Dagstuhl Seminar on Program Analysis

organised by
Hanne Riis Nielson, Aarhus University
Mooly Sagiv, Tel Aviv University

12.-16. April 1999

Motivation for the seminar. 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. Traditionally, program analysis has been used in optimising compilers; more recent applications include the validation of safety properties of software for embedded systems, applications in reverse engineering and program understanding, and also program analysis shows promise of becoming an important ingredient in ensuring the acceptable behaviour of software components roaming around on information networks.

Over the years a number of different approaches have been developed for program analysis, all of which have a quite extensive literature. To give an impression of the diversity let us briefly mention some of the main approaches. The *flow based approach* includes the traditional data flow analysis techniques for mainly imperative languages, but also control flow analysis as developed for functional and object oriented languages, and set based analysis as developed for logic and functional languages. The *model based approach* includes the parameterised denotational semantics techniques developed for functional and imperative languages, but also more generally the use of mathematical modelling in the abstract interpretation of imperative, functional, concurrent and logic languages. The *inference based approach* includes general logical techniques touching upon program verification techniques, but also the type and effect systems developed for functional, imperative and concurrent languages.

Typically, the various techniques have been developed and further refined by subcommunities – with the result that often the commonalities between analogous developments are not sufficiently appreciated. To guide the research effort in our community, it is necessary with an appraisal of the current technology. Therefore one of the primary aims of the Dagstuhl Seminar was to bring researchers from the various subcommunities together to share their experience.

The programme of the seminar. The scientific programme of the seminar consisted on six invited talks (60 minutes each) and a number of contributed talks (30 minutes each); in the evenings extensive discussions were taking place in five working groups with the goal of identifying the most challenging research problems for the next few years. The

working groups were within the areas:

- Program modularity and scaling of program analysis;
- New applications of program analysis;
- Security through program analysis;
- Foundations of program analysis;
- Combining static and dynamic analyses.

The findings of the working groups were subsequently presented and discussed in plenum. The abstracts of the talks and the position statements produced by the working groups can be found in this report.

A novel application area. At this seminar we begin to see a novel application area for program analysis: *the analysis of security problems*. As described in the position statement from the working group on "Security through program analysis" further interaction with the security community is required in order to understand the possibilities as well as limitations of our techniques; however, classical program analysis techniques have already been successfully applied to a few security problems:

- Ensuring type and memory safety and thereby detecting leaks of secure information caused by accessing memory not allocated by the application itself – this is important for protection against certain viruses.
- Detection of information flow using program dependency analysis allows one to validate certain security and integrity constraints by imposing a distinction between trusted and untrusted data, between secure and public data etc.
- Validation of security protocols by extraction of the actual protocol used by an application and subsequent analyses of its properties.

Novel results presented at this Dagstuhl seminar indicate that program analysis techniques may be superior to many of the other techniques suggested for analysing security properties: Based on a program analysis it was shown how to construct a "hardest attacker" that then can be analysed in conjunction with the program of interest so as to ensure that certain security properties are fulfilled.

Also the seminar showed quite some interests in applications of program analysis to legacy code. Within the more traditional application areas of program analysis in particular the combination of static and dynamic analysis received attention.

Unifying the community. A previous Dagstuhl Seminar on Abstract Interpretation[1] showed that there were serious gaps between the subcommunities and that there was a need for an exchange of ideas. This view was confirmed in a number of position statements

[1]Patrick Cousot, Radhia Cousot and Alan Mycroft: Dagstuhl Seminar on Abstract Interpretation, September 1995, Dagstuhl Report No.123, 1995.

written for the celebration of the 50th anniversary of the ACM[2]. We are very pleased to observe that today the program analysis community is moving towards a greater appraisal of competing techniques. In addition to the informal discussions between the subcommunities, a number of talks emphasised the relationship between different approaches and so did the discussions of the working group on "Foundations of Program Analysis". Also it was noteworthy that techniques developed in one area are being taken up in other areas – an example is the further development of techniques used for analysing pointer structures to analysing mobile computation.

Acknowledgements. We would like to thank the Scientific Director for inviting us to organise this seminar. Thanks to the staff at the Dagstuhl Office in Saarbrücken and to the staff at Schloss Dagstuhl for making it a very pleasant stay for all of us. Last but not least we would like to thank the participants of the seminar for many interesting talks and exciting discussions; a special thank goes to Flemming Nielson, Jon Riecke, Barbara Ryder, Adam Webber and Reinhard Wilhelm for putting the position statements of the five working groups together and to René Rydhof Hansen for collecting the abstracts of the talks for this report.

Aarhus and Tel Aviv, April 1999

Hanne Riis Nielson
Mooly Sagiv

[2]Chris Hankin, Hanne Riis Nielson and Jens Palsberg (editors): Strategic directions in computer science: models of programming languages. ACM Computing Surveys vol. 28 no. 4, 1996.

1 Abstracts of the talks

1.1 Monday morning

Reinhard Wilhelm: Grammar Flow Analysis

Grammars have been used both as subjects of analyses, i.e., in *Grammar Flow Analysis* [4, 3, 5, 1, 9], and as devices into which other analysis problems have been coded. So called *Grammar Based Analyses* have been used in [2, 6, 7, 9].

Many problems known in the area of Compiler Generation could be regarded as Grammar Flow Problems, e.g., Productivity and Reachability of nonterminals, First and Follow, Lower and Upper attribute dependences. One from each pair is a *bottom-up Grammar Flow Analysis Problem*, the other a *top-down Grammar Flow Analysis Problem*. Typical for all these pairs is, that the solution of a bottom-up problem is part of the specification of a top-down problem.

Rephrasing these problems in one common framework, in particular in the study of attribute dependences, allowed to order and relate a whole variety of known methods and even generate some new ones [8].

Coding an interprocedural Data Flow Analysis Problem into a grammar allows a natural characterization of *interprocedurally feasible paths*. Associating the corresponding transfer functions with the productions of the grammar then completes the encoding. Computing a fixpoint by bottom-up iteration over the grammar solves the problem.

[1] M. Jourdan and D. Parigot. Techniques to improve grammar flow analysis. In N. Jones, editor, *Proc. of the European Symposium on Programming'90*, pages 240 – 255. LNCS 432, Springer Verlag, 1990.

[2] D.E. Knuth. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6(1):1 – 5, 1977.

[3] U. Möncke. *Generierung von Systemen zur Transformation attributierter Operatorbäume, – Komponenten des Systems und Mechanismen der Generierung.* PhD thesis, Universität des Saarlandes, 1985.

[4] U. Möncke and R. Wilhelm. Iterative algorithms on grammar graphs. In H. Goettler, editor, *8th Workshop on Graphtheoretic Concepts in Computer Science*, pages 177–194. Hanser-Verlag, 1982.

[5] U. Möncke and R. Wilhelm. Grammar flow analysis. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, pages 151–186. LNCS 545, Springer Verlag, 1991.

[6] G. Ramalingam. *Bounded Incremental Computation.* PhD thesis, University of Wisconsin, Madison, 1993.

[7] J. Uhl and R.N. Horspool. Flow grammars: A flow analysis methodology. In P.A. Fritzson, editor, *Proc. of the Fifth Int. Conf. on Comp. Construct.*, pages 203–217. LNCS 786, Springer Verlag, 1994.

[8] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995. German edition: *Übersetzerbau – Theorie, Konstruktion, Generierung*. Springer-Verlag, 1992. French edition: *Les Compilateurs. Théorie. Construction. Génération.* Masson, 1994.

[9] W. Yang. A lattice framework for analyzing context-free languages with applications in parser simplification and data-flow analysis. *Journal of Information Science and Engineering*, 1997.

Laurie Hendren: Tools and Analysis for Optimizing Java Bytecode

The Java programming language presents many interesting opportunities for optimizations. The Sable research group is building frameworks and compiler analyses for facilitating the analysis and optimization of Java bytecode.

This talk motivated the need for a three-address code intermediate representation, and gave some performance figures showing what is important to optimize in Java.

Then an overview of the projects currently underway was given, including a brief discussion of the Soot(Jimple) framework which provides a typed three-address intermediate form for Java bytecode. Based on this overview, two kinds of analysis were presented in more detail. The first is an intraprocedural type inference that is used inside the Soot framework to assign a type to each parameter and local variable. The second is a family of analyses that are used to resolve virtual method calls. In both cases experimental results on benchmarks, including the specJVM benchmarks and bytecode generated from other languages (Ada, ML, Eiffel, Pizza) were given.

This is joint work with Raja Vallee-Rai, Etienne Gagnon, Vijay Sundaresan, Patrick Lam and Chrislain Razafimahefa.

Evelyn Duesterwald: Transparent Dynamic Optimization: The Dynamo Project

Dynamic optimization refers to the runtime optimization of a native program binary. This talk discusses the design and implementation of Dynamo, a prototype dynamic optimizer, that is the product of three years of research conducted HP Labs Cambridge. Dynamo improves program performance by extracting hot program traces during execution. The traces are optimized and then placed into an optimized code cache for future execution. Dynamo does not rely on annotations in the original program binary: it can operate on legacy binaries without recompilation or instrumentation. Dynamo may be implemented entirely in software and does not depend on special compiler, operating system, or hardware support. The compact code footprint of Dynamo allows is to comfortably fit into ROM, and its data memory footprint is configurable for a variety of performance/memory-footprint tradeoffs. This makes Dynamo ideal for use in a wide range of platforms from large servers to embedded devices.

1.2 Monday afternoon

Greg Nelson: Extended static checking

The talk described a system for detecting at compile time certain errors that are normally not detected until run time, and sometimes not even then. For example, array bounds errors, NIL dereferences, and race conditions and deadlocks in multi-threaded programs. The system has been implemented both for Modula-3 and for Java. The system requires the programmer to annotate procedure and method declarations with simple preconditions and postconditions. These annotations are much less onerous than the annotations that would be required for a full program correctness proof. Loop invariants are not required. The checking is totally automatic. The checker reports errors by line number. The system handles essentially all features of the Java 1.0 language, including concurrency and object references with inheritance. About thirty thousand lines of code have been checked with the system, and a number of errors have been found.

The talk included a demonstration of the Java version of the system.

1.3 Tuesday morning

Flemming Nielson: Flow Logics for Calculi of Computation

Flow Logic is an approach to the static analysis of programs so as to predict statically (and potentially fully automatically) safe approximations to the dynamic behaviour of programs or processes. It is mainly based on a popular approach to Control Flow Analysis but allows to integrate insights from Abstract Interpretation and Data Flow Analysis. In several recent developments it has been demonstrated that Flow Logic is a robust approach that is able to deal with a variety of calculi of computation: the lambda-calculus, Concurrent ML, the Imperative Object Calculus, the pi-calculus, the Mobile Ambients calculus, and the spi-calculus. In this talk we will give an overview of the basic approach of Flow Logic and some of its applications. This will focus on the Mobile Ambients calculus and showing that a proposed firewall is protective in the sense that it cannot be entered by attackers not knowing the necessary passwords.

This is joint work with Hanne Riis Nielson, René Rydhof Hansen, Jacob Grydholt Jensen, Pierpaolo Degano and Chiara Bodei. Information about the flow logic approach to program analysis is available at `http://www.daimi.au.dk/~fn/FlowLogic.html`.

Andrei Sabelfeld: Probabilistic Secure Information Flow

You have received a program from an untrusted source. Let us call it company M. M promises to help you to optimise your personal financial investments, information about which you have stored in a database on your home computer. The software is free (for a limited time), under the condition that you permit a log-file containing a summary of your usage of the program to be automatically emailed back to the developers of the program (who claim they wish to determine the most commonly used features of their tool). Is such a program safe to use? The program must be allowed access to your personal investment

information, and is allowed to send information, via the log-file, back to M. But how can you be sure that M is not obtaining your sensitive private financial information by cunningly encoding it in the contents of the innocent-looking log-file? This is an example of the problem of determining that the program has secure information flow. Information about your sensitive "high-security" data should not be able to propagate to the "low-security" output (the log-file). Traditional methods of access control are of limited use here since the program has legitimate access to the database.

This talk proposes an extensional semantics-based formal specification of secure information-flow properties in sequential programs based on representing degrees of security by partial equivalence relations. The specification clarifies and unifies a number of specific correctness arguments in the literature, and connections to other forms of program analysis. The approach is inspired by (and equivalent to) the use of partial equivalence relations in specifying binding-time analysis, and is thus able to specify security properties of higher order functions and "partially confidential data" (e.g. one's financial database could be deemed to be partially confidential if the number of entries is not deemed to be confidential even though the entries themselves are). We show how the approach can be extended to handle nondeterminism, and illustrate how the various choices of powerdomain semantics affect the kinds of security properties that can be expressed, ranging from termination-insensitive properties (corresponding to the use of the Hoare (partial correctness) powerdomain) to probabilistic security properties, obtained when one uses a probabilistic powerdomain.

This is joint work with Dave Sands.

Hanne Riis Nielson: Shape Analysis for Mobile Ambients

Mobile Ambients is a calculus of computation (developed by Cardelli and Gordon) that allows active processes to move between sites. Since processes may evolve when moving around it is far from trivial to predict which processes may turn up inside given sites. As an example consider the system

client1 | client2 | ! server

that may evolve into

server[client1] | server[client2] | ! server

showing that each of the clients may move into a copy of the server. However, how can we ensure that the two clients never enter the same copy of the server? The aim of this work is to develop a static analysis that can be used to validate such properties.

The domains of the analysis are sets of shape grammars. A shape grammar is a modification of a context free grammar: the symbols on the right hand side of a production are unordered, each of them can only appear once and each of them has a counter associated. The counter indicates whether the symbol is an abstraction of a single subambient, a finite (>1) set of subambients or an infinite set of subambients. The symbols of the grammar are constructed from the ambient names and capabilities of the mobile ambient

of interest; however, to enhance the precision of the analysis the symbols will contain additional information about the subambients as computed by a deterministic bottom-up tree automaton.

An extraction function will associate a shape grammar with each mobile ambient and a closure operation will then construct a set of shape grammars reflecting how the ambient may evolve during computation. The closure operation will first use materialisation to identify a redex and this may give rise to a set of shape grammars. For each of these it will mimic the reduction of the redex by replacing some of the productions with others and finally a normalisation step takes place to get a shape grammar according to the above definition; also this step may give rise to a set of shape grammars.

This is joint work with Flemming Nielson.

Thomas Jensen: Verification of control flow based security properties

A fundamental problem in software-based security is whether *local* security checks inserted into the code are sufficient to implement a *global* security property. We introduce a formalism based on a two-level linear-time temporal logic for specifying global security properties pertaining to the control-flow of the program, and illustrate its expressive power with a number of existing properties. We define a minimalistic, security-dedicated program model that only contains procedure call and run-time security checks and propose an automatic method for verifying that an implementation using local security checks satisfies a global security property. For a given formula in the temporal logic we prove that there exists a bound on the size of the states that have to be considered in order to assure the validity of the formula: this reduces the problem to finite-state model checking. Finally, we instantiate the framework to the security architecture proposed for Java (JDK 1.2).

This is joint work with D. Le Métayer and T. Thorn.

1.4 Tuesday afternoon

Thomas Reps: Parametric Shape Analysis via 3-Valued Logic

We present a family of abstract-interpretation algorithms that are capable of determining "shape invariants" of programs that perform destructive updating on dynamically allocated storage. The main idea is to represent the stores that can possibly arise during execution using three-valued logical structures.

Questions about properties of stores can be answered by evaluating predicate-logic formulae using Kleene's semantics of three-valued logic:

- If a formula evaluates to *true*, then the formula holds in every store represented by the three-valued structure.
- If a formula evaluates to *false*, then the formula does not hold in any store represented by the three-valued structure.

- If a formula evaluates to *unknown*, then we do not know if this formula always holds, never holds, or sometimes holds and sometimes does not hold in the stores represented by the three-valued structure.

Three-valued logical structures are thus a conservative representation of memory stores.

This paper presents a *parametric* framework for shape analysis: It provides the basis for generating different shape-analysis algorithms by varying the predicates used in the three-valued logic. The analysis algorithms generated handle every program, but may produce conservative results due to the class of predicates employed; that is, different three-valued logics may be needed, depending on the kinds of linked data structures used in the program and on the link-rearrangement operations performed by the program statements.

The paper is available at `http://www.cs.wisc.edu/wpis/papers/popl99.ps`

This is joint work with: Mooly Sagiv (Tel-Aviv Univ.) and Reinhard Wilhelm (Univ. des Saarlandes).

Mooly Sagiv: A Decidable Logic for Describing Linked Data Structures

This paper aims to provide a better formalism for describing properties of linked data structures (e.g., lists, trees, graphs), as well as the intermediate states that arise when such structures are destructively updated. The paper defines a new logic that is suitable for these purposes (called Lr, for "logic of reachability expressions"). We show that Lr is decidable, and explain how Lr relates to two previously defined structure-description formalisms ("path matrices" and "static shape graphs") by showing how an arbitrary shape descriptor from each of these formalisms can be translated into an Lr formula.

The paper is available at `www.math.tau.ac.il/~sagiv/esop99.ps`

This is joint work with: Michael Benedikt, Thomas Reps.

Barbara G. Ryder: Practical Program Analysis for Object-oriented Statically Typed Languages

Relevant context inference (RCI) is a schema for interprocedural flow- and context-sensitive data-flow analysis of statically typed object-oriented programming languages such as C++ and Java. RCI can be used to analyze complete programs and incomplete programs such as libraries. This approach is designed to minimize the memory needs of the analysis; it does not require that the entire program be in memory throughout the analysis. RCI will be explained in the context of points-to analysis of a realistic subset of C++. Initial empirical results measured on the problems of virtual function resolution and statement-level MOD (side effects) are encouraging; on average, a 10-fold reduction in memory usage for analysis is observed.

This is joint work with Ramkrishna Chatterjee.

Manuel Fähndrich: Scalable Implementation of Simple Set Constraints

Simple Set Constraints are useful to express a variety of analyses, including closure analysis and points-to analysis. Resolution of such set constraints involves a closure operation on the constraint graph derived from the program under analysis. This closure consists of finding pairs of sources and sinks connected via a path in the constraint graph. When such pairs are found, new edges are added into the constraint graph, corresponding to constraints on the sub-expressions of sources and sinks.

Standard implementations of this closure are based on forward propagation of sources. Unfortunately, this approach combines the detection of source-sink pairs with the computation of the set of reaching sources for all vertices in the constraint graph. As a result, the space requirements for large constraint problems become impractical.

We present an alternative approach, based on a combination of forward propagation of sources and backward propagation of sinks. The new approach focuses on detecting paths between sources and sinks. Sets of reaching sources are computed on demand after the closure is completed. Together with techniques to eliminate cycles arising in the constraint graph and merging of similar sink vertices, this new approach yields significant speedups—one to two orders of magnitude—over the standard approach for an implementation of a flow and context insensitive points-to analysis.

Satish Chandra: Some software engineering applications of program analysis

The effectiveness of traditional type checking in C is limited by the presence of type conversions using type casts. Because the C standard allows arbitrary type conversions between pointer types, neither C compilers, nor tools such as *lint*, can guarantee type safety in the presence of such type conversions. In particular, by using casts involving pointers to structures (C `struct`'s), a programmer can interpret any memory region to be of any desired type, further compromising C's weak type system. Not only do type casts make a program vulnerable to type errors, they hinder program comprehension and maintenance by creating latent dependencies between seemingly independent pieces of code.

To address these problems, we have developed a stronger form of type checking for C programs, called *physical type checking*. Physical type checking takes into account the layout of C `struct` fields in memory. This paper describes an inference-based physical type checking algorithm and its implementation. Our algorithm can be used to perform static safety checks, as well as compute useful information for software engineering applications.

1.5 Wednesday morning

Jon G. Riecke: Region analysis and the polymorphic lambda calculus

The region calculus of Tofte and Talpin is a typed lambda calculus that can statically delimit the lifetimes of dynamically created objects. It uses a notion of "regions" that appear in types, and a construct for delimiting the scope of regions. Tofte and Talpin

show, using a detailed operational semantics with memory, that regions can be safely deallocated at the end of the scope. The proof of the property is complicated by subtle facts about the calculus, e.g., that certain regions can be deallocated safely even though objects in them may be reachable.

In order to understand the calculus better, we give a translation of the region calculus into an extension of the polymorphic lambda calculus called F#. We give a denotational semantics of F#, and use it to give a simple and abstract proof of correctness of memory deallocation. We have used the translation to simplify the typing rules of the region calculus, and believe the techniques extend to the design of other region-based calculi.

This is joint work with Anindya Banerjee (Stevens Institute of Technology) and Nevin Heintze (Bell Laboratories, Lucent Technologies).

Dave Schmidt: A Graphical Presentation of Control-Flow Analysis

Control-flow analysis (CFA) is used to calculate, for languages with dynamic method invocation, which methods might be invoked at an invocation point. The analysis, originally designed by Shivers and Sestoft, has been extensively studied, and in particular, Nielson and Nielson have designed a generic framework from which one can generate the k-CFA and Uniform-k-CFA hierarchies.

In this presentation, we review the crucial aspects of Shivers's formulation of CFA and the key components of the k-CFA and Uk-CFA hierarchies. Then, building on Mossin's formulation of 0-CFA as a program flow chart that is completed by a transitive closure rule, we show how to generate similar control-flow charts for k-CFA and Uk-CFA analyses. The control-flow charts are analysed for constant propagation and escape analysis. Finally, we suggest how to apply model-checking techniques to the flow charts for computing simple storage management analyses.

Jörgen Gustavsson: A Type Based Sharing Analysis for Update Avoidance and Optimisation

Sharing of evaluation is crucial for the efficiency of lazy functional languages, but unfortunately the machinery to implement it carries an inherent overhead. In abstract machines this overhead shows up as the cost of performing updates, many of them actually unnecessary, and also in the cost of the associated bookkeeping, that is keeping track of when and where to update. The aim of this work is to reduce the overhead of lazy evaluation by trying to reduce the number of performed updates and also by reducing the cost of the associated bookkeeping.

In this talk I will explain how spineless abstract machines, such as the STG-machine and the TIM, keeps track of updates through pushing, checking for and popping update markers. I will also present some initial results which indicates that with our analysis many of the updates and virtually all update marker checks can be avoided.

The paper is available at `http://www.cs.chalmers.se/~gustavss` and in the proceedings of ICFP'98.

Chris Hankin: Games and Program Analysis

We present Game Semantics as an alternative basis for semantics-based program analysis. We present the basic concepts of game semantics; games are associated with types and programs are modelled by strategies. We introduce a simple abstraction of strategies that leads to a cubic algorithm for Control Flow Analysis (0-CFA). We use this example to motivate the use of a category of nondeterministic games as abstract "domain". We have used this approach to analyse programs in the simply typed functional language PCF and its imperative extension Idealised Algol. In addition to Control Flow Analysis, we have used this approach to define a pre-order on program points that represents a higher-order generalisation of flowcharts. We use the flowcharts as a basis for data flow and information flow analysis. The main advantages that we claim for the approach are that it is modular (language extensions only require local changes) and intensional analyses are "calculated" in a fairly direct way.

This is joint work with Pasquale Malacaria (Imperial College, London).

Jean-Pierre Talpin: Higher-order and typed synchronous programming

Synchronous languages are well suited for the design of dependable real-time systems: they enable a very high-level specification and an extremely modular implementation of complex systems by structurally decomposing them into elementary synchronous processes. To maximize the support for the generic implementation and the dynamic reconfigurability of reactive systems, we introduce a semantics of higher-order processes in a seamless extension of the synchronous language SIGNAL. To enable the correct specification and the modular implementation of systems, we introduce an expressive type inference system for characterizing their temporal and causal invariants.

1.6 Thursday morning

Patrick Cousot: Abstract Interpretation, Modal Logic and Data Flow Analysis

The objective of the talk is to derive a dataflow analysis by abstract interpretation of its temporal logic specification.

The link between abstract interpretation and dataflow analysis was first exhibited by P.& R. Cousot (POPL'79) by showing that the merge over all paths and fixpoint solutions used in dataflow analysis are all abstract interpretations of (at the time finite prefix closed) sets of traces. So the equations to be solved can be derived from the program semantics (the considered example was that of available expressions).

In another direction, Bernhard Steffen established a link between data-flow analysis as model-checking (TACS'91, LNCS 536) and studied the generation of data-flow analysis algorithms from model specifications (SCP 21:115-139, 1993): the program is abstracted into a model which is checked with a modal μ-calculus formula. Abstract interpretation is used only at the flowchart node level to specify the correctness of the dataflow transfer functions.

Dave Schmidt builds upon these previous works by introducing the point of view that "an iterative data-flow analysis is a model-check of a modal logic formula on a program's abstract interpretation" (POPL'98). The idea is that the model used by Bernhard Steffen can be obtained by an abstract interpretation of a trace-based operational semantics and that the model check of the modal logic formula on this model "yield the same information as" the solution to the data-flow equations. We are not fully satisfied by this point of view because it gives the impression to have two different ways (model checking and dataflow analysis) to do the same thing. In particular the presence of a bug in live variables analysis shows that the methodology does not guarantee soundness. This is because the dataflow equations are not derived by abstraction of their specification. This is a general problem with model-checking where the abstraction process is (in general) not (fully) taken into account.

We show that model-checking is an abstract interpretation and then instantiate to the modal logic formula specifying the data-flow property. Hence the data-flow equations are derived by calculus by abstraction of their specification, a point which is left informal in the previous works and will broaden our POPL'79 point of view (where the abstraction was specific to each considered example).

We first introduce a new temporal logic RTL, with a reversal operator making past and future completely symmetric. The semantics of temporal formulae is given in compositional fixpoint form. The semantics of programs is a trace-based operational semantics generated by a total transition system. We next introduce universal and existential (may be state and location partitioned) abstractions and their duals, checking a temporal formula for an operational semantics. We show how to derive the boolean model-checking equations by fixpoint abstraction (for short only the existential abstraction of forward state properties is considered).

This is finally applied to live-variables. Liveness is specified along one path. The existential abstraction classical is used to merge over some path. The classical dataflow equations are derived for this existential abstraction. Hence dead variables are correct for the universal abstraction as observed by D. Schmidt. Thus, abstract interpretation provides an unambiguous understanding of the abstraction process and the dataflow equations are correct by construction.

In conclusion, we stress that abstract interpretation is a theory of discrete approximation of semantics, not only a peculiar static program analysis method. In this talk, we have seen that it covers both model-checking and dataflow analysis.

This is joint work with Radhia Cousot.

Jens Knoop: Program Optimization under the Perspective of Reuse

Reuse of successful approaches in new environments is an inexpensively means for mastering the constant increase of complexity of software systems. This also applies to the specific field of optimizing compilation. In this talk we investigate the benefits and limitations of reuse in this field. Choosing soundness and completeness of analyses and transformations together with the existence of generic tools for their automatic generation as the basic criteria, we consider the transferability of analyses and transformations

together with the extensibility of the respective generic tools to advanced settings and paradigms under the preservation of soundness and completeness as surveyor's rod for measuring the success of reuse.

While on the analysis level the extensibility of abstract-interpretation based analysis frameworks to advanced settings and paradigms (interprocedural, explicitly parallel, data-parallel, object-oriented, ...) make reuse of analyses reasonably successful with respect to this surveyor's rod, on the transformation level additional care is usually required. We show that the rationale underlying the soundness and completeness of performance improving transformations, which on this level means semantics preservation and optimality in a specific well-defined sense, is usually quite sensitive to paradigm changes. Using partial redundancy elimination for illustration, we demonstrate that neither completeness nor soundness are generally preserved. However, success stories reported for properly adapted techniques, whose straightforwardly reused counterparts fail, demonstrate that reuse generally pays back on the transformation level, too.

This is joint work with Oliver Rüthing, University of Dortmund.

John Field: Equations as a Uniform Framework for Operational Semantics, Partial Evaluation, and Abstract Interpretation

A variety of disparate methods have traditionally been used to define the operational semantics of languages, to describe partial evaluation, to formalize program analysis as abstract interpretation, and to implement each of these operations in practical systems. In this talk, I argue that *equational logic* can serve to unify each of these aspects of language manipulation. The benefits of such a unified view are considerable:

- Specification of interpreters, partial evaluators, or analyzers can often be done in a *modular* fashion, simply by adding or removing sets of equations.
- Equational semantics is very well understood.
- Equational systems are particularly amenable to efficient mechanical implementation.

I illustrate the relation between equational reasoning, operational semantics, and partial evaluation using a logic for imperative programs called PIM. I then sketch preliminary work on a new approach for systematically developing abstract interpretation from concrete equational specifications. These ideas are illustrated using a "classical" program analysis problem. While this problem would traditionally require three staged analyses, our approach combines these stages in a single fixpoint computation.

Parts of this work are joint with J. Heering, T.B. Dinesh, and J. Bergstra.

Jakob Rehof: Very Large Scale Type Inference with Polymorphic Recursion

We describe experiments with a type inference system for C with polymorphic recursion. The system is based on solving instantiation (semi-unification) constraints, leading to a

completely modular type analysis of a system. We show how this technology together with the AST-Toolkit developed at MSR makes it possible to seamlessly plug in the inferencer into an existing build environment. We then describe preliminary performance results from experiments with the inferencer, demonstrating that this technology scales up to systems of the order of several millions of lines of real C-code.

1.7 Thursday afternoon

Fritz Henglein: Fighting the Millennium Bug: Software Reengineering Using Type-based Program Analysis

We employ type-based modelling, inference and transformation as a basis for remediation of COBOL applications with Year 2000 problems. The basic software reengineering method consists of: identifying concrete representations of intended abstract data types ('calendar year'); ensuring that only permissible operations with type-checked interfaces operate on them; and, after type checking, replacing the inadequate representations, along with their operations, to Year 2000 safe ones.

Type System 2000 (TS2K) is a type system for annotating the storage areas of COBOL programs with year type information. In COBOL it is necessary to type physical storage areas ("physical typing") since COBOL has aliasing and allows arbitrary record block moves from one storage area to another. TS2K makes sure that all storage areas that may contain years during program execution are carefully tracked. Type errors indicate potential Year 2000 problems. These are flagged to the user, together with a number of plausible corrective actions. A type checked program is converted automatically after specifying new data types for year-carrying data.

Type checking in TS2K can be reduced to associative unification, though restricted to size-bounded types, since the sizes of types are given by the underlying COBOL data types. We show that associative unification has most general unifiers in this case and present an algorithm for computing them efficiently.

The above method has been incorporated in AnnoDomini, a commercially available tool (see `www.cgsinc.com` or `www.hafnium.com`) for Year 2000 remediation of OS/VS COBOL/ CICS applications. AnnoDomini's core inference engine is programmed in Standard ML; its editor, LPEX, is provided by IBM; and its graphical user interface, which controls both the inference engine and the editor, is developed in Visual Basic.

This is joint work with Peter Harry Eidorff, Christian Mossin, Henning Niss, Morten Heine Sørensen and Mads Tofte of DIKU and Hafnium ApS.

Arie van Deursen: Understanding COBOL Systems using Inferred Types

Types are a good starting point for various software reengineering tasks. Unfortunately, programs requiring reengineering most desperately are written in languages without an adequate type system (such as COBOL). To solve this problem, we propose a method of automated type inference for these languages (A. van Deursen and L. Moonen. Type Inference for COBOL Systems. In Proc. of the fifth Working Conference on Reverse

Engineering, pages 220–230. IEEE Computer Society, 1998). The main ingredients are that if variables are compared using some relational operator their types must be the same; likewise if an expression is assigned to a variable, the type of the expression must be a subtype of that of the variable. We use subtyping to prevent pollution: the phenomenon that types become too large, and contain variables that intuitively should not belong to the same type. We provide empirical evidence for this hypothesis(A. van Deursen and L. Moonen. Understanding COBOL Systems using Inferred Types. In Proc. of the seventh International Workshop on Program Comprehension. IEEE Computer Society Press, May 1999).

The analysis consists of two phases: first, a single pass over the sources of COBOL system to derive relational facts (type constraints) based on variable usage in that system. This is followed by a phase in which these facts are manipulated using relational algebra.

The main results include a formal type system and inference rules for this approach, a scalable tool set to carry out modular type inference experiments on COBOL systems, a suite of metrics characterizing type inference outcomes. Applications include: documentation generation, constant depropagation, regression testing, object identification and language migration.

This is joint work with L. Moonen (CWI).

1.8 Friday morning

Adam Webber: Analysis of class invariant binary relations

The value of information about class invariants in object-oriented programs is very high: they can be used to justify many instances of standard compiler optimizations. The cost of such information is also usually very high: so high that software tools have at most attempted to verify user-supplied invariants, but have not attempted to identify them automatically. I introduce a technique for doing this. The technique applies a relational constraint to the individual methods of a class, extracting tables of canonical binary relations expressed using an efficient fixed vocabulary. The results of these individual analyses are then combined to identify a table of class-invariant relations: a set of relations on the fields of the class that are provably true on entry to a certain subset of the methods (the "clean-called" methods). The analysis has been implemented for a tool called Bounder that identifies unnecessary array bounds checks in Java bytecode.

Sabine Glesner: Natural Semantics for Imperative and Object-Oriented Programming Languages

We present a declarative specification method based on natural semantics which is suitable for the definition of the static and dynamic semantics of imperative and object-oriented programming languages. We show how the semantic analysis can be generated automatically by creating, for each program under consideration, a constraint problem whose solution is also a valid attribution. In contrast to previous implementations of natural semantics, our generation algorithm for the semantic analysis is more flexible because

it allows us to compute a program's attributes independently from its syntax tree. For special cases, we have defined efficient solution strategies. We also describe our prototype implementation using the concurrent constraint programming language Oz.

Florian Martin: Analysis of Loops

Programs spend most of their time in loops and procedures. Therefore, most program transformations and the necessary static analyses deal with these. It has been long recognized, that different execution contexts for procedures may induce different execution properties. There are well established techniques for interprocedural analysis like the call string approach. Loops have not received similar attention. All executions are treated in the same way, although typically the first and later executions may exhibit very different properties.

In this talk a new technique is presented that allows the application of the well known and established interprocedural theory to loops. An extension to the call string approach is presented, that gives greater flexibility to group several calling contexts together for the analysis.

The traditional and the new techniques are implemented in the Program Analyzer Generator PAG, to demonstrate the applicability of the results.

David Melski: Interprocedural Path Profiling

In path profiling, a program is instrumented with code that counts the number of times particular path fragments of the program are executed. This paper extends the intraprocedural path-profiling technique of Ball and Larus to collect information about interprocedural paths (i.e., paths that may cross procedure boundaries).

Interprocedural path profiling is complicated by the need to account for a procedure's calling context. To handle this complication, we generalize the "path-naming" scheme of the Ball-Larus instrumentation algorithm. In the Ball-Larus work, each edge is labelled with a number, and the "name" of a path is the sum of the numbers on the edges of the path. Our instrumentation technique uses an edge-labelling scheme that is in much the same spirit, but to handle the calling-context problem, edges are labelled with functions instead of values. In effect, the edge-functions allow edges to be numbered differently depending on the calling context. A key step in the process of creating the proper edge functions is related to a method proposed by Sharir and Pnueli for solving context-sensitive interprocedural dataflow-analysis problems.

This is joint work with Thomas Reps.

Allyn Dimock: The Church ML Compiler: Goals, Technology, Status

The "pollution problem" or "W problem" is the problem of having the representation of one piece of data determine the representations of other pieces of data. To avoid the W problem, it is common to use a single convention to represent each kind of data.

The Church ML compiler seeks to solve the W problem, so as to be able to use efficient representations wherever possible.

Our method is to use a combination of intersection types, union types, and flow types in a typed intermediate language. The flow information on the types gives an interprocedural equivalent of the use-def / def-use web which other compilers encode into SSA and auxiliary data structures. Intersection and union types are used not only to represent polymorphism in the source language, but also to separate flow paths by desired representation. A transformation that turns some intersection types into product types and some union types into sum types is then able to "break the W" and allow multiple representations at the cost of some duplication of code.

We further discuss some of the ways that we can test the extent of the W problem. We discuss the issue of shallow subtyping on flows, and the tension between using flow information for program analysis vs flow information for program transformation.

We conclude with a brief comparison of our technology and that of the MLton compiler, and with a summary of project status and upcoming additions to the compiler.

The Church ML compiler is joint work with Glenn Holloway, Elena Machkasova, Robert Muller, Santiago Pericas, Franklyn Turbak, Joe Wells, and Ian Westmacott. The typed intermediate language is joint work with Robert Muller, Franklyn Turbak, and Joe Wells.

2 Position statements from the working groups

2.1 Program modularity and scaling of program analysis

An important problem in program analysis is the problem of methods that do not scale well with program size. The group discussed this problem from many angles and produced the following observations and directions for future work.

- **Algorithms**. In program analysis, as in all other areas of computer science, there are relatively few known lower bounds on time and space complexity. More research on algorithmic analysis, to find improved algorithms, reductions, and (if possible) tight lower bounds on common analysis problems, is necessary. We observe that space complexity is often especially critical, with many algorithms requiring, for example, large amounts of information to be stored for every program point. For some applications quadratic time may be acceptable, but quadratic space generally is not acceptable.

- **Test Cases**. There is a need for large test cases—targets for analysis that are similar to the largest commercial and government applications. Such test cases serve many purposes: they help in testing during development, they provide experimental evidence of scaling properties, and they can be used to compare implementations. We avoided using the word "benchmark" to refer to these test cases, as we recognize the potential for abuse. But the danger of placing too much reliance on benchmarks seems very distant, since there are at present no standard test cases of the necessary size. Unfortunately, these programs tend to be mostly proprietary or classified. What can we do about this?

- **Large Heterogeneous Systems**. One new challenge for analysis is large *heterogeneous* systems: systems with parts written in different languages, running on different architectures, communicating over different interfaces, and so on.

- **Modularity**. In some cases, we can design algorithms that scale well if the code being analyzed is modular. This means, at least, that the program being analyzed is presented in small pieces with well-designed, limited interfaces. It may also mean that the interfaces are to some degree declared: effects, assumptions, invariants, exceptions, and so on. Such declared, checkable interfaces will not be adopted by programmers if they serve no other purpose than to make analysis easier. But they may be a win all around: good software engineering practice *and* an aid to scalability of analysis.

- **Interfaces**. What kinds of interface languages are helpful for analysis? The interface can augment a program with information not derivable from the text—for example, assumptions about external inputs. It can also function as a cache for information that could be reproduced by analysis. When is it a good idea to cache information there, and when is it a good idea to regenerate the information by analysis? What kinds of interface languages are helpful; how expressive and how complex should they be? (A sufficiently complex and expressive interface language

might itself require analysis!) There is a need for useful program logics or type systems or annotation systems designed to represent interface information.

- **Mechanically Generated Code**. Programs generated by automatic tools need not be modular. There are many compilers that compile to C. Can such tools preserve modularity?

- **Legacy Code**. A common source of non-modular analysis problems is legacy code. There is a need for tools to help programmers modularize legacy code retroactively. The group does not believe that this is a task that can be fully automated, but does believe that better tools would help.

Group members: Allyn Dimock, John H. Field, Fritz Henglein, David Melski, Greg Nelson, Steffen Priebe, Jakob Rehof, Thomas Reps, Mooly Sagiv, Adam B. Webber (editor of the position statement)

2.2 New applications

Coping with modern Hardware

- **Highly irregular architectures.** Roughly 90% of the processor market lies in embedded applications. Most of the recent digital signal processors (DSPs) have very irregular architectures, for which code generation is extremely difficult. Extensive program analysis may make this task easier.

- **Caches, Pipelines, out-of-order-execution.** Architecture components and features such as caches, pipelines, and out-of-order execution may make traditional "optimizations" obsolete if they are not taken into consideration. Therefore, more cache aware optimizations have to be developed. In addition these components present problems when determining precise worst case execution times as needed in real-time programs.

- **Power consumption.** Optimizing programs for minimal power consumption is needed in many embedded application in particular in medicine.

- **Shared memory.** How to compile correctly to shared memory machines is still an unsolved problem.

- **Speculative optimizations.** Aggressive optimizations may be needed in some cases, even though the enabling conditions could not be found to be satisfied. The compiler, however, has to prepare for the case that these enabling conditions indeed are violated.

Coping with modern Software

- **Preparing for dynamic optimization and cheap dynamic analysis.** Many applications need dynamic optimizations as not the whole program is available to

the compiler. However, the compiler could prepare the necessary dynamic analyses by extensive static analyses of the available parts.

- **Analyzing web-based applications.** Ubiquitous web based applications written in a mixture of languages e.g., XML, HTML, dynamic HTML, CGI, and interfaced through middleware, e.g., CORBA, produce worst legacy problems than nowadays Cobol programs. There is no approach visible to analyze those.

Coping with old Software

- **Reverse engineering and reengineering.** This is an ongoing activity which will benefit from unconventional analyses.

Analyzing application-specific code

- **Data base programs, transactional systems.** What can program analyses do when they "know about the application"?

Applied Techniques Several techniques are used across these different application areas.

- Testing, profiling, debugging,
- Analyzing binaries,
- Statically controlled dynamic memory management.

Group members: John Field, Sabine Glesner, Leon Moonen, Arie van Deursen, Thomas Reps, Reinhard Wilhelm (editor of the position statement), and others.

2.3 Security through program analysis

A classification of security. Security is an important area where there is reason to believe that techniques from program analysis may prove very helpful. This includes applications to the following ingredients of security:

- *Confidentiality* (or *secrecy*). This amounts to making sure that certain sensitive data is not becoming generally known; to model systems, several levels of confidentiality may be appropriate. The properties are mainly safety properties in the sense that certain invariants must continue to hold; we ignore here the possibility that keys used in encryption can be compromised using brute-force computational attacks.
- *Integrity.* This is based around the notion of trust, meaning the use of data or code that is deemed to not lead to any breaches of security; again this is largely a safety issue. *Authentication* is a branch of integrity aiming at preventing a document to be

tampered with; a useful technique is digital signatures (which in RSA is much the same as encryption). *Watermarking* is a branch of integrity aiming at preventing the origins of a document to be obfuscated even though attempts are made at masquerading it.

- *Availability.* This deals with ensuring that certain services are always possible (e.g. for creating fresh keys that have never been used before). This is mainly a liveness issue.
- *Auditing.* This does not enforce security per se, but is a technique for ensuring accountability: that breaches of security can be traced back to the offender; here it is important to check against the possibility of performing actions that are not logged appropriately.

Clarifying breaches of security. It is difficult to perform a convincing application of the techniques of program analysis unless it is made clear what constitutes an attack. To some extent this is a "moving target" that should be addressed by the security community rather than by the program analysis community. Two facets of this are:

- How does one detect breaches of security? It makes a big difference whether one is only allowed to inspect the values communicated, or also to measure computation time, or even to monitor the areas of a chip that produce the most heat (in case an algorithm has been laid out in silicon).
- The assumptions that can be made on the attacker. As an example, is it sensible to assume that an attacker is always written in the same programming language being used for the application being analysed?

Possible answers might include a formalisation in terms of testing equivalence or one of a number of bisimulations (as has been done for security studies expressed in terms of the spi-calculus); somewhat similar techniques have already been used in program analysis for validating classical analyses (like live variables).

Identifying the important issues. It is a useful technique to reduce the size of the trusted base; for example proof carrying code only requires the checker to be trusted and not the theorem prover proving the result in the first place. As a result, scalability of the analyses will be less of an issue for program analysis; hence it may be feasible to perform relatively expensive analyses in order to gain the necessary precision.

Formal notations are likely to be indispensable in expressing the bad or unacceptable behaviour of systems. This is also related to the issue of proof carrying code in the sense that it is useful for the code to be able to contain annotations about its behaviour. A more general treatment make use of probabilistic notions in order to perform the risk analyses: the rate at which semi-confidential data might leak. Unfortunately, it is unclear how to integrate probabilistic considerations with program analysis.

The precise details of the encryption and decryption techniques are probably less relevant for program analysis. (This presupposes that we ignore the risk of brute forcing cracking a code as was already stated above.)

It is also important to understand the limitations of our techniques in attempting to guarantee against attackers. Various form of code obfuscation might be used to severely limit the abilities of program analysis to detect useful information about the information flow happening in a system (like multiple uses of the same variable for different purposes or obscuring the control structure). Also it may be necessary to integrate dynamic checks (e.g. taintedness checks) with the static analysis in order to ensure confidentiality.

Techniques from program analysis. The classical program analyses are often necessary preconditions for ensuring security; as an example type and memory safety can be used to detect leaks of secure information caused by accessing arrays outside their declared bounds. Only when studying special purpose designed calculi (such as the pi-calculus, the spi-calculus or the ambients calculus) is it meaningful to study security without first performing the standard analyses for ensuring type and memory safety.

Several of the approaches to *program dependency analysis* (like control flow analysis, use definition chaining etc.) may form the core of analyses for detecting information flow (including implicit flows). Another useful technique for maintaining security and integrity constraints is type systems with *effect annotations* (in the form of trusted/untrusted, secure/public etc.). Both of these techniques are likely to be very useful in determining the information flow that lies at the heart of many security notions. Indeed, on top of demonstrating that certain illegal flows do not take place, they may also be useful in determining the potential breaches of security that may result from accidental or deliberate instances of compromising data (e.g. by allowing only partly trusted applications to operate on selected parts of the sensitive data).

Techniques from type and effect systems may be useful for extracting the inherent protocols used for communication in programs (including legacy code) so as to allow validating the overall protocol of a software system. This will also help detecting errors that arise when the system is implemented; similar considerations apply to deliberate code modification so as to facilitate a later attack. Also auditing can be ensured through these techniques by ensuring that data is only accessed or modified after the appropriate logging actions have been performed.

Recent research suggests that program analyses that abstract the semantics of the program might be useful for devising tests that can be used to validate key software components. To be specific, based on the analysis it may be possible to characterise an infinite set of attackers by a finite set of hard attackers; then one can simply analyse the software component under each of the attackers and validate the component in case none of the analyses exhibit illegal behaviour.

Challenges for program analysis. The following challenges are aimed at demonstrating the usefulness of program analysis for security and at studying which of our techniques are likely to be useful for security. It should be possible to obtain progress on both within a few years from now.

- Perform a successful study of one system or protocol; either find an undetected flaw or prove that no attacks of a certain kind can succeed. (This would be useful even for toy protocols from text books such as the alternating bit protocol.)

- Identify techniques, beyond "extended reachability analyses" and "effect annotations" that are applicable; to which extent does this uncover techniques not already known in the security community?

Group members: Chris Hankin, René Rydhof Hansen, Thomas Jensen, Flemming Nielson (editor of the position statement), Jon Riecke, Hanne Riis Nielson, Andrei Sabelfeld, Mooly Sagiv and Helmut Seidl.

2.4 Foundations of program analysis

Program analysis has a number of foundational approaches. We identified ten such approaches, some of which overlap with the others. Those ten approaches, with their defining characteristics, are

1. Type based: Uses an inductive, compositional definition of a typing relation. Type-based analyses may require types or other annotations in programs.
2. Constraint based: Uses systems of equations, inequations, or, more generally, conditional constraints generated from the program by an algorithm. These systems are solved by another algorithm to yield an analysis result.
3. Abstract interpretation: Uses concrete and abstract domains of values, with concretization and abstraction functions between them that form a Galois connection. Widening and narrowing are important techniques to improve the efficiency and precision of the analyses. Algorithms in this area are based primarily on least fixpoint computations.
4. Grammar flow: A method that traces the state spaces of tree automata, and uses alternations between bottom-up and top-down phases.
5. Data flow: A methodology based on equations between sets of values and transfer functions, typically solved by least or greatest fixpoints.
6. Context-free language reachability: A method using context-free grammars, usually focusing on interprocedural problems and usually taking $O(n^3)$ time.
7. Dependence graphs: An analysis technique using data structures with control dependence and flow dependence edges; the data structures are the basis for many algorithms.
8. Temporal logics and model checking: Specifies properties of programs using temporal logics. Checking of properties (that is, determining the truth/falsity of them) is done by model checking. This technique generalizes many ideas from the data flow approach, but is different in using logic to specify "elements" of data flow sets rather than using sets directly.
9. Denotational based: Uses a compositional translation from syntax into mathematical spaces of values. These mathematical spaces, typically called "domains," can be based on syntax as well as partial orders. Strictness analysis is an example.

10. Abstract reduction: Uses a nonstandard rewrite semantics, often requiring an interesting form of loop detection via a syntactic notion of "widening." This technique is used in the CLEAN system, a compiler for a lazy functional language.

Some simple analyses, e.g., Barendregt's neededness analysis for the untyped lambda calculus, do not seem to fit naturally into one of these classifications. We may also have missed some frameworks, since not all analysis communities were represented.

Criteria for choosing an analysis framework

When choosing a suitable framework for approaching an analysis problem, a number of criteria can be used:

1. Utility: Does the framework naturally express the analysis problem?
2. Reliance on syntax: Does the approach rely on the syntax of the language, or does it use some other generic data structure?
3. Specification cleanliness: How well does the approach separate specification of "what" the analysis does from "how" the analysis is performed?
4. Algorithmic issues: Does the analysis suggest an obvious algorithm? An efficient algorithm?
5. Expressivity: Can one formalism express the analysis as well as another? How succinctly can the analysis be expressed? Can an analysis be done as efficiently in one formalism as it can in another? For instance, temporal logics seem strictly more powerful than data flow approaches.
6. Scalability: How well does an analysis in one formalism scale to large programs?
7. Modularity: How modular is the specification of an analysis? Can it be extended easily to larger programs, or to other forms of analysis?
8. Semantics directedness: Does the framework force the analysis to look like the semantics of the language?
9. Tools: Does the framework admit the construction of tools? How wide is the coverage of these tools?

Challenges

1. Low-complexity analyses: Are there methods in the various frameworks for constructing lower complexity analyses? Techniques from different frameworks may need to be combined.
2. Reductions: Can one construct reductions between different analyses written in different frameworks, or between different frameworks themselves? If so, do these reductions preserve complexity bounds of the analyses?

3. Expand scope of foundations: Can one reformulate certain analyses as type systems or in one of the other frameworks? Foundational work should promulgate the use of frameworks in analyses that don't seem to use them, and should consider these analyses in potential expansions of the frameworks.

4. Tools: Can one say general things about how much of a particular framework—that is, how many analyses developed in a certain framework—a tool covers? Can one expand the coverage of tools using foundational methods?

5. Open systems: Can one devise or extend analysis frameworks to dealing with open systems (i.e., those with processes, mobile code, or simply modules)?

6. Probabilities: Can one devise foundational understandings of analyses that use probabilities (e.g., an analysis that predicts, with probabilities rather than yes/no, when a definition reaches a use)?

7. Redundant code: Are there ways of building analyses that eliminate redundant code?

Group members: Chris Hankin, Jörgen Gustavsson, René Rydhof Hansen, Thomas Jensen, Flemming Nielson, Hanne Riis Nielson, Jon G. Riecke (editor of the position statement) and Mooly Sagiv.

2.5 Static and dynamic analyses

> *This subsection represents informal discussions among the group and our first thoughts on this topic. It is not to be taken to be either comprehensive or final.*

There are many meanings attributed to the terms static and dynamic analysis. We will use the term *static* to mean associated with compile-time and *dynamic* with run-time. It is perhaps useful, therefore, to consider these terms as applied to something specific, such as compiler optimization, in order to distinguish them. The major question relating to static and dynamic analyses in general, is "What kinds of feedbacks will occur between the dynamic analyses (i.e., the profiler) and the static analyses, and vice versa?"

Static and dynamic analysis can be used for optimizing programs, or for other applications such as software engineering tools. We first consider a spectrum of static and dynamic techniques for optimizations, and then we briefly summarize other applications.

Optimizing Compilers

There are six classes of optimizing compilers, which do various combinations of static and dynamic analyses and transformations. We summarize our comparisons here and present further discussion below.

- **traditional optimizing compilers**

  - **Analysis.** static transformations
  - **Example.** g++

- **optimizing compilers with specialization**
  - **Analysis.** static, predictive transformations
  - **Example.** vectorizing compilers

- **profile-driven compilers**
  - **Analysis.** dynamic measurements, static transformations
  - **Example.** Vortex, current commercial (IBM,HP,MIPS) compilers, IMPACT, other trace scheduling compilers, Compaq FX32 (continuous offline profiling)

- **run-time compilation**
  - **Analysis.** static analysis and transformations, restricted dynamic code generation
  - **Example.** 'C and Vcode (MIT), Fabius (M. Leone), Tempo (C. Consel), U Washington's compiler (Eggers,Chambers, et. al.)

- **JIT compilers and runtime systems**
  - **Analysis.** dynamic analysis, transformations and code generation done just **before** execution)
  - **Example.** Kaffe, SUN's JDK

- **dynamic optimizing compilers**
  - **Analysis.** dynamic measurements and dynamic transformations **during** execution
  - **Example.** Dynamo (HP), Hotspot (SUN)

Profile-Driven Compilation An *execution trace* is defined as the sequence of dynamic instructions executed by a program on a given input. A *profiler* defines a set of events that can occur at run-time and precisely records how often each event occurs in a given execution trace. For example, a *statement frequency profile* records the number of instances of a program statement that occur in an execution trace. Other examples of profiles include *edge frequency profiles*, *path frequency profiles*, *control-flow traces* (which do not include the data values for the instructions in an execution trace), *memory leak profiles* (as in Purify, *receiver-class profiles* (which record the receiver type for a dynamically dispatched message in an object-oriented program), and *value profiles* (which record the sequence of values taken on by a program variable in an execution trace).

Profile-driven compilation uses the information from a profile to drive the analysis and transformation of a program during compilation. In this paper, profile-driven compilation denotes techniques that make use of profile information collected "offline," in an execution of the program that occurs before compilation. This is in contrast to systems that collect profile information "online" (*i.e.*, at runtime) and transform the program at runtime (*e.g.*, SELF, DYNAMO, Just-in-time Optimizers).

Partial Summary of Previous Work. There is a long history of research in profile-driven compilation. Examples of profile-driven compilers include the following systems:

VORTEX: A compiler for C++, Cecil, and Java that uses a set of profiles to drive specialization (for object-oriented systems) at compilation time.

IMPACT: A compiler for C that uses edge profiles to drive program analysis and transformation. The techniques used include superblock formation for instruction scheduling, trace duplication for improving classic code optimization, and profile-guided function inlining.

IBM/HP/MIT compilers: Other compilers that use profiling at least to drive instruction scheduling and function inlining.

There is also a large amount of research into profile-driven compilation that is not available in a widely-distributed compiler. A recent example includes the work of Ammons and Larus where they use path-profile information to improve the quality of intraprocedural data-flow analysis. Another recent example is the work of Gupta, Berson, and Fang. They have used path-profile information to increase the scope of profitable code motion used in partial-redundancy elimination and partial dead-code elimination.

Open Questions. The success of existing research in profile-driven compilation is encouraging. Further research in this area may be profitable. Directions for future research include the following:

- What other techniques exist for efficient path profiling.
- What kinds of profiling are useful? Recent research has suggested that edge profiles are often sufficient for optimization applications. When are other types of profiling useful? What are other applications of profiling? How can profiling be better used to create program understanding tools (like the hot-path browser)? It may be possible to use profiling techniques that detect "odd" behavior in program executions that exhibit a bug (*e.g.*, paths that are executed in the buggy run that are not executed in healthy runs). Is this effective? Is it useful?
- What opportunities are there for feedback between static analysis and profiling? Can static analysis be used to determine interesting parts of a program to profile? What other opportunities are there to use profiling to focus the attention of static analysis techniques?

Just-in-time Compilers and Runtime Systems. Just-in-time compilers are becoming popular and have been used recently for many Java implementations. The key idea is that an intermediate representation like bytecode is translated to native code on-the-fly. As each method is encountered for the first time, the JIT compiler performs the translation, and the native code is executed. A key point is that the compile-time must be kept to a minimum since this is part of the run-time, and thus any optimizations must also be fast.

Another major performance factor is the runtime system, including the garbage collector. In fact, as JIT compilers produce ever faster code, the fraction of the overall runtime spent in garbage collection increases.

There are two areas to consider: improving the execution of bytecode, and improving the runtime system.

Improving bytecode execution. One static approach is to improve the bytecode before it reaches the JIT. This requires either partial or full analysis of an application, leading to optimized bytecode. In this approach we can use all the standard techniques, as the time for analysis is not crucial.

The completely static approach will not catch all opportunities for optimization because the bytecode is quite high-level, and does not expose things such as register allocation, or array-bounds checks. Further, complete application analysis is not always possible since Java dynamically loads classes.

A possible connection between static analysis and the JIT would be for the static analysis to compute information that is conveyed to the JIT via attributes in the class files. For example, attributes could give hints for the register allocator by providing live variable and/or frequently used variable information. Certainly other attributes could also be productive, and determining these is a major challenge. Also, we foresee the need for new analyses. Another problem is how to provide the attributes in a secure fashion.

Improving the runtime system. One major problem in the runtime system is garbage collection. Although this is a runtime problem, some static analyses can be used to minimize the amount of garbage. A simple technique is to remove dead members, thus reducing the size of some objects. Another popular technique used in garbage-collected languages is to promote heap-allocated structures to the stack, when appropriate scopes can be found. These techniques can be done on bytecode as well. Another technique is to detect reuse of objects.

However, there are other ways of improving garbage collection. For example, the compiler can give explicit deallocation hints. Another recent approach is to use runtime profiling to direct generational garbage collection. The challenges here will be to find what information can be collected statically, and what information is needed at runtime, and to make the garbage collector aware of this information.

Dynamic Optimization Dynamic optimization is the only class of optimization techniques that has no static or off-line component. All three optimization phases: profiling, analysis and code transformation are performed online, i.e., take place at runtime. The program behavior is monitored and the generated profiles are consumed immediately to trigger code optimization.

The online aspect of these technique creates new challenges. New fast profiling techniques are needed since previous techniques developed for off-line profiling are likely to be too costly to be employed on-the-fly. Also, analysis and transformation takes place entirely at runtime and thus have to be performed under severe time constraints.

Examples of dynamic optimization systems include the following:

- The Dynamo system is a transparent optimization system that optimizes native binaries by extracting and optimizing hot execution traces.
- Sun's Java Hotspot is an advanced virtual machine implementation enhanced with dynamic optimization technology for adaptive hot spot detection that focuses optimization on performance-critical code. As of today, concrete documentation on the HotSpot technology is not available.

Research Challenges

- Can static analysis be used to support dynamic optimization? Code annotations may be used to facilitate profiling, for example, by pointing the dynamic profiler to the code regions/procedures of interest as well as by informing the profiler about regions/procedures that can be excluded from profiling because they are known to not possibly constitute hot code. Furthermore, static analysis may be used to communicate hints to the dynamic optimizer to enable more aggressive optimizations.
- Can feedback from the dynamic optimizer from one execution be used to improve dynamic optimization in future executions of the same program?
- In addition to dynamically optimizing the instruction stream, can the data distribution/allocation be optimized dynamically (i.e., promotion from heap to stack allocations)?

Other Applications

Besides driving compiler optimizations, static and dynamic program analyses have been used in many software engineering applications, such as correctness checking of programs and software maintenance.

Perhaps the most widespread application of static analysis for correctness checking is via type checking, which ensures absence of certain kinds of errors in a program. For type-safe languages such as ML, type checking is an integral part of the compilation process. However, type checking can also be used for safety checking in unsafe languages such as C. Static analysis can also be used to aid correctness checking by testing, for example, by automatic test-case generation, and by automatic regression test selection. Program understanding and maintenance applications of static analysis include slicing, merging, and side-effect analysis. Recently, type inference has been used for program maintenance, such as identifying and correcting the Y2K problem.

Dynamic analysis has been used for many program correctness applications, such as detecting memory access errors and race conditions. Dynamic analysis has also been used for program maintenance applications, such as path-spectrum analysis and hot-path browsing.

Interesting questions in this area are similar to the questions raised about the interaction of static and dynamic analysis for compiler optimizations, e.g, how can static analysis

help in reducing the overhead of dynamic analysis techniques? As an example, Patil et al. used the results of static slicing to reduce the overhead of their dynamic technique to detect memory access errors. Another interesting possibility is to explore the applicability of static and dynamic analyses for performance debugging.

Group members: Satish Chandra, Evelyn Duesterwald, Laurie Hendren, David Melski, Shlomit Pinter, Barbara Ryder (editor of the position statement).