Preface

Partial evaluation has reached a point where theory and techniques are mature, substantial systems have been developed, and it appears feasible to use partial evaluation in realistic applications. This development is documented in a series of ACM SIGPLAN-sponsored conferences and workshops, Partial Evaluation and Semantics-Based Program Manipulation, held both in the United States and in Europe.

In 1987, the first meeting of researchers in partial evaluation took place in Gammel Avernæs, Denmark. Almost ten years later, the time was due to evaluate the progress that has been achieved during the last decade and to discuss open problems, novel approaches, and research directions. A seminar at the *International Conference and Research Center for Computer Science* at Schloß Dagstuhl, located in a beautiful scenic region in the southwest of Germany, seemed ideally suited for that purpose.

The meeting brought together specialists on partial evaluation, partial deduction, metacomputation, program analysis, automatic program transformation, and semantics-based program manipulation. The attendants were invited to explore the dimensions of program specialization, program analysis, treatment of programs as data objects, and their applications. Besides discussing major achievements or failures and their reasons, the main topics were:

- **Advances in Theory:** Program specialization has undergone a rapid development during the last decade. Despite its widespread use, a number of theoretical issues still need to be resolved, including efficient treatment of programs as data objects; metalevel techniques including reflection, self-application, and metasystem transition; issues in generating and/or hand-writing program generators; termination and generalization issues in different languages; related topics in program analysis including abstract interpretation, flow analysis, and type inference; the relationships between different transformation paradigms, such as automated deduction, theorem proving, and program synthesis.

- **Towards Computational Practice:** Academic research has thrived in many locations. Broad practical experience has been gained, and stronger and larger program specializers have been built for a variety of languages, including Scheme, ML, Prolog, and C. Work is being initiated to bring the achievements of theory into practical use but a number of pragmatic issues still need to be resolved: progress towards medium- and large-scale applications; environments and user interfaces (*e.g.*, binding-time debuggers); integration of partial evaluation into the software development process; automated software reuse.

- **Larger Perspectives:** We also wanted to critically assess state-of-the-art techniques, summarize new approaches and insights, and survey challenging problems.

The Proceedings

All participants were invited to submit a full paper of their contribution for a proceedings volume [7]. The submitted papers have been reviewed with outside assistance and each paper read by at least three referees. Technical quality, significance, and originality were the primary criteria for selection. The selected papers cover a wide and representative spectrum of topics and document state-of-the-art research activities in the area of partial evaluation. Three surveys were invited to complete the volume:

- In *What not to do when writing an interpreter for specialization*, Neil Jones carefully reviews the (sometimes non-obvious) pitfalls that must be avoided by the partial-evaluation apprentice in order to successfully generate compilers by partial evaluation,
- In *A comparative revisitation of some program transformation techniques*, Alberto Pettorossi and Maurizio Proietti survey the use of well-known transformation techniques in recent work, and
- In *Metacomputation: Metasystem transition plus supercompilation*, Valentin Turchin provides us with a personal account of the history and the present state of supercompilation and metasystem transitions. Professor Turchin presented his invited lecture on his 65th birthday, during the seminar.

Here is a brief topical summary of the contributions in the proceedings:

Partial evaluation:

- Thomas Reps and Todd Turnidge. *Program specialization via program slicing.* Demonstrates that specialization can be achieved by program slicing. A bridge-building paper between two research areas.
- Torben Mogensen. *Evolution of partial evaluators: Removing inherited limits.* Provides a rationalized, unified, and insightful view of recent developments in partial evaluation.

- Michael Sperber. *Self-applicable online partial evaluation.* Reports the successful self-application of a realistic online partial evaluator. Solves a long-standing problem.

- Alain Miniussi and David Sherman. *Squeezing intermediate construction in equational programs.* Applies specialization of stack-machine code to optimize equational programs.

Imperative programming:

- Mikhail A. Bulyonkov and Dmitry V. Kochetov. *Practical aspects of specialization of Algol-like programs.* Presents a new technique for decreasing the memory usage in the specialization of imperative programs.

- Alexander Sakharov. *Specialization of imperative programs through analysis of relational expressions.* Presents a powerful optimizer applying techniques from supercompilation and generalized partial computation to improve imperative code, using a dependence graph.

Functional programming:

- John Hughes. *Type specialisation for the lambda calculus.* An elegant step towards solving the (in)famous tagging problem in specializing higher-order interpreters for typed programs.

- Olivier Danvy. *Pragmatics of type-directed partial evaluation.* Extends the author's approach to specialize compiled programs by avoiding computation duplication and making residual programs readable, using type annotations.

- Peter Sestoft. *ML pattern match compilation and partial evaluation.* Applies information-propagation techniques to the compilation of pattern matching. An excellent application of partial evaluation in a compiler.

Logic programming:

- Alberto Pettorossi and Maurizio Proietti. *Automatic techniques for logic program specialization.* Describes a general method to specialize logic programs with respect to sets of partially known data given as a conjunction of atoms. The correctness of the technique is shown by a proof based on unfolding/folding transformations.

- Michael Leuschel and Bern Martens. *Global control for partial deduction through characteristic atoms and global trees.* Describes a global control of partial deduction that ensures effective specialization and fine-grained polyvariance, and whose termination only depends on the termination of the local control component—a significant progress.

Metacomputation by supercompilation:

- Andrei P. Nemytykh, Victoria A. Pinchuk, and Valentin F. Turchin. *A self-applicable supercompiler.* Shows the feasibility of self-application of supercompilers by demonstrating its use with examples. The solution of a long-standing open problem.
- Robert Glück and Morten Heine Sørensen. *A roadmap to metacomputation by supercompilation.* A survey paper with a careful explanation of supercompilation and its application in metacomputation, reporting state-of-the-art techniques and including a comprehensive list of references on the topic.

Generating extensions:

- Jesper Jørgensen and Michael Leuschel. *Efficiently generating efficient generating extensions in Prolog.* Reports the first successful hand-written program-generator generator (cogen) for a logic programming language. Exhibits significant speedups with respect to generic specializers and cogens generated by self-application.
- Scott Draves. *Compiler generation for interactive graphics using intermediate code.* Reports on a hand-written cogen for an intermediate language and its successful application to graphics programming. Includes the possibility of runtime code generation for specialized procedures.

Multi-level systems:

- Flemming Nielson and Hanne Riis Nielson. *Multi-level lambda-calculi: an algebraic description.* Considering the recent surge of interest in multi-level calculi, the authors revisit the "B-level-languages" of their book on two-level functional languages. In this paper, they provide us with a unifying treatment of various different calculi by exhibiting a general algebraic framework which can be instantiated to all other known calculi.

- John Hatcliff and Robert Glück. *Reasoning about hierarchies of online program specialization systems.* Successful self-application and metasystem transition is harder to achieve in online specializers than in offline ones. The paper develops a framework which facilitates metasystem transitions involving multiple levels for online specializers.

Program analyses:

- John Gallagher and Laura Lafave. *Regular approximation of computation paths in logic and functional languages.* Introduces computation paths as a novel means to control the polyvariance and termination of specialization techniques for logic and functional languages.
- Wei-Ngan Chin, Siau-Cheng Khoo, and Peter Thiemann. *Synchronization analyses for multiple recursion parameters.* Extends the tupling transformation to functions with multiple recursion parameters. Develops an original framework to identify synchronous handling of recursion parameters and gives terminating algorithms to perform the tupling transformation.

Applications:

- Sandrine Blazy and Philippe Facon. *An approach for the understanding of scientific application programs based on program specialization.* Applies specialization to understanding and maintaining "dusty deck" FORTRAN programs. The paper extends previous work with an interprocedural analysis.
- Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. *A uniform approach for compile-time and run-time specialization.* Presents a language-independent partial-evaluation framework which is suited for standard partial-evaluation techniques, as well as for run-time specialization. The major application is run-time specialization of one of the most widely used programming languages: C.

Further Partial-Evaluation Resources

Good starting points for the study of partial evaluation are the textbook by Jones, Gomard, and Sestoft [13], the tutorial notes by Consel and Danvy [5], and the tutorial notes on partial deduction by Gallagher [9]. Further material can be found

in the proceedings of the Gammel Avernæs meeting [2, 8], in the proceedings of the ACM conferences and workshops on Partial Evaluation and Semantics-Based Program Manipulation (PEPM) [4, 10, 16, 17, 19], and in special issues of various journals [11,12,14]. An online bibliography is being maintained by Peter Sestoft [18].

Acknowledgements

We wish to thank the staff at Schloß Dagstuhl for their efficient assistance and for making the meeting possible. Thanks are also due to Elisabeth Meinhardt and Gebhard Engelhart for the help with organizational matters. And finally, thanks to all participants for a lively and inspiring meeting.

1 Seminar Program

Monday, February 12

Session 1: Specialization of Imperative Programs I
Chair: Sandrine Blazy

A Uniform Approach for Compile-Time and Run-Time Specialization
Charles Consel, Luke Hornof, François Noël, Jacques Noyé, Nicolae Volanschi

Session 2: Specialization of Imperative Programs II
Chair: Julia L. Lawall

Data Specialization
Erik Ruf, Todd Knoblock

Toward Partial Evaluation of Side-Effecting Scheme
Kenichi Asai

Session 3: Specialization of Imperative Programs III
Chair: Peter Sestoft

Practical Aspects of Specialization of Algol-like Programs
Mikhail A. Bulyonkov, Dmitry V. Kochetov

Specialization of Imperative Programs through Analysis of Relational Expressions
Alexander Sakharov

Session 4: Specialization of Imperative Programs IV
Chair: Alexander Letichevsky

An Automatic Interprocedural Analysis for the Understanding of Scientific Application Programs Based on Program Specialization
Sandrine Blazy, Philippe Facon

Squeezing Intermediate Construction Equational Programs
Alain Miniussi, David Sherman

Session 5: Demonstrations
Chair: Michael Sperber

The M2Mix System
Mikhail A. Bulyonkov, Dmitry V. Kochetov

Tempo: A Partial Evaluation System for the C Programming Language
Charles Consel, Luke Hornof, François Noël, Jacques Noyé, Nicolae Volanschi

Meta-ML
Tim Sheard

Tuesday, February 13

Session 6: Theory I
Chair: Helmut Schwichtenberg

Multi-Level Lambda-Calculi: an Algebraic Description
Flemming Nielson, Hanne Riis Nielson

Regular Approximations of Computation Paths in Logic and Functional Languages

John P. Gallagher, Laura Lafave

The Functional Approach to Analyses and Transformations of Imperative Programs

Barbara Moura, Charles Consel

Session 7: Invited Lecture
Chair: Mikhail Bulyonkov

What Not *to Do When Writing an Interpreter for Specialisation*
Neil D. Jones

Session 8: Theory II
Chair: Torben Mogensen

Program Development by Proof Transformation
Helmut Schwichtenberg

Deriving and Applying Program Transformers
David Basin

Session 9: Logic Programming I
Chair: John Gallagher

A Theory of Logic Program Specialization and Generalization for Dealing with Input Data Properties
Alberto Pettorossi, Maurizio Proietti

Global Control for Partial Deduction through Characteristic Atoms and Global Trees

Michael Leuschel, Bern Martens

Session 10: Logic Programming II
Chair: Manuel Hermenegildo

Efficiently Generating Efficient Generating Extensions in Prolog
Jesper Jørgensen, Michael Leuschel

Semantics-Directed Generation of Abstract Machines
Stephan Diehl

Session 11: Demonstrations
Chair: Kyung-Goo Doh

Regular Approximations of Computation Paths in Logic and Functional Languages

John P. Gallagher, Laura Lafave

Data Specialization of Graphics Shading Code
Todd Knoblock, Erik Ruf

Psychedelic Graphics
Scott Draves

Experiments on Partial Evaluation in APS
Alexander Letichevsky

Wednesday, February 14

Session 12: Supercompilation I
Chair: John Hatcliff

A Self-Applicable Supercompiler
Andrei P. Nemytykh, Victoria A. Pinchuk, Valentin F. Turchin

Specification of a Class of Supercompilers
Andrei Klimov

A Roadmap to Metacomputation by Supercompilation
Robert Glück, Morten Heine Sørensen

Session 13: Supercompilation II
Chair: Neil D. Jones

Nonstandard Semantics of Programming Languages
Sergei Abramov

Invited Lecture: *Metacomputation: MST plus SCP*
Valentin F. Turchin

Thursday, February 15

Session 14: Functional Programming I
Chair: Charles Consel

Type Specialisation for the Lambda Calculus
John Hughes

Some Relationships between Partial Evaluation and Meta-Programming
Tim Sheard

Compiler Generation for Interactive Graphics using Intermediate Code
Scott Draves

Session 15: Functional Programming II
Chair: Yoshihiko Futamura

Self-Applicable Online Partial Evaluation
Michael Sperber

Program Specialization via Program Slicing
Thomas Reps, Todd Turnidge

A Comparative Revisitation of Some Program Transformation Techniques
Alberto Pettorossi, Maurizio Proietti

Session 16: Functional Programming III
Chair: Erik Ruf

Evolution of Partial Evaluators: Removing Inherited Limits
Torben Æ. Mogensen

ML Pattern Match Compilation and Partial Evaluation
Peter Sestoft

Session 17: Functional Programming IV
Chair: Thomas Reps

Lockstep Transformations
Kristian Nielsen

Unravelling Computations
S. Doaitse Swierstra

Session 18: Demonstrations
Chair: Wei-Ngan Chin

A Binding-Time Analysis Ensuring Termination of Partial Evaluation
Neil D. Jones

C-Mix Demonstration
Peter H. Andersen

Session 19: Demonstrations
Chair: Todd Knoblock

Program Slicing, Differencing, and Merging
Thomas Reps

Applying Multiple Specialization to Program Parallelization
Manuel Hermenegildo, German Puebla

Friday, February 16

Session 20: Functional Programming V
Chair: Dirk Dussart

Hylomorphism in Triplet Form, an Ideal Internal Form for Functional Program Optimization
Akihiko Takano

Partial Evaluation and Separate Compilation
Rogardt Heldal, John Hughes

Session 21: Functional Programming VI
Chair: David Schmidt

Synchronization Analyses for Multiple Recursion Parameters
Wei-Ngan Chin, Siau-Cheng Khoo, Peter Thiemann

Reasoning about Hierarchies of Online Program Specialization Systems
John Hatcliff, Robert Glück

Pragmatics of Type-Directed Partial Evaluation
Olivier Danvy

2 Abstracts of Presentations (alphabetical)

Nonstandard Semantics of Programming Languages

Sergei Abramov

This paper introduces notions of *semantics modifier* and *on-standard semantics* of programming languages. The *universal resolving algorithm* and *neighborhood analyzer* are considered as examples of semantics modifiers. The precise definition of semantics modifier and non-standard semantics is given and illustrated with additional examples. Finally it is shown that programming tools for effective implementation of non-standard semantics – non-standard interpreters, non-standard compilers – can be obtained using a powerful specializer (e.g. a supercompiler) as a result of several metasystem transitions.

The paper is available by anonymous FTP from
`ftp://ftp.botik.ru/pub/local/Sergei.Abramov/nons-sem.dvi.zip`

C-Mix Demonstration

Peter Holst Andersen

C-Mix is a partial evaluator for the ANSI C programming language. It was originally developed by Lars Ole Andersen, and is currently developed and maintained by the author. The theory is developed to cover almost all of ANSI C [1], and most of it is implemented.

The implementation supports programs that are *strictly conforming* to the ANSI C standard. A strictly conforming program shall only use features described in the standard, and may not produce any result that depends on undefined, unspecified or implementation-defined behavior. In general, C-Mix will not optimize non-strictly conforming parts of the program, but rather suspend the offending operations to run-time.

The C-Mix prototype implementation is available for non-commercial use for research and teaching. The distribution includes source code and a user manual. More information and links to C-Mix papers can be found on the C-Mix home page:
`http://www.diku.dk/research-groups/topps/activities/cmix.html`

Towards Partial Evaluation of Side-Effecting Scheme

Kenichi Asai

I will present my attempts to make a partial evaluator for Scheme programs which contain side-effects using the online approach. After presenting my motivation in terms of reflection, I will show the preaction mechanism, which are used to residualize I/O-type side-effects. It also gives us a convenient way to avoid a code elimination problem and to keep the order of side-effects. Then, I will consider how I can cope with assignments to variables and data structures. The concept of destroy lists and scopes are introduced here to find executable assignments. Despite the various attempts, it is still insufficient to use in reflective languages, and some kind of offline analyses seem to be inevitable for the better partial evaluation.

Deriving and Applying Program Transformers

David Basin

Not available.

An Approach for the Understanding of Scientific Application Programs Based on Program Specialization

Sandrine Blazy and Philippe Facon

This paper reports on an approach for improving the understanding of old programs which have become very complex due to numerous extensions. We have adapted partial evaluation techniques for program understanding. These techniques mainly use propagation through statements and simplifications of statements. We focus here on the automatic interprocedural analysis and we specify both tasks for call-statements, in terms of inference rules with notations taken from the specification languages B and VDM. We describe how we have implemented in a tool and used that interprocedural analysis to improve program understanding. The difficulty of that analysis comes from the lack of well defined interprocedural mechanisms and the complexity of visibility rules in Fortran.

Practical Aspects of Specialization of Algol-like Programs

Mikhail A. Bulyonkov and Dmitry V. Kochetov

A "linearized" scheme of polyvariant specialization for imperative languages is described in the paper. The scheme is intended for increasing efficiency of specialization. Main properties of the scheme are linear generation of residual code and single

memory shared by different variants of specialization process. The scheme was used in partial evaluator for the Modula-2 language. Some benchmarks of the evaluator are discussed to demonstrate efficiency of the processor.

The M2Mix Partial Evaluator

Mikhail A. Bulyonkov and Dmitry V. Kochetov

The *M2Mix* is a partial evaluator for the complete Modula-2 language. It is implemented as a compiler generator, so it accepts an annotated Modula-2 program and static data and produces another Modula-2 program, called generating extension, which in turn is (compiled and) executed to generate residual code in a slightly extended Modula-2 language. Special postprocessing phase optimizes the residual code and brings it back to Modula-2.

The static/dynamic annotations are provided by a user in a form of pseudo-comments. Typically some input files which store unknown data are annotated as dynamic. However, a user may annotate any global or local variable or a procedure parameter in order to control partial evaluator. Special simple annotations are required for external procedures.

Upon a user's request the *M2Mix* can generate a listing containing information from various program analysis, such as binding time analysis, pointers analysis, configuration analysis, etc. Since in the M2Mix system static/dynamic division depends not only on def/use chains, the special binding-time debugger explaining *why* something became dynamic turned out to be especially helpful.

The *M2Mix* system is implemented in C and runs on IBM PC under MS DOS or OS/2. Obviously a Modula-2 compiler is needed to compile source programs and generating extensions. The system shows very modest memory requirements and high speed at the same time. For example, specialization of *Lex* kernel interpreter (translated into Modula-2 and slightly modified for better specializability) with respect to tables, produced by *Lex* for Modula-2, performed on a machine with Intel 486 processor and 4MB of RAM took 2.52 seconds and 189KB of extra memory.

Synchronization Analyses for Multiple Recursion Parameters

Wei-Ngan Chin, Siau-Cheng Khoo, and Peter Thiemann

Tupling is a transformation tactic to obtain new functions, without redundant calls and/or multiple traversals of common inputs. In [3], we presented an automatic method for tupling functions with a single recursion parameter each. In this paper, we propose a new family of parameter analyses, called *synchronization analyses*, to

help extend the tupling method to functions with multiple recursion parameters. To achieve better optimization, we formulate three different forms of tupling optimizations for the elimination of intra-call traversals, the elimination of inter-call traversals and the elimination of redundant calls. We also guarantee the safety of the extended method by ensuring that its transformation always terminates.

A Uniform Approach for Compile-Time and Run-Time Specialization
Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi

As partial evaluation gets more mature, it is now possible to use this program transformation technique to tackle realistic languages and real-size application programs. However, this evolution raises a number of critical issues that need to be addressed before the approach becomes truly practical. First of all, most existing partial evaluators have been developed based on the assumption that they could process any kind of application program. This attempt to develop universal partial evaluators does not address some critical needs of real-size application programs. Furthermore, as partial evaluators treat richer and richer languages , their size and complexity increase drastically. This increasing complexity revealed the need to enhance design principles. Finally, exclusively specializing programs at compile time seriously limits the applicability of partial evaluation since a large class of invariants in real-size programs are not known until run time and therefore cannot be taken into account. In this paper, we propose design principles and techniques to deal with each of these issues. By defining an architecture for a partial evaluator and its essential components, we are able to to tackle a rich language like C without compromising the design and the structure of the resulting implementation. By designing a partial evaluator targeted towards a specific application area, namely system software, we have developed a system capable of treating realistic programs. Because our approach to designing a partial evaluator clearly separates preprocessing and processing aspects, we are able to introduce run-time specialization in our partial evaluation system as a new way of exploiting information produced by the preprocessing phase.

Tempo: A Partial Evaluation System for the C Programming Language
Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi

Tempo is a partial evaluation system for the the C programming language. It incorporates an off-line partial evaluation strategy, consisting of an analysis phase and a specialization phase. Tempo is designed so that the specialization phase can be performed either at compile time or at run time. To avoid the problems associated

with universal partial evaluators, Tempo was created to specialize a well defined set of of applications, namely "system" programs. This choice allows realistic programs to be treated.

We will demonstrate the principles, functionality, and unique characteristics (flow-sensitive binding-time analysis, accurate treatment of complex data structures and pointers, run-time specialization, etc.) of Tempo by running it on a number of example programs.

Type-Directed Partial Evaluation

Olivier Danvy

Type-directed partial evaluation stems from the residualization of arbitrary static values in dynamic contexts, given their type. Its algorithm coincides with the one for coercing a subtype value into a supertype value, which itself coincides with the one of normalization in the lambda-calculus. Type-directed partial evaluation is thus used to specialize compiled, closed programs, given their type.

Since Similix, let-insertion is a cornerstone of partial evaluators for call-by-value procedural programs with computational effects. It prevents the duplication of residual computations, and more generally maintains the order of dynamic side effects in residual programs.

This article describes the extension of type-directed partial evaluation to insert residual let expressions. This extension requires the user to annotate arrow types with effect information. It is achieved by delimiting and abstracting control, comparably to continuation-based specialization in direct style. It enables type-directed partial evaluation of effectful programs (eg, a definitional lambda-interpreter for an imperative language) that are in direct style. The residual programs are in A-normal form.

Semantics-Directed Generation of Abstract Machines

Stephan Diehl

Abstract machines are virtual target architectures which support the concepts of the source language. We present a generator which automatically produces a compiler and abstract machine from a natural semantics specification. Whereas all existing semantics-directed compiler generators use partial evaluation or a direct translation into a fixed target language, we chose pass separation as the key transformation of our system. The execution times of the abstract machine programs produced by our generated compiler compare to those of target programs produced by compilers

generated by other semantics-directed generators. The generated specifications of compilers and abstract machines are suitable as a starting point for handwriting compilers and abstract machines. Our generator is fully automated and its core transformations are proved correct.

Compiler Generation for Interactive Graphics using Intermediate Code

Scott Draves

This paper describes a compiler generator (cogen) designed for interactive graphics, and presents preliminary results of its application to pixel-level code. The cogen accepts and produces a reflective intermediate code in continuation-passing, closure-passing style. This allows low overhead run-time code generation as well as multi-stage compiler generation. We extend partial evaluation techniques by allowing unrestricted lifting and *partially static integers*. In addition to some standard examples, we examine graphics kernels such as bcopy, one-dimensional finite filtering, and packed pixel access.

Psychedelic Graphics

Scott Draves

Not available.

Regular Approximation of Computation Paths in Logic and Functional Languages

John P. Gallagher and Laura Lafave

The aim of this work is to compute descriptions of successful computation paths in logic or functional program executions. Computations paths are represented as terms, built from special constructor symbols, each constructor and symbol corresponding to a specific clause or equation in a program. Such terms, called trace-terms, are abstractions of computation trees, which capture information about the control flow of the program. A method of approximating trace-terms is described, based on well-established methods for computing regular approximations of terms. The special function symbols are first introduced into programs as extra arguments in predicates or functions. Then a regular approximation (with respect to a given class of program executions) is computed, giving a regular description of the terms appearing in every argument in the program. The approximation of the extra arguments (the trace-terms) can then be examined to see what computation paths were followed during the computation. This information can then be used to control an

off-line specialisation system. A key aspect of the analysis is the use of suitable widening operations during the regular approximation, in order to preserve information on determinacy and branching structure of the computation. This method is applicable to both logic and functional languages, and appears to offer appropriate control information in both formalisms.

A Roadmap to Metacomputation by Supercompilation

Robert Glück and Morten Heine Sørensen

This paper gives a gentle introduction to Turchin's supercompilation and its applications in metacomputation with an emphasis on recent developments. First, a complete supercompiler, including positive driving and generalization, is defined for a functional language and illustrated with examples. Then a taxonomy of related transformers is given and compared to the supercompiler. Finally, we put supercompilation into the larger perspective of metacomputation and consider three metacomputation tasks: specialization, composition, and inversion.

Reasoning about Hierarchies of Online Program Specialization Systems

John Hatcliff and Robert Glück

The conventional wisdom is that online techniques are not well-suited for obtaining useful results from metasystem hierarchies. However, this conclusion ignores both practical and theoretical progress. Our goal is to identify and clarify the foundational issues involved in hierarchies of online specialization systems. To achieve this, we develop a very simple online specialization system which focuses tightly on the following problematic points of online specialization: (1) semantics of specialization, (2) properties of program encodings and identifying position of entities in a hierarchy, (3) tracking unknown values across levels in metasystem hierarchies.

Partial Evaluation and Separate Compilation

Rogardt Heldal and John Hughes

Hitherto all partial evaluators have processed a complete program to produce a complete residual program. We are interested in treating programs as collections of modules which can be processed independently: 'separate partial evaluation', so to speak. In this paper we still assume that the *original program* is processed in its entirety, but we show how to specialise it to the static data bit-by-bit, generating a different module for each bit. When the program to be specialised is an interpreter,

this corresponds to specialising it to one module of its object language at a time: each module of the object language gives rise to one module of the residual program.

We have been forced to distinguish *three* different binding-times, which we call static, *late static*, and dynamic. Static data is present when a module is compiled. Late static data is present when compiling a module which *imports* such a module. Dynamic data is of course present only at run-time. As an example, we have constructed an interpreter for a simple language with jumps, in which each module can import other modules and jump to labels in them. The main function in the interpreter has three parameters: the module, a label to interpret from, and the register contents. The first is of course static, and the last is dynamic. But the second parameter is late static: this function is invoked from other modules, and at the time of invocation the label to jump to will be known.

When we specialise the interpreter to a 'module', only the static data is known. Late static values are therefore treated as dynamic. But the residual code is divided into two parts: 'purely dynamic' functions with only dynamic parameters, and the rest. Functions with late static parameters are placed in an *interface file*; they may be invoked when other modules are compiled, and are then specialised further with the late static parameters treated as static. Purely dynamic functions on the other hand need no further specialisation, and are placed in a *code module* for immediate compilation. These functions need not be processed further during later specialisations, but they may of course be called from the interface file, which will result in residual calls to the code module from other compiled modules.

Applying Abstract Multiple Specialization to Program Parallelization

Manuel Hermenegildo and Germán Puebla

We present and demonstrate an application of automatic multiple specialization in logic program parallelization, in the context of the CIAO system. CIAO is a multi-paradigm compiler and run-time system aimed at providing efficient implementations of a range of LP, CLP, and CC programming languages. The CIAO compiler performs automatic parallelization of (constraint) logic programs using global analysis information based on abstract interpretation. During parallelization, analysis information is used to detect calls that should be run in parallel using as few run-time tests as possible. Sometimes, especially when the user provides no information to the analyzer, it is not possible to parallelize a program without introducing run-time tests. In this case, multiple specialization is used to optimize the automatically parallelized programs. First, multi-variant re-analysis of the program is efficiently performed using incremental analysis algorithms. Then, the optimizations allowed in each version of a procedure generated during multi-variant (re-)analysis are determined and a minimizing algorithm is applied which obtains minimal programs while

retaining all possible optimizations. Finally, this minimal program is materialized and optimized. The program specialization used is abstract in the sense that it is performed with respect to abstract values rather than concrete ones, as is the case in more traditional partial evaluation systems. This is, to best of our knowledge, the first logic program compiler to automatically make use of abstract (multiple) specialization.

Type Specialisation for the Lambda Calculus

John Hughes

Partial evaluation of typed languages has long suffered from the problem that the types appearing in residual programs are constrained too strongly by those appearing in the unspecialised original program. In the simplest case, the only types that can appear in residual programs are those from the original source. When for example programs are compiled by specialising an interpreter, then the only types that can appear in the compiled code are those used in the interpreter. In particular, since the interpreter must represent values of different types by injecting them into a universal type, then in the compiled code values are also tagged members of this type, and tags must be checked whenever a value is used. If the language being compiled is itself typed, then these tags and checks are unnecessary. When the interpreter is a self-interpreter, the 'optimality' criterion is not met: mix sint p = p because the left hand side tags its data and the right hand side does not. In this paper we present what we believe is the first partial evaluator that can remove these tags. The object language is the simply typed lambda calculus with a two-level type system: static tags are removed, while dynamic tags remain. A novel feature of our partial evaluator is that we can derive static information about expressions which others treat as purely dynamic, such as a dynamic conditional. But just as compilers reject programs whose types do not match, so our partial evaluator rejects programs whose static information does not match. For example, the two arms of a dynamic conditional must carry the same static information. We are forced to relax the functional nature of the mapping from source expressions to static information; rather a relation holds between them, which means that the same source expression may be related to many different static values. Static information can't be computed by a bottom-up evaluation, therefore, and indeed our specialiser is expressed not as an interpreter, but as a system of inference rules specifying how static information can be inferred. Unsurprisingly, the inference of static information (which includes type information) is very similar to type inference. The static value inferred for an expression is then used to derive a suitable residual type for the specialised expression. Since the same expression in the source can be related to many different static values, its specialisations can have many different residual

types. An interesting aspect is that our specialiser need not use unfolding: we can 'evaluate' static expressions without unfolding function calls by inferring their static value. To demonstrate this we have implemented a specialised which doesn't unfold at all. But for this reason our specialiser is not 'optimal': when we compute mix sint p we eliminate type tags, but the residual program must be unfolded somewhat in order to recover p. The present specialised is just a toy, but we believe the techniques will scale up to make possible improved specialisation of real typed languages.

What *Not* to Do When Writing an Interpreter for Specialisation

Neil D. Jones

A partial evaluator, given a program and a known "static" part of its input data, outputs a residual program in which computations depending only on the static data have been performed. *Ideally* the partial evaluator would be a "black box" able to extract nontrivial static computations whenever possible; which never fails to terminate; and which always produces residual programs of reasonable size and maximal efficiency, so all possible static computations have been done. *Practically* speaking, partial evaluators fall short of this goal; they may loop, sometimes pessimise, and/or explode code size. A partial evaluator is analogous to a spirited horse: while impressive results can be obtained when used well, the user must know what he/she is doing. Our thesis is that *this knowledge can be communicated* to new users of these tools. This paper presents a series of examples, concentrating on a quite broad and on the whole successful application area: using specialisation to remove interpretative overhead. It presents a series of examples, both positive and negative, to illustrate the effects of program style on the efficiency and size of the of target programs obtained by specialising an interpreter with respect to a known source program.

BTA Algorithms to Ensure Termination of Off-line Partial Evaluation

Neil D. Jones and Arne J. Glenstrup

A partial evaluator, given a program and a known "static" part of its input data, outputs a residual program in which computations depending only on the static data have been precomputed. Ideally the partial evaluator is a "black box" which both extracts nontrivial static computations whenever possible, and never fails to terminate. Practically speaking, partial evaluators fall short of this goal: they sometimes loop (typical of functional programming systems), or never loop but often give excessively conservative results (typical in logic programming).

This paper presents efficient algorithms (currently being implemented) for binding-time analysis as used by off-line specialisers. These algorithms ensure that the

specialiser performs nontrivial static computations if possible, and is at the same time guaranteed to terminate.

Further, the developed techniques have more general applications, including termination of term rewriting systems.

Efficiently Generating Efficient Generating Extensions in Prolog

Jesper Jørgensen and Michael Leuschel

The so called "cogen approach", writing a compiler generator instead of a specializer, to program specialization has been used with considerably success in partial evaluation of both functional and imperative languages. This paper demonstrates that this approach is also applicable to partial evaluation of logic programming languages, also called partial deduction. Self-application has not been as much in focus in partial deduction as in partial evaluation of functional and imperative languages, and the attempts to self-apply partial deduction system have, of yet, not been all together that successful. So especially for partial deduction the cogen approach could prove to have a considerable importance when it come to practical applications. It is demonstrated that using the cogen approach one gets very efficient compiler generators which generate very efficient generating extensions which in turn yield very good and non-trivial specialisation.

Specification of a Class of Supercompilers

Andrei Klimov

The specification of a class of supercompilers for a small functional language in form of Natural Semantics is presented. It defines a relation between source and residual programs with respect to an initial configuration, and the notions of driving, generalization, configuration splitting. The specification states *what* is to be a supercompiled program, and says nothing about *how* a supercompiler takes decisions to fold, to generalize, to split a configuration.

The purpose of the specification is to divide the task of proving properties of supercompilers into two parts: proving the property of the specification and proving that a particular supercompiler agrees with the specification. For example, the correctness theorem is proven once for the specification. Certain termination properties can be formulated and proven as well, such as the termination of the Turchin's algorithm of configuration splitting (1987).

Global Control for Partial Deduction through Characteristic Atoms and Global Trees

Michael Leuschel and Bern Martens

Recently, considerable advances have been made in the (online) control of logic program specialisation. A clear conceptual distinction has been established between local and global control and on both levels concrete strategies as well as general frameworks have been proposed. For global control in particular, recent work has developed concrete techniques based on the preservation of characteristic trees (limited, however, by a given, arbitrary depth bound) to obtain a very precise control of polyvariance. On the other hand, the concept of an m-tree has been introduced as a refined way to trace "relationships" of partially deduced atoms, thus serving as the basis for a general framework within which global termination of partial deduction can be ensured in a non ad hoc way. Blending both, formerly separate, contributions, in this paper, we present an elegant and sophisticated technique to globally control partial deduction of normal logic programs. Leaving unspecified the specific local control one may wish to plug in, we develop a concrete global control strategy combining the use of characteristic atoms and trees with global (m-)trees. We thus obtain partial deduction that always terminates in an elegant, non ad hoc way, while providing excellent specialisation as well as fine-grained (but reasonable) polyvariance. We conjecture that a similar approach may contribute to improve upon current (online) control strategies for functional program transformation methods such as (positive) supercompilation.

The Experiments on Partial Evaluation in APS

Alexander A. Letichevsky

Algebraic programming system APS [15] has been used for the development of a partial evaluator for extended higher order untyped lambda calculus. Partial evaluator is considered as a generic program which defines an operational semantics of a programming language based on $\breve{}$-calculus. To obtain satisfactory results on self-application, three consistent levels of extension for $\breve{}$-calculus are considered: pure $\breve{}$-calculus, semantic extension compatible with its classical denotational semantics, and metalevel extension, which allows to manipulate with $\breve{}$-terms as syntactic objects. The partial evaluators for pure $\breve{}$-calculus and its semantic extension are proved to possess main properties: correctness, completeness, termination and optimality with respect to generic parameters. There are no restrictions on specialized program and partial evaluator diverges only when specialized program diverges for all values of dynamic variables. Evaluators use online binding time analysis and are

expressed as systems of rewriting rules implemented in algebraic programming system APS. All partial evaluators may be expressed in metalevel extension of $\breve{}$-calculi and used for self-application and specialization.

Current Developments in Partial Evaluation for Equational Programs

Alain Miniussi and David Sherman

Equational programs, which use term-rewriting as their basic implementation method, can be greatly improved by partial evaluation of intermediate code programs written in EM code. Such transformation removes various kinds of overhead associated with rewriting, such as intermediate rewriting steps, boxing and unboxing of arithmetic values, and intermediate constructions resulting from recursive definitions. Unfolding strategies for equational programs must be both *terminating*, and *good enough* with respect to pragmatic criteria. In this paper we give a general introduction to the particular problems associated with partial evaluation of equational programs, and propose some criteria for deciding whether an unfolding strategy is good enough from a practical standpoint. We then present a new algorithm for driving unfolding of EM code programs, and show that, in addition to terminating, it produces a good result. As we have already implemented the new strategy, this final point is demonstrated with a number of concrete examples.

Evolution of Partial Evaluators: Removing Inherited Limits

Torben Æ. Mogensen

We show the evolution of partial evaluators over the past ten years from a particular perspective: the attempt to remove limits on the structure of residual programs that are inherited by structural bounds in the original programs. It will often be the case that a language allows an unbounded number or size of a particular features, but each program (being finite) will only have a finite number or size of these features. If the residual programs cannot overcome the bounds given in the original program, that can be seen as a weakness in the partial evaluator, as it potentially limits the effectiveness of residual programs. The inherited limits are best observed through specializing a self-interpreter and examining the object programs produced by specialisation of this. We show how historical developments in partial evaluators gradually remove inherited limits, and suggest how this principle can be used as a guideline for further development.

The Functional Approach to Analyses and Transformations of Imperative Programs

Barbara Moura and Charles Consel

Since the late eighties, the imperative programming language community has been studying intermediate program representations, where imperative features are represented functionally [6, 22]. These intermediate representations, such as Static Single Assignment [6], are widely used in advanced optimizing compilation. However, although imperative features are dealt with in a functional fashion, they are still based upon an imperative framework. We propose to go a step further by transforming an imperative program into a fully executable functional form.
We present a set of transformations on the functional representation to put it into a suitable form for accurate program analyses and transformation — as accurate as the result of analyzing the imperative program directly. The main contributions of this work are:

- to propose a set of transformations that makes imperative programs amenable to functional programming technology,
- to show that new program analyses and transformations can be developed in a functional framework and applied to imperative languages.

We apply our transformation scheme to programs written in higher-order, imperative language and show its practicality and accuracy in context of a partial evaluator for higher- order, pure, functional programs.

A Self-Applicable Supercompiler

Andrei P. Nemytykh, Victoria A. Pinchuk, and Valentin F. Turchin

A *supercompiler* is a program which can perform a deep transformation of programs using a principle which is similar to *partial evaluation*, and can be referred to as *metacomputation*. Supercompilers that have been in existence up to now (see [21], [20]) were not self-applicable: this is a more difficult problem than self-application of a partial evaluator, because of the more intricate logic of supercompilation. In the present paper we describe the first self-applicable model of a supercompiler and present some tests. Three features distinguish it from the previous models and make self-application possible: (1) The input language is a subset of Refal which we refer to as *flat* Refal. (2) The process of *driving* is performed as a transformation of *pattern-matching graphs*. (3) *Metasystem jumps* are implemented, which allows the supercompiler to avoid interpretation whenever direct computation is possible.

Lockstep Transformations

Kristian Nielsen

We view program transformation as consisting of three distinct parts: external interface, choice of internal specialization technique, and control of transformation. The aim of this work is to achieve a better understanding of the relation between the internal specialization techniques used in existing partial evaluation and related transformation techniques.

A class of *simple functional languages* is proposed for which a class of *lockstep transformations* is defined and proven correct w.r.t. the operational semantics. A simple functional language represents a "typical" language in the field of partial evaluation, and lockstep transformation captures some common elements of partial evaluation-like transformations. The framework is general enough to describe faithfully a wide range of transformations including for example the partial evaluator Similix and Wadler's deforestation.

By recasting different techniques as instances of a single, more general framework their similarities are exposed thus facilitating the exchange of techniques and ideas among different parts of the field. As a concrete example of this we give an efficient implementation of Wadler's deforestation based on the implementation techniques used in Similix.

Multi-Level Lambda-Calculi: an Algebraic Description

Flemming Nielson and Hanne Riis Nielson

Two-level lambda-calculi have been heavily utilised for applications such as partial evaluation, abstract interpretation and code generation. Each of these applications pose different demands on the exact details of the two-level structure and the corresponding inference rules. We therefore formulate a number of existing systems in a common framework. This is done in such a way as to conceal those differences between the systems that are not essential for the multi-level ideas (like whether or not one restricts the domain of the type environment to the free identifiers of the expression) and thereby to reveal the deeper similarities and differences. In their most general guise the multi-level lambda-calculi allow multi-level structures that are not restricted to (possibly finite) linear orders and thereby generalise previous treatments in the literature.

A Comparative Revisitation of Some Program Transformation Techniques

Alberto Pettorossi and Maurizio Proietti

We revisit the main techniques of program transformation which are used in partial evaluation, mixed computation, supercompilation, generalized partial computation, rule-based program derivation, program specialization, compiling control, and the like. We present a methodology which underlines these techniques as a common pattern of reasoning, and it explains the various correspondences which can be established among them. This methodology consists of three steps: i) symbolic computation, ii) search for regularities, and iii) program extraction. We also discuss some control issues which occur in performing these steps.

A Theory of Logic Program Specialization and Generalization for Dealing with Input Data Properties

Alberto Pettorossi and Maurizio Proietti

We address the problem of specializing logic programs w.r.t. the contexts where they are used. We assume that these contexts are specified by means of computable properties of the input data. We describe a general method by which, given a program P, we can derive a specialized program P' such that P and P' are equivalent w.r.t. every input data satisfying a given property. Our method extends the techniques for partial evaluation of logic programs based on Lloyd and Shepherdson's approach, where a context can only be specified by means of a finite set of bindings for the variables of the input goal. In contrast to most program specialization techniques based on partial evaluation, our method may achieve superlinear speedups, and it does so by using a novel generalization technique.

Program Slicing, Differencing, and Merging

Thomas Reps

Not available.

Program Specialization via Program Slicing

Thomas Reps and Todd Turnidge

I will describe the use of program slicing to perform a certain kind of program-specialization operation. The specialization operation that slicing performs is different from the specialization operations performed by algorithms for partial evaluation, supercompilation, bifurcation, and deforestation. In particular, I present an

example in which the specialized program that is created via slicing could not be created as the result of applying partial evaluation, supercompilation, bifurcation, or deforestation to the original unspecialized program. Specialization via slicing also possesses an interesting property that partial evaluation, supercompilation, and bifurcation do not possess: The latter operations are somewhat limited in the sense that they support tailoring of existing software only according to the ways in which parameters of functions and procedures are used in a program. Because parameters to functions and procedures represent the range of usage patterns that the designer of a piece of software has anticipated, partial evaluation, supercompilation, and bifurcation support specialization only in ways that have already been "foreseen" by the software's author. In contrast, the specialization operation that slicing supports permits programs to be specialized in ways that do not have to be anticipated by the author of the original program.

Data Specialization

Erik Ruf and Todd Knoblock

Program staging improves the performance of a program whose inputs vary at different rates by separating it into two phases: an early phase which performs computations dependent only on slowly-changing inputs, and a late phase which performs the remainder of the work given the remaining inputs and the results of the early computations.Staging techniques based on partial evaluation achieve power and generality by reifying these results as code in the form of a dynamically generated residual program. In this talk, we introduce an alternative approach in which the results of early computations are reified as a data structure,allowing both the early and late phases to be generated statically.By avoiding dynamic code manipulation, we trade some power and generality in return for simplicity of implementation and rapid payback. We describe an implementation based on memoization and its use in staging computations in an interactive graphics rendering system .

Data Specialization of Graphics Shading Code

Erik Ruf and Todd Knoblock

Demonstration: Data Specialization for Interactive Graphics We will demonstrate our system for interactive manipulation of shading parameters for three-dimensional rendering. The system takes as input user-defined shading procedures, written in a subset of C, which are then automatically restaged for interactive use. Users of the system typically experiment with multiple values for a single shader parameter while leaving the others constant. Thus, we benefit by generating a late stage

that performs only those computations depending on the parameter being varied; all other values needed by the shader are precomputed and cached by the early stage. The resulting improvement in speed makes it possible to interactively view parameter changes for relatively complex shading models such as procedural solid texturing.

Specialization of Imperative Programs through Analysis of Relational Expressions

Alexander Sakharov

An original analysis method for specialization of imperative programs is described in this paper. This analysis is an inter-procedural data flow method operating on control flow graphs and collecting information about program expressions. It applies to various procedural languages. The set of analyzed formulas includes equivalences between program expressions and constants, linear-ordering inequalities between program expressions and constants, equalities originating from some program assignments, and atomic constituents of controlling expressions of program branches. Analysis is executed by a worklist-based fixpoint algorithm which interprets conditional branches and ignores some impossible paths. This analysis algorithm incorporates a simple inference procedure that utilizes both positive and negative information. The analysis algorithm is shown to be conservative; its asymptotic time complexity is cubic. A polyvariant specialization of imperative programs, that is based on the information collected by the analysis, is also defined at the level of nodes and edges of control flow graphs. The specialization incorporates a further refinement of analysis information through local propagation. Multiple variants are produced by replicating subgraphs whose in-links are limited to one node.

Program Development by Proof Transformation

Helmut Schwichtenberg

Goad's technique of program development by 'pruning' proof trees is explained. As an example we discuss the maximal segment problem of Bates/Constable. The existence proof corresponding to the obvious quadratic algorithm is transformed into a proof yielding a linear algorithm, using additional knowledge about the data (in this case monotonicity of the measure function).

ML Pattern Match Compilation and Partial Evaluation

Peter Sestoft

We derive a compiler for ML-style pattern matches. It is conceptually simple and produces reasonably good compiled matches. The derivation is inspired by the instrumentation and partial evaluation of naive string matchers (Consel and Danvy 1989; Futamura and Nogi 1988). Following that paradigm, we first present a general and naive ML pattern matcher, instrument it to collect and exploit extra information, and show that partial evaluation of the instrumented general matcher with respect to a given match produces an efficient specialized matcher.

We then discard the partial evaluator and show that a match compiler can be obtained just by slightly modifying the instrumented general matcher. The resulting match compiler is interesting in its own right, and naturally detects inexhaustive matches and redundant match rules.

Some Relationships between Partial Evaluation and Meta-Programming

Tim Sheard

Meta-programs are programs that manipulate object-programs. Language tools usually consist of an *object-language* in which the programs being manipulated are expressed, and a *meta-language* that is used to describe the manipulation. One use of meta-programming technology is to *generate* object-programs.

We describe a system of phased computation where meta-programming abilities are built into a language rather than added on as extra-language pre- or post-processors. We describe how such a system can be statically typed, and how it can provide many of the benefits usually associated with macro systems, partial evaluators, and high level specification languages.

Meta-ML

Tim Sheard

Not available.

Self-Applicable Online Partial Evaluation

Michael Sperber

We propose a hybrid approach to partial evaluation to achieve self-application of realistic online partial evaluators. Whereas the offline approach to partial evaluation

leads to efficient specializers and self-application, online partial evaluators perform better specialization at the price of efficiency. Moreover, no online partial evaluator for a realistic language has been successfully self-applied. We propose a binding-time analysis for an online partial evaluator. The analysis distinguishes between static, dynamic, and unknown binding times. Thus, it makes some reduce/residualize decisions offline while leaving others to the specializer. The analysis does not introduce unnecessary generalizations. After some standard binding-time improvements, our partial evaluator successfully self-applies.

Partial Evaluation through Partial Parametrisation

S. Doaitse Swierstra

We have presented a way of defining combinator parsers in such a way that they achieve deterministic parsing and error recovery for LL(1) grammars. Only the basic combinators have to be redefined, whereas all higher level combinators, which have been defined using the low level ones, can be used without having to be adapted.

The program, which makes use of partial parametrisation, implicitly performs a grammar analysis, and optimises itself—provided the implementation of the functional language is fully lazy—into the efficient—more conventional—form, which is normally generated by parser generators.

We further have shown how this method cannot be easily extended to parsers which have been formulated in the so-called monadic form. We indicated however, how by making use of techniques borrowed from the attribute grammar area, program transformations can be performed which enable the aforementioned optimising form of execution.

Hylomorphism in Triplet Form: An Ideal Internal Form for Functional Program Optimization

Akihiko Takano (joint work with Erik Meijer)

Hylomorphism is originally defined as a composition of a catamorphism (a generalized fold) and an anamorphism (a generalized unfold). In our previous work, we introduced a triplet form representation of hylos and showed how the essence of the shortcut deforestation could be captured in more general setting. In this talk, we provide a brief and intuitive view for the three components of hylo, and demonstrate how it facilitates automatic program optimization.

Metacomputation: Metasystem Transitions plus Supercompilation

Valentin F. Turchin

Metacomputation is a computation which involves *metasystem transitions(MST* for short) from a computing machine *M* to a metamachine *M'* which controls, analyzes and imitates the work of *M*. Semantics-based program transformation, such as partial evaluation and supercompilation *(SCP)*, is metacomputation. Metasystem transitions may be repeated, as when a program transformer gets transformed itself. In this manner MST hierarchies of any height can be formed.
The paper reviews one strain of research which was started in Russia in the late 1960s - early 1970s and became known for the development of supercompilation as a distinct method of program transformation. After a brief description of the history of this research line, the paper concentrates on those results and problems where supercompilation is combined with repeated metasystem transitions.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Dept. of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 København Ø, May 1994.

[2] D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation*, Amsterdam, 1988. North-Holland.

[3] W.-N. Chin. Towards an automated tupling strategy. In Schmidt [17], pages 119–132.

[4] C. Consel, editor. *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '92*, San Francisco, CA, June 1992. Yale University. Report YALEU/DCS/RR-909.

[5] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, Jan. 1993. ACM Press.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control flow graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[7] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, Dagstuhl, Germany, Feb. 1996. Springer Verlag, Heidelberg.

[8] A. P. Ershov, D. Bjørner, Y. Futamura, K. Furukawa, A. Haraldsson, and W. Scherlis, editors. *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987 (New Generation Computing, vol. 6, nos. 2,3)*. Ohmsha Ltd. and Springer-Verlag, 1988.

[9] J. Gallagher. Specialization of logic programs. In Schmidt [17], pages 88–98.

[10] P. Hudak and N. D. Jones, editors. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '91*, New Haven, CT, June 1991. ACM. SIGPLAN Notices 26(9).

[11] Journal of Functional Programming 3(3), special issue on partial evaluation, July 1993. Neil D. Jones, editor.

[12] Journal of Logic Programming 16 (1,2), special issue on partial deduction, 1993. Jan Komorowski, editor.

[13] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[14] Lisp and Symbolic Computation 8 (3), special issue on partial evaluation, 1995. Peter Sestoft and Harald Søndergaard, editors.

[15] A. A. Letichevsky, J. V. Kapitonova, and S. V. Konozenko. Computations in APS. *Theoretical Comput. Sci.*, 119:145–171, 1993.

[16] W. Scherlis, editor. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, La Jolla, CA, June 1995. ACM Press.

[17] D. Schmidt, editor. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '93*, Copenhagen, Denmark, June 1993. ACM Press.

[18] P. Sestoft. Bibliography on partial evaluation. Available through URL ftp://ftp.diku.dk/pub/diku/dists/jones-book/partial-eval.bib.Z.

[19] P. Sestoft and H. Søndergaard, editors. *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '94*, Orlando, Fla., June 1994. ACM.

[20] V. F. Turchin. The concept of a supercompiler. *ACM Trans. Prog. Lang. Syst.*, 8(3):292–325, July 1986.

[21] V. F. Turchin, R. M. Nirenberg, and D. V. Turchin. Experiments with a supercompiler. In *1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pennsylvania*, pages 47–55. ACM, 1982.

[22] D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 297–310, Portland, OG, Jan. 1994. ACM Press.