Towards Automating Inductive Proofs for State Monads

Richard J. Boulton
Department of Computing Science, University of Glasgow
17 Lilybank Gardens, Glasgow G12 8QQ, Scotland
E-mail: boulton@dcs.gla.ac.uk

July 2000

1 Introduction

A *monad* is an algebraic structure that distinguishes between values and computations [Mog91]. Monads are used extensively in functional programming to produce programs that are easily modified and also to capture non-functional features such as input/output [Wad92]. They are also used in programming language semantics. Thus, reasoning about functional programs and certain styles of semantics may require proofs involving monads. Typically, the proofs will also involve recursive functions, so mathematical induction is usually required.

One successful approach to automating inductive proofs is proof planning, which uses artificial intelligence planning techniques where the objects considered are proof methods. I have been investigating how proof planning might be used to automate proofs involving monads and *state monads* in particular. I have identified two features of these proofs that challenge current proof planning technology: higher-order terms and nested recursion. Each of these is discussed briefly below but first I give an introduction to monads.

2 Monads and State Monads

There is more than one (equivalent) formulation of a monad. Here, the formulation using the functions `unit` and `bind` is used. In this formulation a monad is a triple consisting of a polymorphic type constructor `monad` and the two functions just mentioned. In their most general form the functions have the following types:

$$\begin{aligned} \texttt{unit} &: \alpha \to (\alpha)\texttt{monad} \\ \texttt{bind} &: (\alpha)\texttt{monad} \to (\alpha \to (\beta)\texttt{monad}) \to (\beta)\texttt{monad} \end{aligned}$$

and satisfy the following three equations:

$$\begin{aligned} \texttt{bind}\ (\texttt{unit}\ a)\ k &= k\ a \\ \texttt{bind}\ m\ \texttt{unit} &= m \\ \texttt{bind}\ m\ (\lambda a.\ \texttt{bind}\ (k\ a)\ h) &= \texttt{bind}\ (\texttt{bind}\ m\ (\lambda a.\ k\ a))\ h \end{aligned}$$

In a state monad the type $(\alpha)\texttt{monad}$ is specialised to $\texttt{state} \to \alpha \times \texttt{state}$ and `unit` and `bind` can be defined (and renamed) as follows:

$$\begin{aligned} \texttt{unitS}\ x &= \lambda s_0.\ (x, s_0) \\ \texttt{bindS}\ m\ f &= \lambda s_0.\ (\lambda(x, s_1).\ f\ x\ s_1)\ (m\ s_0) \end{aligned}$$

The function `bindS` is higher-order and expressions containing it typically involve explicit λ-abstractions.

3 Proof Planning for Higher-Order Logic

Until recently proof planning research had focused on first-order formulae but the λ*Clam* system developed at Edinburgh [RSG98] has provided a way to plan proofs about higher-order formulae. As indicated above, this will be a necessary feature for reasoning about state monads, but as yet I have not implemented such reasoning in λ*Clam* or any other system.

4 Nested Recursion

A nested recursive function is one who's definition involves a recursive call nested within an argument position of another recursive call, e.g.:

$$f(\mathsf{c}(x)) = \ldots f(\ldots f(\ldots)\ldots)\ldots$$

Recursive functions defined in terms of nested applications of `bindS` often have an implicit nested recursion due to the way `bindS` is defined.

It turns out that nested recursive functions can be split into two categories according to whether or not a nested recursive call occurs in a recursive argument position of the outer call. If the nested call is in a non-recursive position, less needs to be done to extend the existing proof planning methods, and fortunately monads are in this category. Nevertheless, extensions to λ*Clam* for the more benign form of nested recursion will be required if it is to be used for planning proofs about monads.

References

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[RSG98] J. Richardson, A. Smaill, and I. Green. System description: Proof planning in higher-order logic with λ*Clam*. In *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 129–133, Springer, July 1998.

[Wad92] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, USA, January 1992.