Lazy Functional State Threads: an abstract

John Launchbury and Simon Peyton Jones
University of Glasgow
G12 8QQ, Scotland

Abstract

Some algorithms make critical internal use of updatable state, even though their external specification is purely functional. Based on earlier work on monads, we present a way of securely encapsulating stateful computations that manipulate multiple, named, mutable objects, in the context of a non-strict, purely-functional language.

The security of the encapsulation is assured by the type system, using parametricity. Intriguingly, this parametricity requires the provision of a (single) constant with a rank-2 polymorphic type.

A full version of this paper appears in the Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI), Orlando, June 1994.

1 Overview

Purely functional programming languages allow many algorithms to be expressed very concisely, but there are a few algorithms in which in-place updatable state seems to play a crucial role. For these algorithms, purely-functional languages, which lack updatable state, appear to be inherently inefficient (Ponder, McGeer & Ng [1988]).

Take, for example, algorithms based on the use of incrementally-modified hash tables, where lookups are interleaved with the insertion of new items. Similarly, the union/find algorithm relies for its efficiency on the set representations being simplified each time the structure is examined. Likewise, many graph algorithms require a dynamically changing structure in which sharing is explicit, so that changes are visible non-locally.

There is, furthermore, one absolutely unavoidable use of state in every functional program: input/output. The plain fact of the matter is that the whole purpose of running a program, functional or otherwise, is to make some side effect on the world — an update-in-place, if you please. In many programs these I/O effects are rather complex, involving interleaved reads from and writes to the world state.

We use the term "stateful" to describe computations or algorithms in which the programmer really does want to manipulate (updatable) state. What has been lacking until now is a clean way of describing such algorithms in a functional language — especially a non-strict one — without throwing away the main virtues of functional languages: independence of order of evaluation (the Church-Rosser property), referential transparency, non-strict semantics, and so on.

In this paper we describe a way to express stateful algorithms in non-strict, purely-functional languages. The approach is a development of our earlier work on monadic I/O and state encapsulation (Launchbury [1993]; Peyton Jones & Wadler [1993]), but with an important technical innovation: we use parametric polymorphism to achieve safe encapsulation of state. It turns out that this allows mutable objects to be named without losing safety, and it also allows input/output to be smoothly integrated with other state mainpulation.

The other important feature of this paper is that it describes a complete system, and one that is implemented in the Glasgow Haskell compiler and freely available. The system has the following properties:

- Complete referential transparency is maintained. At first it is not clear what this statement means: how can a stateful computation be said to be referentially transparent? To be more precise, a stateful computation is a *state transformer*, that is, a function from an initial state to a final state. It is like a "script", detailing the actions to be performed on its input state. Like any other function, it is quite possible to apply a single stateful computation to more than one input state.

  So, a state transformer is a pure function. But, because we guarantee that the state is used in a single-threaded way, the final state can be constructed by modifying the input state *in-place*. This efficient implementation respects the purely-functional semantics of the state-transformer function, so all the usual techniques for reasoning about functional programs continue to work. Similarly, stateful programs can be exposed to the full range of program transformations applied by a compiler, with no special cases or side conditions.

- The programmer has complete control over where in-place updates are used and where they are not. For example, there is no complex analysis to determine when an array is used in a single-threaded way. Since the viability of the entire program may be predicated on the use of in-place updates, the programmer must be confident in, and be able to reason about, the outcome.

- Mutable objects can be *named.* This ability sounds innocuous enough, but once an object can be named its use cannot be controlled as readily. Yet naming is important. For example, it gives us the ability to manipulate multiple mutable objects simultaneously.

- Input/output takes its place as a specialised form of stateful computation. Indeed, the type of I/O-performing computations is an instance of the (more polymorphic) type of stateful computations. Along with I/O comes the ability to call imperative procedures written in other languages.

- It is possible to *encapsulate* stateful computations so that they appear to the rest of the program as pure (stateless) functions which are *guaranteed* by the type system to have no interactions whatever with other computations, whether stateful or otherwise (except via the values of arguments and results, of course).

  Complete safety is maintained by this encapsulation. A program may contain an arbitrary number of stateful sub-computations, each simultaneously active, without concern that a mutable object from one might be mutated by another.

- Stateful computations can even be performed *lazily* without losing safety. For example, suppose that stateful depth-first search of a graph returns a list of vertices in depth-first order. If the consumer of this list only evaluates the first few elements of the list, then only enough of the stateful computation is executed to produce those elements.

The full paper can be obtained by anonymous FTP from

```
ftp.dcs.glasgow.ac.uk
pub/glasgow-fp/tech-reports/FP-94-05:state.ps.Z
```

References

J Launchbury [June 1993], "Lazy imperative programming," in *Proc ACM Sigplan Workshop on State in Programming Languages, Copenhagen (available as YALEU/DCS/RR-968, Yale University)*, pp46–56.

SL Peyton Jones & PL Wadler [Jan 1993], "Imperative functional programming," in *20th ACM Symposium on Principles of Programming Languages, Charleston*, ACM, 71–84.

CG Ponder, PC McGeer & A P-C Ng [June 1988], "Are applicative languages inefficient?," *SIGPLAN Notices* 23, 135–139.