J. Functional Programming **11** (5): 439–440, September 2001. Printed in the United Kingdom

Special issue on

Algorithmic aspects of functional programming languages

Edited by

CHRIS OKASAKI
Department of Computer Science, Columbia University, NY, USA
(*e-mail:* cdo@cs.columbia.edu)

Algorithms can be dramatically affected by the language in which they are implemented. An algorithm that is elegant and efficient in one language may be ugly and inefficient in another. If you have ever attempted to implement an assignment-intensive algorithm in a functional programming language, you are probably more familiar with this phenomenon than you ever wanted to be! But this sword does not cut in only one direction. Functional programming languages are wonderfully suited to expressing certain kinds of algorithms in a clean, modular way, and researchers over the last five to ten years have greatly expanded the range of algorithms for which this is true.

In September 1999, the First Workshop on Algorithmic Aspects of Advanced Programming Languages (WAAAPL) was held in Paris, in conjunction with PLI'99, to bring together researchers in this fledgling field. This special issue builds on the success of that workshop by featuring expanded versions of three of its papers, together with two new papers on related topics.

In 'Modular lazy search for constraint satisfaction problems', Nordin and Tolmach demonstrate, once again, the modularity benefits of functional programming, especially lazy functional programming. They describe a framework for solving constraint satisfaction problems in which the important algorithmic components of common algorithms are isolated into separate functions, which can then be combined in nearly arbitrary ways. The contrast with traditional approaches to constraint satisfaction problems, which are typically presented as large monolithic chunks of code, is remarkable.

In 'Inductive graphs and functional graph algorithms', Erwig tackles the problem of how to write graph algorithms such as depth-first search cleanly in a functional language. Tree-based algorithms can often be implemented very elegantly in functional languages, but graph algorithms have historically been much more difficult to express. One of the major differences is pattern matching – trees are usually easy to express as inductive datatypes that mesh smoothly with pattern matching, whereas graphs are usually represented as some kind of array (perhaps an array of adjacency lists). Erwig proposes a view-like mechanism for pattern matching on graphs as if they were inductively defined, and shows how his approach increases the clarity of many common graph algorithms.

In 'Persistent triangulations', Blelloch *et al.* investigate uses of persistence in computational geometry. A persistent representation would seem to add an $O(\log n)$ overhead, but they show how one can sometimes compensate for this overhead by requiring fewer operations overall. In particular, they present a three-dimensional convex-hull algorithm that matches the $\Omega(n \log n)$ lower bound for this problem in spite of using persistent triangulations.

In 'Inductive benchmarking for purely functional data structures', Moss and Runciman tackle the problem of how to benchmark data structures in a purely functional setting. The catch is persistence – no one yet knows how to effectively benchmark persistent data structures. Moss and Runciman propose a method for accounting for persistence and describe *Auburn*, a tool they have developed that largely automates the benchmarking of data structures in Haskell.

In 'Manufacturing datatypes', Hinze describes a methodology for designing new functional data structures. He advocates first capturing the essence of some desired structural constraint by writing simple recursion equations on multisets of natural numbers that satisfy some related constraint. For example, if you wanted a data structure for square matrices you might first define the set of all square numbers. These recursion equations can then be mechanically transformed into Haskell type definitions.

My thanks to the many referees who aided in the production of this special issue, and especially to the original members of the program committee for WAAAPL.