J. Functional Programming **2** (2): 227–231, April 1992

A review of the FPCA '91 proceedings

THOMAS JOHNSSON
Department of Computer Science, Chalmers University of Technology, Sweden

---

John Hughes (Ed); *Functional Programming Languages and Computer Architecture*. Proceedings of the fifth conference (Cambridge, MA, 28–30 August 1991). Volume 523 of *Lecture Notes in Computer Science*, Springer-Verlag. 666 pp.

These proceedings contain the 30 papers of the fifth biennial conference on functional programming and computer architecture (FPCA), which was held at Harvard University, Cambridge, Massachusetts, from 28 to 30 August 1991. The papers represent the hottest and best in functional programming. Some topics were particularly prominent: types, partial evaluation, compiling techniques for sequential and parallel architectures, and manipulating and updating states and arrays in functional languages. Below follows a brief description of the general themes and the individual papers of the conference. Numbers refer to the table of contents at the end.

Types continue to be a hot topic in functional programming research, and there are several papers on both theoretical and practical aspects of types. There is also a growing interest in type systems with overloading, such as that in the functional language Haskell; the first two papers are on this topic.

Nipkow and Snelting (1) present a type inference algorithm for a language with type classes and overloading, where the ordinary unification algorithm in the Hindley–Milner system is replaced by order-sorted unification from the world of many-sorted algebras.

Volpano and Smith (2) explore the computational consequences of the increased expressiveness of Haskell-like type systems with overloading; some variants are undecidable.

Thatte (3) describes an extension of the Hingley–Milner type inference system to allow isomorphism declarations, for example between rectangular and polar representations of complex numbers. The user provides conversion functions, which the compiler applies where needed. His type inference algorithm is based on equational unification.

On the practical side of typing, Aditya and Nikhil (19) attempt to reconcile polymorphic type checking with an incremental programming environment. On the one hand one wants to develop (and edit) the program incrementally, on the other hand the standard polymorphic type-inference algorithm examines the whole program at once. They present a type-inference algorithm which produces the same

typings as obtained via the standard 'batch' algorithm, but produces them incrementally.

Leroy and Mauny (20) explore the combination of dynamic typing, i.e. type checking at run time, with the static polymorphic type discipline. They present two different extensions to ML with dynamic objects.

Aiken and Murphy (21) have developed efficient algorithms for operations on regular tree expressions. The authors use regular tree expressions for representing types in a type inference system for FL.

On the theory of types, Lillie and Harrison (13) present a model of recursive types and subtypes based on a metric space of projections.

Types and non-standard type inference also play a key rôle in papers on abstract interpretation and strictness analysis (16, 18), binding time analysis (22), dealing safely and efficiently with states in a functional language (10, 11), and low-level implementation details (30); more on these below.

Frandsen and Sturtivant (14) provide a complexity theoretic framework for studying the efficiency of functional language implementations. They study the pure lambda calculus, arguing that any feature in a functional language can be encoded into it. Surprisingly, combinator- and supercombinator-based implementations are exponentially inefficient!

The purpose of Mairson's paper (15) is to clarify some aspects of Wadler's 'Theorems for Free', and to provide a complementary view of them. He develops a constructive framework for proving equalities about programs, and shows how to derive the free theorems in a purely syntactic way. As a result of the clarification he uncovers a hidden cost of the free theorems: there is a genuine need for structural induction.

The functional programming process is represented by the paper by Meijer *et al.* (7). Using a categorical framework, they develop a Squiggol-style calculus for lazy functional programming based on recursion operators associated with data-type definitions.

There are now two radically different approaches to strictness analysis: the conventional approach based on abstract interpretation, and the novel approach based on non-standard type inference.

Leung and Mishra (16) present a strictness analyser for a higher-order lazy language which discovers both simple strictness and hyper-strictness. It is based on non-standard subtype inference and does not require any fixpoint iteration; hence, their analyser should be more amenable to practical use.

Jensen (17) presents a formal framework for comparing the two approaches. He shows that strictness analysis by abstract interpretation and by type inference are equally powerful techniques.

On the subject of abstract interpretation in general, Baraki (18) develops a method for computing safe approximations to abstract functions of polymorphic higher-order functions; he makes extensive use of category theory.

Partial evaluation has clearly become a part of the main stream of functional programming research (as is also evident by the Partial Evaluation and Program

Manipulation (PEPM) conference earlier this year). There are five papers on or relating to this topic in these proceedings.

Launchbury (8) describes partial evaluation and self-application in a strongly typed context. He reports on the first successful attempt to write such a partial evaluator, and describes some of the problems encountered, notably a double encoding problem which, if not solved, threatens to make the partial evaluator grossly inefficient.

Weise *et al.* (9) have constructed the first fully automatic online partial evaluator (i.e. using both the input program and the actual static data values) for an untyped functional language. Their work emphasizes program optimization (while essentially ignoring self application), and their specializer produces highly specialized programs.

Three papers deal with program analysis techniques aiding partial evaluation. Binding time analysis determines which values are static, i.e. known at partial evaluation time, and which ones are dynamic. Henglein (22) describes an almost linear higher-order binding time analysis based on type inference. Holst's finiteness analysis (23) answers the question of whether a program will go through only a finite number of program states, a necessary and sufficient condition for partial evaluation to terminate. The analysis designed by Consel and Danvy (24) aims at discovering how to change the control flow of a program to make the binding time analysis yield better results, i.e. to get bigger static expressions.

A decade or so ago it was thought that to get competitive performance out of functional languages, special architectures had to be designed. Over the years, however, the von Neumann architecture has proved itself surprisingly competitive also for these languages – performance can be gained by clever compilation rather than requiring radically different architectures. Pure dataflow architectures have evolved into general von Neumann-like multithreaded architectures which separate control flow and dataflow. Two papers, Schauser *et al.* (4) and Traub (5), describe different aspects of compiling the lenient language Id for such multithreaded architectures. Both treat multithreaded execution as a compilation problem, thereby getting performance gains that hardware mechanisms alone cannot provide.

Maranget (6) compiles a parallel version of the 'standard' G-machine for a conventional shared-memory multiprocessor, the Sequent Balance. He obtains a performance competitive with previously compiled graph reducers, which are based on refined versions of the standard G-machine: the $\langle v, G \rangle$-machine and the Spineless Tagless G-machine on the GRIP multiprocessor.

The paper by Smetsers *et al.* (28) describes compilation of the lazy language Concurrent Clean into very efficient code for a conventional uniprocessor (the MC 68020), via an abstract machine called the ABC machine. Strictness analysis, and/or programmer-supplied strictness annotations, play an essential rôle in obtaining the high speed (even faster than C for some of the benchmarks).

The paper by Peyton Jones and Launchbury (30) can also be said to be about compilation. They describe a systematic way to deal with the boxing and unboxing that an implementation of a lazy language must do to treat numbers (and similar types) properly, using a sequence of program transformations in the compiler.

Despite the title of this conference, there is only one paper on computer

architecture proper. Chiueh (25) observes that heap-intensive programs have a higher than usual cache-miss ratio. His paper describes an elegant architectural technique to do some garbage collection work on the cells in the cache. On a cache miss, the processor switches to a GC thread which performs a (partial) reference-count garbage collection (cells have reference counts only in the cache), while waiting for the data to arrive in the cache. These cycles would otherwise have been wasted just waiting. New cells can often be allocated from those in the cache just released by the GC without going to the main memory, thereby reducing the cache-miss ratio.

Despite all their qualities, there are things that pure functional languages are not good at; dealing with states and updating is one of them. Four papers describe approaches to remedying this deficiency.

Swarup *et al.* (10) add imperative features to a functional language while still retaining referential transparency; expressions are still side-effect-free.

Wakeling and Runciman (11) have added linear (i.e. single-threaded) types to an implementation of Lazy ML. They report that linear types can be a mixed blessing: programming can become more difficult because of the linearity constraint, and the performance can also be disappointing.

Fradet (12) describes an analysis which detects single-threading syntactically, using continuations as a way of formalizing the evaluation strategy.

Barth *et al.* (26) take a more blunt approach: they extend the non-strict parallel functional language Id with M-structures, which are mutable data structures with built-in synchronization. The language then loses referential transparency, and it is the programmer's responsibility to get it right. Yet surprisingly they argue that such programs can be more declarative and parallel than their purely functional counterpart!

Efficient compilation of list comprehensions and optimization of database queries have a lot in common. Heytens and Nikhil (27) explore the use of list comprehensions as a database query language, and describe the optimization and translation of list comprehensions in AGNA, a parallel persistent object system.

The overall goal of Hannan's work (29) is to develop techniques for mechanically constructing provably correct and efficient implementations of programming languages based on operational semantics for languages. This paper addresses the problem of transforming abstract machines (obtained from operational semantics) into more concrete ones, that are more amenable to efficient implementation.

Table of contents

5. Multi-thread Code Generation for Data Flow Architectures from Non-strict Programs (K. R. Traub).
6. GAML: a Parallel Implementation of Lazy ML (L. Maranget).
7. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire (E. Meijer, M. Fokkinga, R. Paterson).
8. A Strongly-Typed Self-Applicable Partial Evaluator (J. Launchbury).
9. Automatic Online Parallel Evaluation (D. Weise, R. Conybeare, E. Ruf, S. Seligman).
10. Assignments for Applicative Languages (V. Swarup, U. S. Reddy, E. Ireland).
11. Linearity and Laziness (D. Wakeling, C. Runciman).
12. Syntactic Detection of Single-Threading using Continuations (P. Fradet).
13. A Projection Model of Types (D. J. Lillic, P. G. Harrison).
14. What is an Efficient Implementation of the λ-calculus? (G. S. Frandsen, C. Sturtivant).
15. Outline of a Proof Theory of Parametricity (H. G. Mairson).
16. Reasoning about Simple and Exhaustive Demand in Higher-Order Lazy Languages (A. Leung, P. Mishra).
17. Strictness Analysis in Logical Form (T. P. Jensen).
18. A Note on Abstract Interpretation of Polymorphic Functions (G. Baraki).
19. Incremental Polymorphism (S. Aditya, R. S. Nikhil).
20. Dynamics in ML (X. Leroy, M. Mauny).
21. Implementing Regular Tree Expressions (A. Aiken, B. R. Murphy).
22. Efficient Type Inference for Higher-Order Binding-Time Analysis (F. Henglein).
23. Finiteness Analysis (C. Kehler Holst).
24. For a Better Support of Static Data Flow (C. Consel, O. Danvy).
25. An Architectural Technique for Cache-level Garbage Collection (T. Chiueh).
26. M-Structures: Extending a Parallel, Non-strict, Functional Language with State (P. S. Barth, R. S. Nikhil, Arvind).
27. List Comprehensions in Agna, A Parallel Persistent Object System (M. L. Heytens, R. S. Nikhil).
28. Generating Efficient Code for Lazy Functional Languages (S. Smetsers, E. Nocker, J. van Groningen, R. Plasmeijer).
29. Making Abstract Machines Less Abstract (J. Hannan).
30. Unboxed Values as First Class Citizens (S. L. Peyton Jones, J. Launchbury).