Imperative functional programming
Brief summary

Simon L Peyton Jones Philip Wadler*

September 21, 1992

Abstract

This is a brief summary of a paper which will appear in the proceedings of POPL 1993.

We present a new model, based on monads, for performing input/output in a non-strict, purely functional language. It is composable, extensible, efficient, requires no extensions to the type system, and extends smoothly to incorporate mixed-language working and in-place array updates.

1 Introduction

Input/output has always appeared to be one of the less satisfactory features of purely functional languages: fitting action into the functional paradigm feels like fitting a square block into a round hole. Closely related difficulties are associated with performing in-place update operations on arrays, and calling arbitrary procedures written in some other (possibly side-effecting) language.

There is no shortage of proposed solutions to these difficulties, some of which we survey later, but no one solution has become accepted as the consensus. This paper outlines a new approach based on monads [1–3], with a number of noteworthy features.

- *It is composable.* Large programs which engage in I/O are constructed by gluing together smaller programs that do so. Combined with higher-order functions and lazy evaluation, this gives a highly expressive medium in which to express I/O-performing computations — quite the reverse of the sentiment with which we began this section.

  We compare the monadic approach to I/O with other standard approaches: dialogues, continuations, and linear types.

- *It is easily extensible.* The key to our implementation is to extend Haskell with a single form that allows one to call an arbitrary C procedure, without losing referential transparency. Using it programmers can readily extend the power of the I/O system, by writing Haskell functions which call operating system procedures.

*Dept of Computing Science, University of Glasgow. Email: `{simonpj,wadler}@dcs.glagsow.ac.uk`

- *It is efficient.* Our Haskell compiler has C as its target code. Given a Haskell program performing an I/O loop, the compiler can produce C code which is very similar to that which one would write by hand.
- *Its efficiency is achieved by applying simple program transformations.* We use unboxed data types [4] to expose representation and order-of-evaluation detail to code-improving transformations, rather than relying on *ad hoc* optimisations in the code generator.
- *It extends uniformly to provide interleaved I/O and reference types.*
- *It extends uniformly to support incremental arrays with in-place update.* Our implementation is efficient enough that we can define monolithic Haskell array operations in terms of incremental arrays. Hudak have proposed a similar method based on continuations. Our method is more general than his in the following sense: monads can implement continuations, but not the converse. We also give a comparison with work on linear types.
- *It is based (only) on the Hindley-Milner type system.* Some other proposals require linear types or existential types; ours does not.

We have implemented all that we describe in the context of a compiler for Haskell with the exception of the extension to arrays and reference types. The entire I/O system provided by our compiler is written in Haskell, using the non-standard extensions we describe below. The language's standard `Dialogue` interface for I/O is supported by providing a function to convert a `Dialogue` into our `IO` monad. The system is freely available by FTP.

We do not claim any fundamental expressiveness or efficiency which is not obtainable through existing systems, except where arrays are concerned. Nevertheless we feel that the entire system works particularly smoothly as a whole, from the standpoint of both programmer and implementor.

References

[1] E Moggi, "Computational lambda calculus and monads," in *Logic in Computer Science, California*, IEEE, June 1989.

[2] PL Wadler, "Comprehending monads," in *Proc ACM Conference on Lisp and Functional Programming, Nice*, ACM, June 1990.

[3] PL Wadler, "The essence of functional programming," in *Proc Principles of Programming Languages*, ACM, Jan 1992.

[4] SL Peyton Jones & J Launchbury, "Unboxed values as first class citizens," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag, Sept 1991.