JFP **16** (4&5): 373–374, 2006. © 2006 Cambridge University Press
doi:10.1017/S0956796806005958 Printed in the United Kingdom

Introduction

KATHLEEN FISHER
AT&T Labs Research, 180 Park Ave, Florham Park, NJ 07932
(*email:* kfisher@research.att.com)

The *Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP)* took place on September 19–21 2004 in Snowbird, Utah. The scope of ICFP includes all languages that encourage programming with functions, with topics ranging from principles to practice, foundations to features, and abstractions to applications. The program committee, which I chaired, selected 21 papers from 80 submissions for presentation at the conference. Afterwards, the program committee invited the authors of nine papers to submit extended versions for this special issue of JFP. The seven papers that appear in this volume were reviewed, revised, and accepted following standard *Journal of Functional Programming* procedures. Reflecting the scope of the conference, the papers cover the theoretical foundations, implementation, design, programming techniques, and applications of functional languages.

The special issue opens with “Mondadic Regions” by Matthew Fluet and Greg Morrisett, in which the authors show how a language with regions can be encoded faithfully in the familiar polymorphism of System F rather than in a type-and-effect system. The encoding adapts ideas from the ST monad of Launchbury and Peyton Jones to manage regions in the target language.

In the second paper, “Sound and Complete Models of Contracts”, Matthias Blume and David McAllester present a denotational semantics for contracts, which are dynamic assertions about the properties that must hold at function boundaries. This semantics allows the authors to show that the Findler–Felleisen algorithm for contract checking is sound and complete under reasonable assumptions. It also highlights places where one’s intuition about contract checking may or may not coincide with a sound algorithm.

In the third paper, “Static Analysis for Path Correctness of XML Queries”, Dario Colazzo, Giorgio Ghelli, Paolo Manghi and Carlo Sartiani present a type system for detecting portions of XML queries that will never contribute to the results of those queries. In the process, the authors revisit some fundamental questions about the basis for modern type systems. This work constitutes a key step towards a complete treatment of static typing for XML query languages.

The special issue continues with “Making a Fast Curry: Push/Enter vs. Eval/Apply for Higher-order Languages” by Simon Marlow and Simon Peyton Jones. In the paper, the authors close the book on the question of whether the “eval/apply” or “push/enter” evaluation strategy is superior for efficiently compiling partial applications of unknown functions in pure languages. On a wide range of Haskell programs, the two strategies have very similar performance characteristics, making “eval/apply” the clear winner as it is significantly simpler to implement.

In "Multi-return Function Call", Olin Shivers and David Fisher extend the notion of function call to allow functions to have multiple return points. The authors provide an unusually broad study of this feature, including a formal semantics, polymorphic and monomorphic type systems, a report on experiences with an implementation, program transformations, and intriguing examples of the use of the feature.

Next, Robert Bruce Findler and Matthew Flatt present a PLT Scheme-based functional system for designing and presenting slides in "Slideshow: Functional Presentations". The paper discusses design at multiple levels: slides, talks composed of slides, and the language abstractions needed for building such artifacts. It demonstrates that functional languages are ideally suited to this task.

Finally, the special issue concludes with "Generics for the Masses" by Ralf Hinze. In the paper, the author shows how to use standard Haskell 98 type classes to write *generic* functions, i.e. functions that behave in type-specific ways when applied to different datatypes. Much recent work has focused on language extensions to support generic programming. This paper shows that a surprising amount of generic programming can be done without such extensions.

I would like to thank my colleagues who served on the *ICFP 2004* program committee and those who participated in the reviewing process for JFP. Their efforts contributed greatly to the quality and timeliness of this special issue.