**A. Poetzsch-Heffter**	Interprocedural Data Flow Analysis Based on Temporal Specifications
10h00–10h30	**O. Rüthing**	The Complexity of Exhaustive Motion-Elimination Frameworks
10h30–11h00	*Coffee break*	
11h00–11h30	**J. Knoop**	Analyzing and Optimizing Parallel Programs
11h30–12h00	**C. Colby**	Trace-Based Program Analysis
12h15–14h30	*Lunch*	

Abstracts of the talks

Abstract Interpretation and Data Flow Analysis with Graph Rewrite Systems

Uwe Assmann

In the talk we defined a special class of graph rewrite systems for program analysis: *edge addition rewrite systems (EARS)*. EARS can be applied to distributive data-flow frameworks over finite power domains, as well as many other program analysis problems, which are normally not related to data-flow analysis.

We also presented some techniques for optimized evaluation of EARS. The main technique is a generic algorithm which evaluates non-recursive EARS in $O(n^k e^{lp})$, where n the maximal number of nodes in a node domain with an arbitrary label, e the maximum out-degree of a node concerning an arbitrary edge label, l the length of the longest path of a path cover over all left hand sides, p the maximum number of paths in a path cover, and k is the *order* of an EARS (the number of source node labels in left hand sides). With this algorithms and some other evaluation techniques in practice often linear and quadratic algorithms result. This shows that EARS are very well suited for generating efficient program analyzers.

[1] Uwe Assmann. On Edge Addition Rewrite Systems and Their Relevance to Program Analysis. In J. Cuny, editor, *5th Workshop on Graph Grammars and Their Application To Computer Science*, to appear in Lecture Notes in Computer Science. Springer, 1995.

[2] Uwe Assmann. *Generierung von Programmoptimierungen mit Graphersetzungssystemen*. PhD thesis, Universitt Karlsruhe, Kaiserstr. 12, 76128 Karlsruhe, Germany, July 1995.

Decoding Models of Definite Logic Programs

Dmitri Boulanger

Model theory is the basis of an elegant approach for static analysis. It is based on the standard logical notions of interpretation, model and logical consequence. The directness of this method contrasts with the need, in most abstract interpretation frameworks, to define abstract and concrete domains with Galois connection, abstract and concrete semantics, and prove their safety.

A program has a model with respect to any pre-interpretation of a language. Properties of a program are encoded by its model. A procedure, which derives declarative properties of a program only using its model, is called a decoding of a model. Once a pre-interpretation is chosen, the *complete* decoding of a model or a superset of a model is implied. It exploits a direct connection between

the pre-interpretation and derivable properties. This enables an automation of construction of analysis algorithms for definite logic programs.

Implicit Higher-Order Polymorphism with Primitive Typing
Francois Bourdoncle

We present a generalization of the Hindley-Milner type system handling finite and partially ordered sets of base types and type constructors. We show how this type system can be used to design programming languages retaining much of the ML spirit while integrating in a seamless fashion higher-order and object-oriented programming, dynamic dispatch on several arguments, parametric polymorphism, and overloading. The semantics of our type system is not based on extensible records with methods attached to them. Rather, we use an intensional framework with two categories of objects: data objects, implemented as tagged tuples, and functional objects, with two possible implementations: lambda-abstractions and sets of functional objects, called methods, dispatching on the type of their arguments in pretty much the same way as ML functions perform pattern matching. The idea of the system is to introduce universally quantified constrained types like: $\forall \alpha : \alpha \leq \mathit{real}.\ \alpha \to \alpha$, whose intuitive meaning is a type transformer with domain "*real*" mapping any run-time type to itself, that is, the identity function over reals. Finally, we propose extensions such as overloading and recursive types.

Framework for Abstract Interpretation of Logic Programs Based on Preinterpretations
Maurice Bruynooghe

For every preinterpretation, we define two domains for abstract substitutions and their corresponding Galois connection and establish a correct abstraction of unification. The first abstraction uses domain relations (relations over the domain of the preinterpretation) and was known before for some preinterpretations, though not formalised as is done here. The second abstraction uses sets of domain relations. Surprisingly, this domain can express properties such as definite freeness which hold for sets of terms but not for all instances of those terms. Whereas conventional abstractions are developed in an ad-hoc way, our approach is more systematic and based on changing the logic of equality through the selection of a preinterpretation. Until now it was unclear whether and how such an approach allows the derivation of properties that hold for sets of terms but not necessarily for all instances of those terms. We report also on a preliminary comparison with conventional approaches.

Analyzing Logic Programs for Real: A Persistent Type Analysis
Michael Codish

This paper describes a practical approach in which simple and efficient analyses for logic programs can be obtained. The main contribution is the application of this approach for polymorphic type analysis. Analyses are obtained by abstracting programs replacing concrete terms by abstract terms and evaluating the minimal models of the corresponding abstract programs. For type analysis, abstract terms are type expressions constructed from type descriptors and a single binary function symbol $\oplus$ which is associative, commutative and idempotent. The unification of type expressions is based on an extension of ACI-unification.

Efficiency is maintained by designing the analysis so that the type of a program is in many cases independent of its context and hence persistent to changes in the underlying type domain. Persistence is obtained by considering a non-ground semantics for abstract programs much the same as in the case of non-ground semantics for logic programs. The advantages for program analysis are comparable to the concrete case: computational efficiency, domain independence, and the persistence of analyses to changes in the underlying alphabet of the type language.

Trace-Based Program Analysis
Christopher Colby

We present *trace-based program analysis*, a semantics-based framework for statically analyzing and transforming programs with loops, assignments, and nested record structures. Trace-based analyses are based on *transfer transition systems*, which define the small-step operational semantics of programming languages. Intuitively, transfer transition systems provide direct support for reasoning about the possible execution traces of a program, instead of just individual program states. The traces in a transfer transition system have many uses, including the finite representation of all possible terminating executions of a loop. Also, traces may be systematically "pieced together", thus allowing the composition of separately analyzed program fragments. The utility of the approach is demonstrated by showing three applications: software pipelining, loop-invariant removal, and data alias detection. We conclude by commenting on how trace-based analysis can provide a general framework for the analysis of concurrent programs.

Abstract Interpretation: Foundations and New Trends in Application
Patrick Cousot

Introduction to Abstract Interpretation

We recall the objectives of abstract interpretation, its main uses in designing proof methods, semantics and analysis methods for a wide variety of program properties and programming languages. We discuss a few applications to pro-

gram testing and manipulation. The methodology for approximating a semantics by abstract interpretation is shortly presented and illustrated in detail by grammar analysis.

Compositional Design of Galois Connections

We discuss the compositional design of fixpoint inducing/approximation using Galois connections/surjections defined by induction on the mathematical structure of the semantic domains. The lattice of abstractions allows for partial comparison of approximations. A few examples are given both in numerical and symbolic domains.

Fixpoint Approximation with Widening/Narrowing

We discuss the use of widening/narrowing operations for space reduction and convergence acceleration in iterative fixpoint approximation. We show that some precise analyses would be impossible to obtain with an abstract domain without infinite chains whence justifying the necessary use of widenings.

GENA — A Tool for Generating Efficient Prolog Analyzers
Christian Fecht

GENA is a tool that allows to generate Prolog analyzers from specifications. An analyzer is specified by an abstract domain which defines a lattice of abstract substitutions and functions for performing abstract unification, abstract procedure entry, and abstract procedure exit. GENA takes a set of abstract domains as input and generates an executable analyzer which contains all the specified abstract domains. The actual analyzer is obtained by plugging an abstract domain into an analysis engine. An analysis engine is an implementation of a particular generic abstract interpreter for Prolog. GENA contains analysis engines for goal dependent and goal independent analyses. Furthermore, normalization is supported. We implemented several different fixpoint algorithms: three working list algorithms and four recursive top-down interpreters. Abstract domains are specified in SML. GENA comes with a large set of data structures which are useful for the analysis of logic programs, such as sets of variables, sets of sets of variables, and binary decision diagrams. So far, we have specified the abstract domains POS for groundness analysis and the domains JL and SUND for sharing analysis. Our implementation of POS is about four times faster than the implementation by Van Hentenryck using the GAIA system.

Cut, Complement and Paste of Abstract Domains
Gilberto Filè

The aim of our work is to define a set of operations on abstract domains that allow to costruct complex domains from simple ones or viceversa that allow to decompose a complex domain into simpler components. In this seminar I describe two such operations: the quotient and the complement and I illustrate their usefulness for decomposing the well-known abstract domain *Sharing*.

In general an abstract domain D expresses several different information. The quotient operation computes the part of an abstract domain that is useful to compute a single particular information. We have shown that the quotient of $Sharing$ with respect to the groundness property is the abstract domain of definite propositional formulas Def. From this result it's easy to prove that the domain Pos of positive formulas is strictly better than $Sharing$ for computing groundness.

Once we know that Def abstracts $Sharing$, it's natural to wonder about what remains of $Sharing$ when Def is "taken out of it". This intuition is formalized by the operation of complementation: we want to know what is the complement of Def wrt $Sharing$ ($Sharing - Def$). We show that such a complement exists in most cases of interest. Moreover we give a characterization of $Sharing - Def$ as a simple upper closure on $Sharing$ that we call $Sh+$. Furthermore we show that also $Sh+$ can be divided into two more abstract components: one expressing the pair-sharing information and the other expressing the set-information.

Joint work with A. Cortesi (Venezia), R. Giacobazzi (Pisa), K. Palamidessi (Genova), F. Ranzato (Padova), and W. Winsborough (PSU, USA).

Improving the Results of Static Analyses of Programs by Local Decreasing Iterations
Philippe Granger

We present a new technique conceived for improving the accuracy of the results of static analyses of programs. It relies on and complements the classical lattice-theoretic model for abstract interpretation, and is therefore defined at a very general level. The idea consists in embedding local decreasing iterations in the global iteration stage of the analysis; by local iterations, we mean that they correspond to some special control points of the program and not to the whole program to be analyzed. Special techniques dedicated to static analyses on numbers are developed to take full advantage of the method, which allows, in practice, to handle more efficiently conditional branches, assignments in backwards analysis, combinations of static analysis frameworks, and to combine abstract interpretation and symbolic evaluation. Our technique may significantly improve the analysis results.

Static Analysis of Value-Passing Process Calculi
Chris Hankin

There are a number of applications in safety critical systems and consumer electronics where simple concurrent languages are used. Often, there is the need for static analysis which tracks the use of values in such programs. Standard approaches to the semantics of value-passing process calculi involve a translation into a pure calculus (only involving pure synchronisation). This expansion

involves a loss of information which is unacceptable for many of the analyses of interest. We present a model of value passing process calculi based on synchronisation trees which gives a direct treatment of values. The construction is general but is illustrated in the context of value-passing CCS. We illustrate the use of the model as the basis for an abstract interpretation which discovers dependency information. The abstract interpretation uses the domain **Def**, which was developed in the logic programming community.

Joint work with David Clark and Lindsay Errington.

Abstract Interpretation of Functional Logic Programs: Problems and Partial Solutions
Michael Hanus

Functional logic languages amalgamate functional and logic programming paradigms. They can be efficiently implemented by extending techniques known from logic programming. We show how global information about the call modes of functions can be used to optimize the compilation of functional logic programs. Since mode information has been successfully used to improve the implementation of pure logic programs and these techniques can be applied to implementations of functional logic programs as well, we concentrate on optimizations which are unique to the operational semantics of functional logic programs. In the first part we consider normalizing innermost narrowing as the operational semantics. Normalizing innermost narrowing combines the deterministic reduction principle of functional languages with the nondeterministic search principle of logic languages. We define a suitable notion of modes for functional logic programs based on this operational semantics and present compile-time techniques to optimize the normalization process during the execution of functional logic programs. Furthermore, we present a framework to derive such global mode information. Due to the normalization process between narrowing steps, standard analysis frameworks for logic programming cannot be applied. Therefore we develop new techniques to correctly approximate the effect of the intermediate normalization process.

In the second part we consider an operational semantics derived from the lazy evaluation principle of functional languages. We show that strictness information is not sufficient to transform lazy narrowing into eager narrowing due to the presence of logical variables (in contrast to pure functional languages). Since additional groundness information is necessary, we discuss possible techniques to derive such information w.r.t. lazy narrowing strategies.

[1] M. Hanus. Towards the global optimization of functional logic programs. In *Proc. 5th International Conference on Compiler Construction*, pages 68–82. Springer LNCS 786, 1994.

[2] M. Hanus and F. Zartmann. Mode analysis of functional logic programs. In *Proc. 1st International Static Analysis Symposium*, pages 26–42. Springer LNCS 864, 1994.

Abstract Interpretation in a Production C/C++ Compiler
Luddy Harrison

I describe the design of a commercial C/C++ compiler that is intended to produce highly optimized machine code for computer systems that rely on concurrency to achieve their peak performance (superscalar, VLIW, SIMD, *etc.*) The compiler operates upon a low-level intermediate form (called Q) which is essentially structured assembly language. In this setting, pointer casting and pointer arithmetic have a natural semantics in terms of machine types. The optimizations performed by the compiler are driven by a dependence graph. The edges of the graph are associated with direction vectors that describe the control-flow movements between the source and sink of the dependence. In restructuring Fortran compilers such direction vectors connect points in an iteration space that is defined by a loop nest. Here, the direction vectors are procedure strings, and connect points in a global iteration space in which there is a dimension for every important control point in a program (subroutine, call site, *etc.*) I illustrate several dependence tests in this setting, including a dependence test over subscripted (array) data, and a test over linked data structures.

Flow Analysis in the Geometry of Interaction
Thomas Jensen

This talk presents an approach to flow analysis of programs with higher-order functions under normal-order reduction. The framework is based on an abstract machine derived from the Geometry of Interaction semantics for cut elimination in Linear Logic proof nets. The transition system defined by the machine induces a set of equations defining the flow between the program points, represented by vertices in the proof net. This set of equations defines a collecting semantics for the program and is amenable to further analysis by standard methods from abstract interpretation. As examples of its application we show how to derive analyses in the framework: strictness analysis, aimed at optimising call-by-name to call-by-need, closure analysis, aimed at deciding which closures are passed around as arguments in a higher-order functional program, and usage analysis for approximating the number of times a particular argument to a function is used.

Removing Redundant Data and Code over a Range of Programming Languages
Neil D. Jones

Meta-interpreters provide a general, convenient way to implement, say, a user-

oriented language using a well-established and engineered implementation language. The price, unfortunately, is often poor efficiency.

Partial evaluation can help, by removing much "interpretation overhead"—indeed, it can remove *all* overhead for simple untyped languages (Lisp, λ-calculus) so one can "have one's cake" (convenient user-oriented language) "and eat it too" (run it efficiently).

Alas, this happy situation does *not* hold for typed languages; the culprit is the necessary use of a universal value domain, often exploding the size of object-language data as represented in the interpreter, and thereby running slower than necessary.

The talk (work in progress) outlined a way to remove this overhead by post-processing the partial evaluator's output on a user-oriented program. This consists of an abstract interpretation, to trace the flow from "producers" of constructors, type tags, pointers, *etc.* to their "consumers" that decompose/use them. The safety of the analysis ensures detection of (many) redundant constructors, *etc.*, and is used to optimize the program by removing much code both to construct and deconstruct values.

Analyzing and Optimizing Parallel Programs without Additional Costs
Jens Knoop

For a sequential program the classical answer of how to get a fast running object program is to compile it by means of an *optimizing* compiler. More recently, a second answer becomes popular and more and more important: To apply an *automatic parallelizer* and to run the generated program on a parallel machine. However, all too often both alternatives are considered to exclude each other because naive adaptations of the sequential optimization methods fail, and their straightforward correct adaptations have unacceptable costs caused by considering all interleavings that manifest the possible executions of a parallel program.

The point of this talk is to show that for the large class of *bitvector* problems, which are most relevant in practice, there is an elegant way out of this dilemma. In fact, we show how to construct for parallel programs with *shared memory* and *interleaving semantics* optimal bitvector analyses which are as efficient as their sequential counterparts, and which can easily be implemented. This is very important as there is a broad variety of powerful classical optimizations like *code motion*, *strength reduction*, *partial dead code elimination*, and *assignment motion* which only require bitvector analyses. All these optimizations are now available for parallel programs almost without any additional costs on the runtime and the implementation side.

Joint work with Bernhard Steffen (University of Passau, Germany) and Jürgen Vollmer (University of Karlsruhe, Germany).

On the Abstract Diagnosis of Logic Programs
Giorgio Levi

Abstract diagnosis of logic programs is an extension of declarative diagnosis, where we deal with specifications of operational properties, which can be characterized as abstractions of *SLD*-trees (observables).

We introduce a simple and efficient method to detect incompleteness errors, which is based on the application of the immediate consequences operator to the specification. The method is proved to be correct and complete whenever the immediate consequences operator has a unique fixpoint. We prove that this property is always satisfied if the program belongs to a large class of programs (acceptable programs). We then show that the same property can be proved for any program P, if the observable belongs to a suitable class of observables. We finally consider the problem of diagnosis of incompleteness for a weaker class of observables, which are typical of program analysis.

Practical Experience with using Abstract Interpretation in a Highly Optimizing Compiler for CLP(R)
Kim Marriott

Constraint Logic Programming (CLP) languages extend logic programming by allowing constraints from different domains such as real numbers or Boolean functions. The considerable expressive power and flexibility gained by combining constraint programming with logic programming is not without cost. Implementations of constraint logic programming (CLP) languages must include expensive constraint solving algorithms tailored to specific domains, such as trees, Booleans, or real numbers. The performance of many current CLP compilers and interpreters does not encourage the widespread use of CLP.

I will describe a highly optimizing compiler for CLP(R), a CLP language which extends Prolog by allowing linear arithmetic constraints. The compiler uses sophisticated global analyses to determine the applicability of different program transformations. The analyzer is based on a generic abstract interpretation engine and six different analysis domains which give information about definite and possible interaction between constraints. Preliminary evaluation of the optimizing compiler and the analyzer is very promising.

Joint work with Andrew Kelly, Andrew Macdonald, Harald Søndergaard, Peter Stuckey, and Roland Yap.

Generation of Efficient Interprocedural Analyzers with PAG
Florian Martin

To produce high quality code, modern compilers use global optimization algorithms based on *abstract interpretation*. These algorithms are rather complex; their implementation is therefore a non–trivial task and error–prone. However,

since they are based on a common theory, they have large similar parts. We conclude that analyzer writing better should be replaced with analyzer generation.

We present the tool PAG that has a high level functional input language to specify data flow analyses. It offers the specification of even recursive data structures and is therefore not limited to bit vector problems. PAG generates efficient analyzers which can be easily integrated in existing compilers. The analyzers are interprocedural, they can handle recursive procedures with local variables and higher order functions. PAG has successfully been tested by generating several analyzers (e.g. alias analysis, conditional constant propagation) for an industrial quality ANSI-C and Fortran90 compiler.

Research in compiler generation has concentrated mostly on front end and lately on back end generation. The optimization phase has not received much attention. Only a few systems [4, 5, 6] for generation of analyzers have been designed and built. All of them apply ad-hoc methods or heuristics if the language has subroutines. We present a new generative system for interprocedural analyses, PAG, that is able to produce analyzers which can be applied in several different compilers by instantiation of a well designed interface. The system is based on the theory of abstract interpretation. The philosophy of PAG is to support the designer of an analyzer by providing three languages for specifying the data flow problem, the abstract domains, and the compiler interface. This simplifies the construction process of the analyzers as well as the correctness proof and it results in a modular structure. The specifier is neither confronted with the implementation details of domain functionality nor with the traversal of the control flow graph or syntax tree nor with the implementation of suitable fixpoint algorithms.

Joint work with Martin Alt.

[1] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *International Conference on Static Analysis (SAS'95)*, Lecture Notes in Computer Science. Springer, 1995. to appear.

[2] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[3] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[4] S. W. K. Tjiang and J. L. Hennessy. Sharlit – A tool for building optimizers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 82–93, San Francisco, CA USA, [7] 1992. ACM

Press , New York, NY , USA. Published as SIGPLAN Notices, volume 27, number 7.

[5] G. Venkatesch and C. N. Fischer. Spare: A development environment For Program Analysis Algorithms. In *IEEE Transactions on Software Engineering*, volume 18, 1992.

[6] K. Yi and W. L. Harrison III. Automatic generation and management of interprocedural program analyses. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246–259, Charleston, South Carolina, Jan. 1993.

Inference Based Analysis
Flemming Nielson

The focus in this talk is the extraction of properties for (functional) programming languages with concurrency and polymorphism; we make this concrete by a study of Concurrent ML. The first stage of the analysis is a type and effect inference that extracts the type of a given expression as well as the communication behaviour (a kind of process algebra expression) its evaluation may give rise to. The second stage of the analysis then analyses the behaviours using techniques from abstract interpretation in order to provide results useful for static and dynamic processor allocation. Both stages incorporate correctness results and algorithms in order to obtain solutions. The talk concludes by identifying the need to integrate polymorphism and polyvariance into the same analysis framework.

Joint work with Hanne Riis Nielson.

A Type System Equivalent to Flow Analysis
Jens Palsberg

Flow-based safety analysis of higher-order languages has been studied by Shivers, and Palsberg and Schwartzbach. Open until now is the problem of finding a type system that accepts exactly the same programs as safety analysis. We prove that Amadio and Cardelli's type system with subtyping and recursive types accepts the same programs as a certain safety analysis. The proof involves mappings from types to flow information and back. As a result, we obtain an inference algorithm for the type system, thereby solving an open problem.

Interprocedural Data Flow Analysis Based on Temporal Specifications
Arnd Poetzsch-Heffter

This paper investigates the specification of data flow problems by temporal logic formulas and proves fixpoint analyses correct. Temporal formulas are in-

terpreted w.r.t. programming language semantics given in the framework of evolving algbras. This enables very high-level specifications, in particular for history sensitive problems. E.g. the classical bit vector analyses can be refined by using information about branch conditions without having to change their specifications. The general semantics framework makes the approach directly applicable to realistic programming languages.

We use the specifications to prove fixpoint analyses of data flow analyses correct. As an example, we develop a powerful interprocedural deadness analysis that uses constant information depending on the context where the active procedure was called. By proving such a combination of backward and forward analyses correct, we illustrate the use of specifications in correctness proofs.

Higher-Order Fixpoint Iteration
Mads Rosendahl

We present a method for doing demand-driven fixpoint iteration on domains with higher-order functions. The technique is based on using partial function graphs to represent higher-order object. The main problem in finding fixpoints for higher-order functions is to establish a notion of *neededness* so as to restrict the iteration to those parts of the function that may influence the result. This is here done through a uniform extension of the domain of values with need information.

Fixpoint equations of this kind may arise from strictness analysis of higher-order functions using the BHA-technique To analyse the strictness one has to evaluate the strictness functions for a small set of arguments: one argument tuple for each parameter to the function. The set of all possible arguments to the strictness function may, however, be many orders of magnitude bigger. The technique has been used in a strictness analyser for Haskell.

The use of fixpoint iteration as a programming paradigm is discussed through examples from language theory and it is argued that it may be used as a generalisation of recursion.

The Complexity of Exhaustive Motion-Elimination Frameworks
Oliver Rüthing

An important class of optimization techniques that mutually take advantage of each other are methods employing code motion in order to increase the potential of elimination transformations.

Common to such *motion-elimination frameworks* is the presence of *second order effects*. Taking these second order effects completely into account requires that both types of transformations are applied *exhaustively*. However, unlike to the components involved, which are usually based on *bit-vector data flow analyses* and whose complexity is well-understood, there is no serious estimation on the number of iteration steps that is necessary in order to stabilize the whole

process.

Measurements indicate that practically relevant motion-elimination frameworks do stabilize considerably fast. In this talk the theoretical evidence for this observation is given. For some important problems we present linear bounds that are even independent from the branching structure of a program. Finally, we provide a new argument for the importance of splitting critical edges, since code motion based on bidirectional data flow analysis is shown to impose extra penalty costs.

Defining Flow Sensitivity in Data Flow Problems
Barbara G. Ryder

Since Banning first introduced flow sensitivity in 1978, the term has been used to indicate hard or complex data flow problems, but there is no consensus as to its precise meaning. We look at Banning's original uses of the term and some interpretations they have generated. Then we consider the multiplicity of meanings in more recent interprocedural analyses, categorizing a number of data flow problems. We also classify several recent interprocedural approximation techniques with respect to properties related to sensitivity and discuss additional data flow problem properties. Finally, we propose a definition for flow sensitivity that attempts to capture the effects of program representation granularity and function space properties.

Joint work with Thomas J. Marlowe and Michael G. Burke.

Solving Shape-Analysis Problems in Languages with Destructive Updating
Mooly Sagiv

This paper concerns the static analysis of programs that perform destructive updating on heap-allocated storage. We give an algorithm that conservatively solves this problem by using finite shape-graphs to approximate the possible "shapes" that heap-allocated structures in a program can take on. In contrast with previous work, our method is even accurate for certain programs that update cyclic data structures.

For certain programs – including ones in which a significant amount of destructive updating takes place – our technique is able to determine such properties as (i) when the input to the program is a list, the output is (still) a list; (ii) when the input to the program is a tree, the output is (still) a tree; and (iii) when the input to the program is a possibly circular list, the output is a possibly circular list. For example, our method can determine that "list-ness" is preserved by (i) a program that performs list reversal via destructive updating of the input list, and (ii) a program that searches a list and splices a new element into the list. Furthermore, our method can determine that "circular list-ness" is preserved by the program that searches a list and splices in a new element.

None of the existing methods that use graphs to model the program's store are capable of determining that "list-ness" is preserved on these examples (or examples of similar complexity). As far as we know, no other existing alias-analysis/shape-analysis method (whether based on graphs or other principles) has the ability to determine that "circular list-ness" is preserved by the list-insert program.

Joint work with Thomas Reps and Reinhard Wilhelm.

Constraints to Stop Deforestation
Helmut Seidl

Deforestation is a transformation of functional programs to remove intermediate data structures. It is based on outermost unfolding of function calls where folding occurs when unfolding takes place within the same nested function call. Since unrestricted unfolding may encounter arbitrarily many terms, M.H. Sørensen in 1994 proposed a grammar based analysis to determine those terms which should better not be transformed. We recast his analysis by means of set constraints and show how it can be made more informative by adding further constraint systems—essentially at no loss in efficiency. The constraint systems we add are

- Boolean constraints to restrict the approximation to terms possibly encountered by the outermost unfolding strategy;
- integer constraints to additionally compute depths or sizes of arguments and/or reduction contexts.

Termination Analysis of Logic Programs
R. K. Shyamasundar

A methodology for proving the termination of well-moded logic programs is developed by reducing the termination problem of logic programs to that of term rewriting systems. A transformation procedure is presented to derive a term rewriting system from a given well-moded logic program such that the termination of the derived rewrite system implies the termination of the logic program for all well moded queries under a class of selection rules. This facilitates applicability of a vast source of termination orderings proposed in the literature on term rewriting, for proving termination of logic programs. The termination of various benchmark programs has been established using this approach. Unlike other mechanizable approaches, the proposed approach does not require any preprocessing and works well even in the presence of mutual recursion. The transformation has also been implemented as a front-end to Rewrite Rule Laboratory (RRL) and has been used in establishing termination of nontrivial Prolog programs such as a prototype compiler for ProCos PL_0 language.

The transformational approach is extended for proving termination of parallel logic programs such as GHC programs; it exploits the fact that unifications in GHC-resolution correspond to matchings. The termination of a GHC program for a class of queries is implied by the termination of the resulting rewrite system. The approach facilitates the applicability of a wide range of termination techniques developed for rewrite systems in proving termination of GHC programs. The method consists of three steps: (a) deriving moding information from a given GHC program, (b) transforming the GHC program into a term rewriting system using the moding information and finally (c) proving termination of the resulting rewrite system. Using this method, the termination of many benchmark GHC programs such as quick-sort, merge-sort, merge, split, fair-split and append, *etc.*, can be proved.

Positive Logic for Dependence Analysis
Harald Søndergaard

Many static analyses for declarative programming/database languages use Boolean functions to express dependencies among variables or argument positions. Examples include groundness analysis for (constraint) logic programs and finiteness analysis and functional dependency analysis for databases. We identify four classes of Boolean functions that are commonly used for program analysis. Two of them, Pos and Def, capture dependencies by admitting implication. We give semantic and syntactic characterisations of the classes and investigate their algebraic properties. We show how the properties of positive logic translate into a groundness analysis which is not only highly precise but also practical as witnessed by several independent implementations.