More information about this series at http://www.springer.com/series/7407
Preface

This volume contains the proceedings of the 13th International Symposium on Functional and Logic Programming – FLOPS 2016 – held in Kochi, Japan, March 4–6, 2016.

FLOPS brings together practitioners, researchers, and implementers of declarative programming, to discuss mutually interesting results and common problems: theoretical advances, their implementations in language systems and tools, and applications of these systems in practice. The scope includes all aspects of the design, semantics, theory, applications, implementations, and teaching of declarative programming. FLOPS specifically aims to promote cross-fertilization between theory and practice and among different styles of declarative programming.

FLOPS 2016 put a particular stress on the connections between theory and practice. This stress was reflected in the composition of the Program Committee, in the call for submissions and, ultimately, in the program of the symposium.

The call for papers attracted 36 submissions, of which the Program Committee, after careful and thorough discussions, accepted 14. The accepted papers cover not just functional and logic programming but also program transformation and re-writing, and extracting programs from proofs of their correctness. The invited speakers, Kazunori Ueda and Atze Dijkstra, reflected on the lessons of two projects (one of which was a national, Japanese project), with declarative programming at their center. In addition to the invited talks and contributed papers, the symposium program included, for the first time, tutorials and a poster session. The tutorials on “Attribute Grammars”, “Agda”, and “Programming in Picat” were presented, respectively, by Atze Dijkstra, Andreas Abel, and Neng-Fa Zhou. These tutorials were designed to complement the invited talk with in-depth expositions.

This year we initiated an award for the best paper submitted to the symposium. We were delighted to announce that the award for FLOPS 2016 went to Arthur Blot, Pierre-Evariste Dagand, and Julia Lawall for their article entitled “From Sets to Bits in Coq.”

Putting together FLOPS 2016 has been a team effort. First of all, we would like to thank the authors of the submitted papers and the presenters of the invited talks and the tutorials. Without the Program Committee (PC) we would have had no program either, and we are very grateful to the PC members for their hard work. Supporting the PC were a number of additional reviewers, and we and the PC would like to acknowledge their contribution. The reviews were unusually detailed and helpful. An author of one rejected paper wrote to us, not to complain but to praise the reviews of his submission. We are greatly indebted to the general chair, Yukiyoshi Kameyama for his advice, encouragement, and support throughout the process and taking on many administrative chores. The local chair, Kiminori Matsuzaki, and the local Organizing Committee were invaluable in setting up the conference and making sure everything ran smoothly.
Finally, we would like to thank our sponsor, the Japan Society for Software Science and Technology (JSSST) SIGPPL, for their continuing support. We acknowledge the cooperation of ACM SIGPLAN, the Asian Association for Foundation of Software (AAFS), and the Association for Logic Programming (ALP).

January 2016

Oleg Kiselyov
Andy King
Organization

Program Chairs

Andy King
University of Kent, UK

Oleg Kiselyov
Tohoku University, Japan

General Chair

Yukiyoshi Kameyama
University of Tsukuba, Japan

Local Chair

Kiminori Matsuzaki
Kochi University of Technology, Japan

Program Committee

Andreas Abel
Gothenburg University, Sweden

Lindsay Errington
USA

Makoto Hamana
Gunma University, Japan

Michael Hanus
CAU Kiel, Germany

Jacob Howe
City University London, UK

Makoto Kanazawa
National Institute of Informatics, Japan

Hsiang-Shang Ko
National Institute of Informatics, Japan

Julia Lawall
Inria-Whisper, France

Andres Löh
Well-Typed LLP, UK

Anil Madhavapeddy
Cambridge University, UK

Jeff Polakow
USA

Marc Pouzet
École normale supérieure, France

Vítor Santos Costa
Universidade do Porto, Portugal

Tom Schrijvers
KU Leuven, Belgium

Zoltan Somogyi
Australia

Alwen Tiu
Nanyang Technological University, Singapore

Sam Tobin-Hochstadt
Indiana University, USA

Hongwei Xi
Boston University, USA

Neng-Fa Zhou
CUNY Brooklyn College and Graduate Center, USA
## External Reviewers

<table>
<thead>
<tr>
<th>External Reviewer</th>
<th>External Reviewer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guillaume Baudart</td>
<td>Anthony Widjaja Lin</td>
</tr>
<tr>
<td>Timothy Bourke</td>
<td>Lunjin Lu</td>
</tr>
<tr>
<td>Benoit Desouter</td>
<td>Kazutaka Matsuda</td>
</tr>
<tr>
<td>Sandra Dylus</td>
<td>Shin-Cheng Mu</td>
</tr>
<tr>
<td>Adrien Guatto</td>
<td>Björn Peemöller</td>
</tr>
<tr>
<td>Geoff Hulette</td>
<td>Lutz Strassburger</td>
</tr>
<tr>
<td>Kazuhiro Inaba</td>
<td>Jan Rasmus Tikovský</td>
</tr>
<tr>
<td>Georgios Karachalias</td>
<td>Paolo Torrini</td>
</tr>
</tbody>
</table>
UHC: Coping with Compiler Complexity  
(Keynote Abstract)

Atze Dijkstra

Utrecht University, Department of Information and Computing Sciences  
atze@uu.nl

Abstract. Programming language design may be difficult, but by now doing an actual design of a language feature is an often repeated and relatively well understood process involving known ingredients: construct a minimal language incorporating the desired feature, define (operational) semantics, a declarative type system, an algorithmic type system, and a prototype implementation. Obviously, this is a gross simplification ignoring the mathematic craftsmanship involved, and not always all of the above ingredients are being dealt with. Still, this is the raw material and mechanics of design found in many conference proceedings on programming languages and their design.

In contrast however, how to implement a designed programming language feature and incorporating it into an existing programming language seems to be less well exposed. A sketch of an implementation and its related issues often is given but of the actual code and its details often at best is summarized by a footnote referring to the (repository of the) implementation. Of course publications exist which specifically address an implementation itself [8] but the size limited nature of a publication forces such descriptions of implementations to narrow down to a limited set of language features and often simplification of the implementation itself is required to obtain clarity and compactness. With the risk of oversimplification we conjecture that design and implementation of individual programming language features is well understood but it is less clear how the implementation of the combination of such individual features can be done in a systematic and predictable way, or, in other words: how do we deal with the complexity arising out of programming language feature implementation both in isolation and combination (as it occurs in a compiler)?

Here we will deal with this issue of compiler complexity by looking at the approaches taken for UHC (Utrecht Haskell Compiler) [4, 3, 2]. UHC is a Haskell compiler intended to be experimented with, both in terms of the use of tools for construction and in terms of being a platform for (relatively) easy experimentation with language features and their implementation. In particular, within UHC two more general sources of complexity are being dealt with:

2. Combination of implementation of individual programming language features into a full compiler. The complexity lies in the interaction between language features.
Over the lifespan of the UHC project the following approaches and solutions have been explored:

- \textit{addressing complexity source 1)} The use of attribute grammars for the specification of programming language feature implementations. The UUAG \cite{SwierstraMiddelkoopBransen:2013} is mostly used for the implementation of UHC. The AG formalism is further explored into various directions, for example Ruler \textit{(also addressing complexity source 2)} \cite{DijkstraSwierstra:2006} specifically targets type system specification, Viera \cite{Viera:2013} embeds the tools for description of programming language implementation (i.e. parser, attribute grammar) as a DSL into Haskell (GHC \cite{GHC}).

- \textit{addressing complexity source 2)} Partitioning the full implementation description into smaller fragments as belonging to a particular language feature; these are then combined together when constructing a compiler using Shuffle \cite{Dijkstra:2005}.

- \textit{addressing complexity source 1)} The use of CHR (Constraint Handling Rules) \cite{Fruehwirth:2009} for type related computations involving backtracking.

The above approaches vary in their success. We will discuss our experience with these approaches, in particular what more formal counterparts can be implemented with our tools, small examples of this looks, and what (in retrospect) did or did not work.

References

1. Dijkstra, A.: Shuffle. \url{http://foswiki.cs.uu.nl/foswiki/Ehc/Shuffle}
7. GHC Team: The Glasgow Haskell Compiler. \url{https://www.haskell.org/ghc/}
Contents

Logic/Constraint Programming and Concurrency: The Hard-Won Lessons of the Fifth Generation Computer Project ................................. 1
  Kazunori Ueda

From Sets to Bits in Coq ................................................................. 12
  Arthur Blot, Pierre-Évariste Dagand, and Julia Lawall

From Proposition to Program: Embedding the Refinement Calculus in Coq ................................................................. 29
  Wouter Swierstra and Joao Alpuim

The Boolean Constraint Solver of SWI-Prolog (System Description) .... 45
  Markus Triska

Probabilistic Inference by Program Transformation in Hakaru
(System Description) .................................................................. 62
  Praveen Narayanan, Jacques Carette, Wren Romano,
  Chung-chieh Shan, and Robert Zinkov

An Interaction Net Encoding of Gödel’s System T: Declarative Pearl ...... 80
  Ian Mackie and Shinya Sato

Space-Efficient Planar Acyclicity Constraints: A Declarative Pearl ....... 94
  Taus Brock-Nannestad

Executable Relational Specifications of Polymorphic Type Systems Using
Prolog .............................................................................................. 109
  Ki Yung Ahn and Andrea Vezzosi

Proof Relevant Corecursive Resolution ............................................. 126
  Peng Fu, Ekaterina Komendantskaya, Tom Schrijvers,
  and Andrew Pond

A Coq Library for Internal Verification of Running-Times ..................... 144
  Jay McCarthy, Burke Fetscher, Max New, Daniel Feltey,
  and Robert Bruce Findler

A Transformational Approach to Parametric Accumulated-Cost Static
Profiling ......................................................................................... 163
  R. Haemmerlé, P. López-García, U. Liqat, M. Klemen, J.P. Gallagher,
  and M.V. Hermenegildo
Polymorphic Types in Erlang Function Specifications .......................... 181
  Francisco J. López-Fraguas, Manuel Montenegro,  
  and Juan Rodríguez-Hortalá

Declarative Foreign Function Binding Through Generic Programming ........ 198
  Jeremy Yallop, David Sheets, and Anil Madhavapeddy

Incremental Computing with Abstract Data Structures ........................ 215
  Akimasa Morihata

Declarative Programming with Algebra ........................................... 232
  Andre van Delft and Anatoliy Kmetyuk

Author Index ................................................................. 253