JFP **26**, e20, 3 pages, 2016. © Cambridge University Press 2016
doi:10.1017/S0956796816000228

Special issue dedicated to ICFP 2014
Editorial

The 19th ACM SIGPLAN International Conference on Functional Programming (ICFP) took place on September 1–3, 2014 in Gothenburg, Sweden. After the conference, the programme committee, chaired by Manuel Chakravarty, selected several outstanding papers and invited their authors to submit to this special issue of JFP. We acted as editors for these submissions. This issue includes the six accepted papers, each of which provides substantial new material beyond the original conference version. The selected papers demonstrate both the quality and the breadth of the conference, with a strong emphasis on types and their applications, and ranging from compilation methods through contract verification to homotopy type theory.

In *Romeo: A System For More Flexible Binding-Safe Programming*, Stansifer and Wand present a programming language, Romeo, that allows the user to define data types for syntax trees with complex binding structure and functions for manipulating such trees. These functions guarantee that binding structures are respected, so that no name can escape its scope, i.e., that alpha-equivalent programs evaluate to alpha-equivalent results. The paper presents a type system that functions as a language for specifying binding structure that is both simple and expressive, inspired by attribute grammars; it goes beyond earlier work in its combination of complex binding structures with flexible programming. The paper's main contribution is an extension of Herman's system for binding-safety in a pattern-matching macro system to cover macros defined by procedures, and thus general meta-programming for terms with bindings.

In *Higher Order Symbolic Execution for Contract Verification and Refutation*, Nguyễn, Tobin-Hochstadt, and Van Horn endow symbolic execution with a new form of higher order values, enabling verification of safety properties of functional programs. In particular, they show how to apply the approach to the static verification of behavioural software contracts in a functional language (Racket). They argue that higher order symbolic execution and behavioural contracts are mutually beneficial. The result is a system for higher order programming that combines effective bug finding and a gradual spread of verified contracts. A key aspect of the approach is the use of contracts to delay higher order checks so that failures always occur with first-order witnesses. A search for counter-examples relies on the fact that, even in a higher order setting, there are relatively few ways in which inputs trigger program errors. This enables surprisingly strong guarantees (soundness and relative completeness given an SMT solver that is complete for base types). A substantial benchmarking effort indicates that the approach is competitive with existing tools ranging from type systems to model checkers, even on their own benchmarks. Case studies on small interactive video games that use first-class and dependent contracts demonstrate that the elimination of dynamic contract monitoring can yield rather large speedups, and indeed that this speedup can be the key to making a game feasible in practice.

In *Gradual Type-and-Effect Systems*, Bañados Schwerter, Garcia, and Tanter develop a theory of gradual effect checking, using the framework of abstract interpretation. This theory is motivated by the observation that effect systems have not been widely used, because in practice it seems difficult to move from a system where effects are implicit and unrestricted to one with a fully static effect discipline. The use of abstract interpretation not only allows the fundamental differences between gradual typing and gradual effect systems to be made clear, but also guides the specification of key notions such as the meaning of unknown effects in terms of abstraction and concretization operations. The authors take as their starting point the generic effect system of Marino and Millstein, which they extend into a system for gradual effects. A concrete instantiation permits the gradual checking of exceptions and provides a compact, easily understood demonstration of the approach. This paper expands considerably on the original conference version, by adding a new operational semantics that does not require values to be tagged, and by showing how gradual effect checking and gradual typing can be combined.

In *Safe Zero-cost Coercions for Haskell*, Breitner, Eisenberg, Peyton Jones, and Weirich show how to support generative type abstraction—a useful feature in many functional languages—whilst also providing the ability to coerce values at no runtime cost between an abstract type and its underlying representation. Previous work by Weirich *et al.* in POPL 2011 had proposed a way of attacking this problem through the introduction of *roles*, which tracked the distinction between “nominal” and “representational” type equality. However, that approach involved a complicated extension to the kind language of Haskell. The present paper presents a much simpler approach in which role information is confined to the parameters of data types instead of invading the kind system. The resulting design loses something in expressiveness, but more than makes up for it in practical implementability. In addition to proving type soundness and showing how to support role inference and simplification of coercion constraints, the authors describe the integration of their design into the GHC compiler.

In *Homotopical Patch Theory*, Angiuli, Morehouse, Licata, and Harper consider a programming application of higher inductive types, a new class of datatypes that arises in homotopy type theory. Homotopy type theory extends Martin-Löf’s intensional type theory with higher inductive types and Voevodsky’s univalence axiom, features motivated by a correspondence between type theory, homotopy theory, and higher category theory. Higher inductive types are specified by constructors not only for points but also for paths between points, paths between paths and so on; these have primarily been used to model spaces in computer-checked proofs of theorems in homotopy theory. However, there has been a dearth of programming examples, rendering homotopy type theory a challenging topic for non-experts. This paper fills that gap by using higher inductive types to model *patch theories*, which are formal models of version control systems. The paper considers a sequence of patch theories, including two that did not appear in the original conference version of the paper, culminating in a patch theory of text files. So for all those readers who have been wondering what homotopy type theory is about, this programming-oriented view could be a good place to start.

In *Eliminating Dependent Pattern Matching Without K*, Cockx, Devriese, and Piessens define a translation from a language with dependent pattern matching on inductive families to a language with only datatype eliminators (induction principles). Such a translation is an important piece of the implementation of a dependently typed programming language, because it justifies a usable source syntax in terms of the basic building blocks that are used to study the metatheory of a language. In his thesis, Conor McBride established such a translation, which was subsequently refined with collaborators and has been used as the basis of the implementation of Epigram 2 and of the Equations library in Coq. However, McBride's translation depended on uniqueness of identity proofs (the principle that any two proofs of $x = y$ are themselves equal), also known as the "K" axiom. Recent work on homotopy type theory has introduced axioms that contradict K, thus motivating the need for a translation of dependent pattern matching that does not depend on K. In this paper, Cockx *et al.* fill this gap, developing a criterion for dependent pattern matching which ensures that it can be implemented without the K axiom. They have implemented their criterion in Agda, and demonstrated that it is sufficient to handle significant examples drawn from homotopy type theory.

We thank the authors and reviewers for their efforts in producing and reviewing these papers within strict time limits. We also gratefully acknowledge the support of the JFP editors-in-chief and editorial office, as well as Manuel Chakravarty and the ICFP 2014 programme committee for their help in selecting the invited papers.

Derek Dreyer
Foundations of Programming Group
Max Planck Institute for Software Systems (MPI-SWS)
`dreyer@mpi-sws.org`

Mary Sheeran
Functional Programming Group
Computer Science and Engineering Department
Chalmers University of Technology
`ms@chalmers.se`