JFP **21** (4 & 5): 331–332, 2011.
doi:10.1017/S0956796811000190 First published online 12 August 2011

Special Issue Dedicated to ICFP 2009 Editorial

The 14th ACM SIGPLAN International Conference on Functional Programming (ICFP) took place on August 31–September 2, 2009 in Edinburgh, Scotland; Andrew Tolmach chaired the program committee. Following the conference, the authors of selected papers were invited to submit extended versions for this special issue of JFP. After review and revision, four papers were accepted for inclusion in this volume. Each paper contains substantial new material beyond the original conference version. The papers are representative of the wide range of topics and methodology that characterize ICFP.

A number of popular recent innovations in Haskell type systems, such as Generalized Algebraic Data Types and type families, introduce local type equality constraints. These constraints pose substantial problems for type inference, including loss of principal types and very complex generalization rules. In *OutsideIn(X): Modular type inference with local assumptions*, Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann synthesize the results presented in their ICFP 2009 paper with material from several earlier conference papers to propose a comprehensive approach to these problems. OutsideIn(X) is a type system for languages with local constraints, parameterized by an underlying constraint system X. It is similar to the well-known HM(X) system, though not a conservative extension, because (somewhat controversially) it does not support automatic generalization of local `let` bindings. The paper gives a general inference algorithm for OutsideIn(X), parameterized by a solver for X. It also describes a solver for X instantiated by the combination of GADTs, type families, and type classes, which is distributed as part of the current implementation of the Glasgow Haskell Compiler.

It is well known that nondeterministic computations can be represented using monads of lists or trees. These encodings can be used to emulate some aspects of functional logic languages, such as Curry, within a deterministic language like Haskell. But in *Purely Functional Lazy Nondeterministic Programming*, Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan show that capturing the full behavior of lazy (i.e., both nonstrict and shared) nondeterminism in Haskell requires a more sophisticated encoding. They describe a Haskell library that implements the desired features, which are characterized by a set of algebraic laws. The paper is interesting both as an exposition of a useful Haskell programming technique, and as a bridge between the functional and functional-logic programming worlds.

Arrows are well established as a useful abstraction for functional reactive programming (FRP). Arrows are useful in part because they are more general than monads, but in some situations they may be *too* general to capture the interesting properties of a system. In *Causal Commutative Arrows*, Hai Liu, Eric Cheng, and

Paul Hudak describe a refinement of arrows, incorporating a commutativity law and an initialization operator, that gives a close fit to the Yampa FRP domain-specific language. Causal Commutative Arrow expressions can be put into a normal form that supports very efficient evaluation, offering large performance improvements over conventional arrow implementations for Yampa. This normalization technique should be applicable to other systems based on dataflow or stream computation.

Some recent functional languages combine parametric polymorphic types, which give powerful guarantees about type abstraction, with run-time type inspection mechanisms such as `typecase`, which can potentially break abstraction boundaries. To address this tension, several researchers have proposed using dynamically generated type names to represent abstract types during run-time inspection. But does this technique actually recover a useful notion of parametricity? In *Non-parametric Parametricity*, Georg Neis, Derek Dreyer, and Andreas Rossberg give an affirmative answer for a language including a type-safe `cast` operator. Along the way, they develop general notions of what it means for terms in a non-parametric language to behave parametrically, including an interesting concept of parametricity polarity. Their proof uses step-indexed Kripke logical relations; while the main body of the paper is highly technical, this is preceded by a very accessible summary of the main ideas behind the technique.

We cordially thank the authors and the referees for their work in producing and reviewing these papers, especially under the strict time limits imposed by a special issue. We hope that you, the reader, will enjoy the fruits of their labors.

Andrew Tolmach
Department of Computer Science
Portland State University
apt@cs.pdx.edu

Xavier Leroy
INRIA Paris-Rocquencourt
xavier.leroy@inria.fr

Special Issue Editors