J. Functional Programming 5 (3): 279–281, July 1995

Special Issue on State-of-the-art applications of pure functional programming languages

Edited By

PIETER HARTEL

Computer Systems Dept., Univ. of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands (email: pieter@fwi.uva.nl*)*

RINUS PLASMEIJER

Computing Science Institute, Univ. of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands (email: rinus@cs.kun.nl*)*

Can functional programs be used to build real applications? The mere fact that this question is being asked is encouraging. Functional programming is all too often perceived as an exotic, mostly theoretical activity that has no bearing on reality. 1994 saw two events specifically devoted to answering this question. The first was the Dagstuhl workshop on 'Functional programming in the real world', and the second the creation of this special issue. During the Dagstuhl workshop some 30 applications were presented. The call for papers for this special issue attracted 21 submissions. The sheer number of people working on applications indicates that solving real problems using functional languages is now becoming a serious proposition.

The contributions in this issue address several diverse application areas: nuclear physics, databases query processing, parallel vision, building a spread sheet, solid modelling and molecular biology. Several other areas such as operational research and engineering design were represented in the papers that could not be included in this issue. A number of papers were also submitted targeting typical computer science applications, such as building parsers, programming environments, and executable specifications. The six papers that make up this issue have been selected purely on the basis of the formal refereeing process.

Three of the papers describe prototypes of real systems. These are the database query processing, parallel vision and molecular biology systems. The production systems, which have been developed on the basis of these prototypes are used 'in anger'. The production systems have been implemented respectively as a shell script, a parallel Occam 2 program and as a C++ program. The prototypes are either incomplete, or too slow or space hungry to be used as a real application.

The other three papers describe kernel applications, that were built to gain understanding of the problem domain. These are the nuclear physics, spreadsheet and solid modelling programs. These kernel applications are not used as real applications because they are not sufficiently complete to do the job properly.

Four applications were built by expert functional programmers. Of the other two applications, the solid modelling code was built by experts in the problem domain to study patterns of computation that arise in constructive solid geometry. The authors chose a functional language because of the close correspondence between the mathematical nature of the problem domain and, especially, the higher order aspects of functional programming. The authors of the database query processing application set out as non-expert functional programmers. They found that when they first undertook to develop their code, that no functional language implementation would satisfy their needs. They had to build an implementation of their own, so that by the time the project was finished they had become experts in functional programming.

Each paper describes the lessons learned whilst writing the application. This includes the whole trajectory: design, implementation, performance and use. All authors found it particularly satisfying to design and implement applications in a functional language. This should not come as a surprise, considering that most authors are expert functional programmers.

The advantages that functional languages have to offer to the software engineer are substantial. These advantages carry over into the design phase of the application. The spread sheet, for example, offers a mode of symbolic evaluation not seen in commercial spreadsheets.

A number of disadvantages of functional programming are encountered in each of the papers in this issue. The space and time complexity of the algorithms that were implemented are difficult to control. Adequate profiling tools are desperately needed. Theory to reason about complexity issues is sorely missed. A big obstacle to really using functional applications is often their performance.

The prototypes and kernels described in this issue are written in the purely-functional languages Clean, Haskell, Hope+ and Miranda, and in the functional subset of the languages Standard ML and ID. No single language proved to be the most popular. However, in the entire collection of 21 submitted papers, Haskell was used by six authors, followed by Miranda (3), Standard ML (3) and ID (2). Some authors used several languages to compare aspects of languages or implementations for the problems at hand.

Most applications demonstrate the advantages and disadvantages of lazy functional programming, by discussing the effects that this has on expressiveness and performance. For example, the nuclear physics code benefits from laziness, but the performance penalty is high. The database query system uses laziness to get initial results for queries, while the system is busy searching for more results. In general, laziness is considered useful. It has the disadvantage that reasoning about the complexity of the programs is difficult.

The vision system is the only paper to discuss parallelism in depth. Of the 21 submitted papers only two mention parallelism. We are not sure what the meaning of this is. It might just be that building sequential applications is difficult enough to start with.

What motivates programmers to use a functional language? The predominant motivation in the submitted papers seems to be to show that a problem can be solved at least as well using a functional language. That is: to do as well or better

than C or Fortran. We think that this motivation is not the best one. The resulting functional programs may be concise and elegant, but almost nobody uses them. This is bound to be disappointing. In our opinion a far better motivation would be to show that functional programming is better for software engineering, design and prototyping than most mainstream languages. What appears to be a weakness is really a strength: using a functional programming language one suspects that the performance might ultimately not be satisfactory, so a strong emphasis can be placed on clarity and understanding rather than squeezing out the last drop of performance. Once the problem is fully and fundamentally understood, one proceeds to implement the solution in a language that delivers the required performance. This might be a traditional language, but we expect that the new generation of compilers for functional languages will deliver code which is sufficiently efficient.

We would hope to review the situation in a few years time to see whether functional programming has found the niche it deserves.