Monads for functional programming

Philip Wadler, University of Glasgow*

Department of Computing Science, University of Glasgow, G12 8QQ, Scotland
(wadler@dcs.glasgow.ac.uk)

Abstract. The use of monads to structure functional programs is described. Monads provide a convenient framework for simulating effects found in other languages, such as global state, exception handling, output, or non-determinism. Three case studies are looked at in detail: how monads ease the modification of a simple evaluator; how monads act as the basis of a datatype of arrays subject to in-place update; and how monads can be used to build parsers.

1 Introduction

Shall I be pure or impure?

The functional programming community divides into two camps. *Pure* languages, such as Miranda[0] and Haskell, are lambda calculus pure and simple. *Impure* languages, such as Scheme and Standard ML, augment lambda calculus with a number of possible *effects*, such as assignment, exceptions, or continuations. Pure languages are easier to reason about and may benefit from lazy evaluation, while impure languages offer efficiency benefits and sometimes make possible a more compact mode of expression.

Recent advances in theoretical computing science, notably in the areas of type theory and category theory, have suggested new approaches that may integrate the benefits of the pure and impure schools. These notes describe one, the use of *monads* to integrate impure effects into pure functional languages.

The concept of a monad, which arises from category theory, has been applied by Moggi to structure the denotational semantics of programming languages [13, 14]. The same technique can be applied to structure functional programs [21, 23].

The applications of monads will be illustrated with three case studies. Section 2 introduces monads by showing how they can be used to structure a simple evaluator so that it is easy to modify. Section 3 describes the laws satisfied by

* Appears in: J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Proceedings of the Båstad Spring School, May 1995, Springer Verlag Lecture Notes in Computer Science 925. A previous version of this note appeared in: M. Broy, editor, *Program Design Calculi*, Proceedings of the Marktoberdorf Summer School, 30 July–8 August 1992. Some errata fixed August 2001; thanks to Dan Friedman for pointing these out.

[0] Miranda is a trademark of Research Software Limited.

monads. Section 4 shows how monads provide a new solution to the old problem of providing updatable state in pure functional languages. Section 5 applies monads to the problem of building recursive descent parsers; this is of interest in its own right, and because it provides a paradigm for sequencing and alternation, two of the central concepts of computing.

It is doubtful that the structuring methods presented here would have been discovered without the insight afforded by category theory. But once discovered they are easily expressed without any reference to things categorical. No knowledge of category theory is required to read these notes.

The examples will be given in Haskell, but no knowledge of that is required either. What is required is a passing familiarity with the basics of pure and impure functional programming; for general background see [3, 12]. The languages refered to are Haskell [4], Miranda [20], Standard ML [11], and Scheme [17].

2 Evaluating monads

Pure functional languages have this advantage: all flow of data is made explicit. And this disadvantage: sometimes it is painfully explicit.

A program in a pure functional language is written as a set of equations. Explicit data flow ensures that the value of an expression depends only on its free variables. Hence substitution of equals for equals is always valid, making such programs especially easy to reason about. Explicit data flow also ensures that the order of computation is irrelevant, making such programs susceptible to lazy evaluation.

It is with regard to modularity that explicit data flow becomes both a blessing and a curse. On the one hand, it is the ultimate in modularity. All data in and all data out are rendered manifest and accessible, providing a maximum of flexibility. On the other hand, it is the nadir of modularity. The essence of an algorithm can become buried under the plumbing required to carry data from its point of creation to its point of use.

Say I write an evaluator in a pure functional language.

- To add error handling to it, I need to modify each recursive call to check for and handle errors appropriately. Had I used an impure language with exceptions, no such restructuring would be needed.
- To add a count of operations performed to it, I need to modify each recursive call to pass around such counts appropriately. Had I used an impure language with a global variable that could be incremented, no such restructuring would be needed.
- To add an execution trace to it, I need to modify each recursive call to pass around such traces appropriately. Had I used an impure language that performed output as a side effect, no such restructuring would be needed.

Or I could use a *monad*.

These notes show how to use monads to structure an evaluator so that the changes mentioned above are simple to make. In each case, all that is required is to redefine the monad and to make a few local changes.

This programming style regains some of the flexibility provided by various features of impure languages. It also may apply when there is no corresponding impure feature. It does not eliminate the tension between the flexibility afforded by explicit data and the brevity afforded by implicit plumbing; but it does ameliorate it to some extent.

The technique applies not just to evaluators, but to a wide range of functional programs. For a number of years, Glasgow has been involved in constructing a compiler for the functional language Haskell. The compiler is itself written in Haskell, and uses the structuring technique described here. Though this paper describes the use of monads in a program tens of lines long, we have experience of using them in a program three orders of magnitude larger.

We begin with the basic evaluator for simple terms, then consider variations that mimic exceptions, state, and output. We analyse these for commonalities, and abstract from these the concept of a monad. We then show how each of the variations fits into the monadic framework.

2.1 Variation zero: The basic evaluator

The evaluator acts on terms, which for purposes of illustration have been taken to be excessively simple.

$$\textbf{data}\quad \mathit{Term} = \mathit{Con}\ \mathit{Int} \mid \mathit{Div}\ \mathit{Term}\ \mathit{Term}$$

A term is either a constant $\mathit{Con}\ a$, where a is an integer, or a quotient, $\mathit{Div}\ t\ u$, where t and u are terms.

The basic evaluator is simplicity itself.

$$\begin{array}{lcl} \mathit{eval} & :: & \mathit{Term} \to \mathit{Int} \\ \mathit{eval}\,(\mathit{Con}\ a) & = & a \\ \mathit{eval}\,(\mathit{Div}\ t\ u) & = & \mathit{eval}\ t \div \mathit{eval}\ u \end{array}$$

The function *eval* takes a term to an integer. If the term is a constant, the constant is returned. If the term is a quotient, its subterms are evaluated and the quotient computed. We use '$\div$' to denote integer division.

The following will provide running examples.

$$\begin{array}{lcl} \mathit{answer}, \mathit{error} & :: & \mathit{Term} \\ \mathit{answer} & = & (\mathit{Div}\,(\mathit{Div}\,(\mathit{Con}\ \mathit{1972})\,(\mathit{Con}\ \mathit{2}))\,(\mathit{Con}\ \mathit{23})) \\ \mathit{error} & = & (\mathit{Div}\,(\mathit{Con}\ \mathit{1})\,(\mathit{Con}\ \mathit{0})) \end{array}$$

Computing *eval answer* yields the value of $((\mathit{1972} \div \mathit{2}) \div \mathit{23})$, which is $\mathit{42}$. The basic evaluator does not incorporate error handling, so the result of *eval error* is undefined.

2.2 Variation one: Exceptions

Say it is desired to add error checking, so that the second example above returns a sensible error message. In an impure language, this is easily achieved with the use of exceptions.

In a pure language, exception handling may be mimicked by introducing a type to represent computations that may raise an exception.

$$\begin{array}{lll} \textbf{data} & M\ a & = \mathit{Raise}\ \mathit{Exception} \mid \mathit{Return}\ a \\ \textbf{type} & \mathit{Exception} & = \mathit{String} \end{array}$$

A value of type $M\ a$ either has the form $\mathit{Raise}\ e$, where e is an exception, or $\mathit{Return}\ a$, where a is a value of type a. By convention, a will be used both as a type variable, as in $M\ a$, and as a variable ranging over values of that type, as in $\mathit{Return}\ a$.

(A word on the difference between 'data' and 'type' declarations. A 'data' declaration introduces a new data type, in this case M, and new constructors for values of that type, in this case *Raise* and *Return*. A 'type' declaration introduces a new name for an existing type, in this case *Exception* becomes another name for *String*.)

It is straightforward, but tedious, to adapt the evaluator to this representation.

```
eval            :: Term → M Int
eval (Con a)    = Return a
eval (Div t u)  = case eval t of
                    Raise e → Raise e
                    Return a →
                      case eval u of
                        Raise e → Raise e
                        Return b →
                          if b = 0
                            then Raise "divide by zero"
                            else Return (a ÷ b)
```

At each call of the evaluator, the form of the result must be checked: if an exception was raised it is re-raised, and if a value was returned it is processed. Applying the new evaluator to *answer* yields ($\mathit{Return}\ 42$), while applying it to *error* yields (*Raise "divide by zero"*).

2.3 Variation two: State

Forgetting errors for the moment, say it is desired to count the number of divisions performed during evaluation. In an impure language, this is easily achieved by the use of state. Set a given variable to zero initially, and increment it by one each time a division occurs.

In a pure language, state may be mimicked by introducing a type to represent computations that act on state.

$$\begin{array}{lll} \textbf{type} & M\ a & = \mathit{State} \to (a, \mathit{State}) \\ \textbf{type} & \mathit{State} & = \mathit{Int} \end{array}$$

Now a value of type $M\ a$ is a function that accepts the initial state, and returns the computed value paired with the final state.

Again, it is straightforward but tedious to adapt the evaluator to this representation.

$$
\begin{array}{lll}
\mathit{eval} & :: & \mathit{Term} \to M\ \mathit{Int} \\
\mathit{eval}\,(\mathit{Con}\ a)\ x & = & (a, x) \\
\mathit{eval}\,(\mathit{Div}\ t\ u)\ x & = & \mathbf{let}\ \ (a, y) = \mathit{eval}\ t\ x\ \ \mathbf{in} \\
 & & \mathbf{let}\ \ (b, z) = \mathit{eval}\ u\ y\ \ \mathbf{in} \\
 & & (a \div b, z + \mathit{1})
\end{array}
$$

At each call of the evaluator, the old state must be passed in, and the new state extracted from the result and passed on appropriately. Computing *eval answer 0* yields $(\mathit{42}, \mathit{2})$, so with initial state *0* the answer is *42* and the final state is *2*, indicating that two divisions were performed.

2.4 Variation three: Output

Finally, say it is desired to display a trace of execution. In an impure language, this is easily done by inserting output commands at appropriate points.

In a pure language, output may be mimicked by introducing a type to represent computations that generate output.

$$
\begin{array}{lll}
\mathbf{type}\ \ M\ a & = & (\mathit{Output}, a) \\
\mathbf{type}\ \ \mathit{Output} & = & \mathit{String}
\end{array}
$$

Now a value of type $M\ a$ consists of the output generated paired with the value computed.

Yet again, it is straightforward but tedious to adapt the evaluator to this representation.

$$
\begin{array}{lll}
\mathit{eval} & :: & \mathit{Term} \to M\ \mathit{Int} \\
\mathit{eval}\,(\mathit{Con}\ a) & = & (\mathit{line}\,(\mathit{Con}\ a)\ a, a) \\
\mathit{eval}\,(\mathit{Div}\ t\ u) & = & \mathbf{let}\ \ (x, a) = \mathit{eval}\ t\ \ \mathbf{in} \\
 & & \mathbf{let}\ \ (y, b) = \mathit{eval}\ u\ \ \mathbf{in} \\
 & & (x +\!\!+\ y +\!\!+\ \mathit{line}\,(\mathit{Div}\ t\ u)\,(a \div b), a \div b) \\
\mathit{line} & :: & \mathit{Term} \to \mathit{Int} \to \mathit{Output} \\
\mathit{line}\ t\ a & = & \text{“}\mathit{eval}\,(\text{”} +\!\!+\ \mathit{showterm}\ t +\!\!+\ \text{“}) \Leftarrow \text{”} +\!\!+\ \mathit{showint}\ a +\!\!+\ \text{“}\hookleftarrow\text{”}
\end{array}
$$

At each call of the evaluator, the outputs must be collected and assembled to form the output of the enclosing call. The function *line* generates one line of the output. Here *showterm* and *showint* convert terms and integers to strings, $+\!\!+$ concatenates strings, and “$\hookleftarrow$” represents the string consisting of a newline.

Computing *eval answer* returns the pair $(x, \mathit{42})$, where x is the string

$$
\begin{array}{l}
\mathit{eval}\,(\mathit{Con}\ \mathit{1972}) \Leftarrow \mathit{1972} \\
\mathit{eval}\,(\mathit{Con}\ \mathit{2}) \Leftarrow \mathit{2} \\
\mathit{eval}\,(\mathit{Div}\,(\mathit{Con}\ \mathit{1972})\,(\mathit{Con}\ \mathit{2})) \Leftarrow \mathit{986} \\
\mathit{eval}\,(\mathit{Con}\ \mathit{23}) \Leftarrow \mathit{23} \\
\mathit{eval}\,(\mathit{Div}\,(\mathit{Div}\,(\mathit{Con}\ \mathit{1972})\,(\mathit{Con}\ \mathit{2}))\,(\mathit{Con}\ \mathit{23}))) \Leftarrow \mathit{42}
\end{array}
$$

which represents a trace of the computation.

From the discussion so far, it may appear that programs in impure languages are easier to modify than those in pure languages. But sometimes the reverse is true. Say that it was desired to modify the previous program to display the execution trace in the reverse order:

$$\begin{array}{l}
\mathit{eval}\,(\mathit{Div}\,(\mathit{Div}\,(\mathit{Con}\ 1972)\,(\mathit{Con}\ 2))\,(\mathit{Con}\ 23)) \Leftarrow 42 \\
\mathit{eval}\,(\mathit{Con}\ 23) \Leftarrow 23 \\
\mathit{eval}\,(\mathit{Div}\,(\mathit{Con}\ 1972)\,(\mathit{Con}\ 2)) \Leftarrow 986 \\
\mathit{eval}\,(\mathit{Con}\ 2) \Leftarrow 2 \\
\mathit{eval}\,(\mathit{Con}\ 1972) \Leftarrow 1972
\end{array}$$

This is simplicity itself to achieve with the pure program: just replace the term

$$x +\!\!+\, y +\!\!+\, \mathit{line}\,(\mathit{Div}\ t\ u)\,(a \div b)$$

with the term

$$\mathit{line}\,(\mathit{Div}\ t\ u)\,(a \div b) +\!\!+\, y +\!\!+\, x.$$

It is not so easy to modify the impure program to achieve this effect. The problem is that output occurs as a side-effect of computation, but one now desires to display the result of computing a term *before* displaying the output generated by that computation. This can be achieved in a variety of ways, but all require substantial modification to the impure program.

2.5 A monadic evaluator

Each of the variations on the interpreter has a similar structure, which may be abstracted to yield the notion of a *monad*.

In each variation, we introduced a type of computations. Respectively, M represented computations that could raise exceptions, act on state, and generate output. By now the reader will have guessed that M stands for *monad*.

The original evaluator has the type $\mathit{Term} \rightarrow \mathit{Int}$, while in each variation its type took the form $\mathit{Term} \rightarrow M\ \mathit{Int}$. In general, a function of type $a \rightarrow b$ is replaced by a function of type $a \rightarrow M\ b$. This can be read as a function that accepts an argument of type a and returns a result of type b, with a possible additional effect captured by M. This effect may be to act on state, generate output, raise an exception, or what have you.

What sort of operations are required on the type M? Examination of the examples reveals two. First, we need a way to turn a value into the computation that returns that value and does nothing else.

$$\mathit{unit} :: a \rightarrow M\ a$$

Second, we need a way to apply a function of type $a \rightarrow M\ b$ to a computation of type $M\ a$. It is convenient to write these in an order where the argument comes before the function.

$$(\star) :: M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$$

A *monad* is a triple $(M, unit, \star)$ consisting of a type constructor M and two operations of the given polymorphic types. These operations must satisfy three laws given in Section 3.

We will often write expressions in the form

$$m \star \lambda a.\, n$$

where m and n are expressions, and a is a variable. The form $\lambda a.\, n$ is a lambda expression, with the scope of the bound variable a being the expression n. The above can be read as follows: perform computation m, bind a to the resulting value, and then perform computation n. Types provide a useful guide. From the type of $(\star)$, we can see that expression m has type $M\ a$, variable a has type a, expression n has type $M\ b$, lambda expression $\lambda a.\, n$ has type $a \rightarrow M\ b$, and the whole expression has type $M\ b$.

The above is analogous to the expression

$$\mathbf{let}\quad a = m \quad \mathbf{in}\quad n.$$

In an impure language, this has the same reading: perform computation m, bind a to the resulting value, then perform computation n and return its value. Here the types say nothing to distinguish values from computations: expression m has type a, variable a has type a, expression n has type b, and the whole expression has type b. The analogy with 'let' explains the choice of the order of arguments to $\star$. It is convenient for argument m to appear before the function $\lambda a.\, n$, since computation m is performed before computation n.

The evaluator is easily rewritten in terms of these abstractions.

$$\begin{array}{lcl}
eval & :: & Term \rightarrow M\ Int \\
eval\,(Con\ a) & = & unit\ a \\
eval\,(Div\ t\ u) & = & eval\ t \star \lambda a.\, eval\ u \star \lambda b.\, unit\,(a \div b)
\end{array}$$

A word on precedence: lambda abstraction binds least tightly and application binds most tightly, so the last equation is equivalent to the following.

$$eval\,(Div\ t\ u) = ((eval\ t) \star (\lambda a.\, ((eval\ u) \star (\lambda b.\, (unit\,(a \div b))))))$$

The type $Term \rightarrow M\ Int$ indicates that the evaluator takes a term and performs a computation yielding an integer. To compute $(Con\ a)$, just return a. To compute $(Div\ t\ u)$, first compute t, bind a to the result, then compute u, bind b to the result, and then return $a \div b$.

The new evaluator is a little more complex than the original basic evaluator, but it is much more flexible. Each of the variations given above may be achieved by simply changing the definitions of M, $unit$, and $\star$, and by making one or two local modifications. It is no longer necessary to re-write the entire evaluator to achieve these simple changes.

2.6 Variation zero, revisited: The basic evaluator

In the simplest monad, a computation is no different from a value.

$$
\begin{array}{ll}
\textbf{type}\;\; M\ a = a & \\
\mathit{unit} & ::\ a \rightarrow I\ a \\
\mathit{unit}\ a & =\ a \\
(\star) & ::\ M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b \\
a \star k & =\ k\ a
\end{array}
$$

This is called the *identity* monad: M is the identity function on types, *unit* is the identity function, and $\star$ is just application.

Taking M, *unit*, and $\star$ as above in the monadic evaluator of Section 2.5 and simplifying yields the basic evaluator of Section 2.1

2.7 Variation one, revisited: Exceptions

In the exception monad, a computation may either raise an exception or return a value.

$$
\begin{array}{ll}
\textbf{data}\;\; M\ a & =\ \mathit{Raise\ Exception} \mid \mathit{Return}\ a \\
\textbf{type}\;\; \mathit{Exception} & =\ \mathit{String} \\
\mathit{unit} & ::\ a \rightarrow M\ a \\
\mathit{unit}\ a & =\ \mathit{Return}\ a \\
(\star) & ::\ M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b \\
m \star k & =\ \textbf{case}\;\; m\;\; \textbf{of} \\
 & \qquad \mathit{Raise}\ e \rightarrow \mathit{Raise}\ e \\
 & \qquad \mathit{Return}\ a \rightarrow k\ a \\
\mathit{raise} & ::\ \mathit{Exception} \rightarrow M\ a \\
\mathit{raise}\ e & =\ \mathit{Raise}\ e
\end{array}
$$

The call *unit* a simply returns the value a. The call $m \star k$ examines the result of the computation m: if it is an exception it is re-raised, otherwise the function k is applied to the value returned. Just as $\star$ in the identity monad is function application, $\star$ in the exception monad may be considered as a form of *strict* function application. Finally, the call *raise* e raises the exception e.

To add error handling to the monadic evaluator, take the monad as above. Then just replace *unit* $(a \div b)$ by

$$
\begin{array}{l}
\textbf{if}\;\; b = 0 \\
\quad \textbf{then}\;\; \mathit{raise}\ \text{“}\mathit{divide\ by\ zero}\text{”} \\
\quad \textbf{else}\;\;\ \mathit{unit}\,(a \div b)
\end{array}
$$

This is commensurate with change required in an impure language.

As one might expect, this evaluator is equivalent to the evaluator with exceptions of Section 2.2. In particular, unfolding the definitions of *unit* and $\star$ in this section and simplifying yields the evaluator of that section.

2.8 Variation two, revisited: State

In the state monad, a computation accepts an initial state and returns a value paired with the final state.

$$
\begin{array}{lll}
\mathbf{type}\ M\ a & = & \mathit{State} \to (a, \mathit{State}) \\
\mathbf{type}\ \mathit{State} & = & \mathit{Int} \\
\\
\mathit{unit} & :: & a \to M\ a \\
\mathit{unit}\ a & = & \lambda x.\, (a, x) \\
\\
(\star) & :: & M\ a \to (a \to M\ b) \to M\ b \\
m \star k & = & \lambda x.\, \mathbf{let}\ \ (a, y) = m\, x\ \ \mathbf{in} \\
 & & \quad\ \ \mathbf{let}\ \ (b, z) = k\ a\ y\ \ \mathbf{in} \\
 & & \quad\ \ (b, z) \\
\\
\mathit{tick} & :: & M\ () \\
\mathit{tick} & = & \lambda x.\, ((), x + \mathit{1})
\end{array}
$$

The call $\mathit{unit}\, a$ returns the computation that accept initial state x and returns value a and final state x; that is, it returns a and leaves the state unchanged. The call $m \star k$ performs computation m in the initial state x, yielding value a and intermediate state y; then performs computation $k\ a$ in state y, yielding value b and final state z. The call *tick* increments the state, and returns the empty value $()$, whose type is also written $()$.

In an impure language, an operation like *tick* would be represented by a function of type $() \to ()$. The spurious argument $()$ is required to delay the effect until the function is applied, and since the output type is $()$ one may guess that the function's purpose lies in a side effect. In contrast, here *tick* has type $M\ ()$: no spurious argument is needed, and the appearance of M explicitly indicates what sort of effect may occur.

To add execution counts to the monadic evaluator, take the monad as above. Then just replace $\mathit{unit}\, (a \div b)$ by

$$\mathit{tick} \star \lambda().\, \mathit{unit}\, (a \div b)$$

(Here $\lambda\_.\, e$ is equivalent to $\lambda x.\, e$ where $x :: ()$ is some fresh variable that does not appear in e; it indicates that the value bound by the lambda expression must be $()$.) Again, this is commensurate with change required in an impure language. Simplifying yields the evaluator with state of Section 2.3.

2.9 Variation three, revisited: Output

In the output monad, a computation consists of the output generated paired with the value returned.

$$
\begin{array}{lll}
\textbf{type}\ M\ a & = & (Output, a) \\
\textbf{type}\ Output & = & String \\
\\
unit & :: & a \to M\ a \\
unit\ a & = & (\text{“ ”}, a) \\
\\
(\star) & :: & M\ a \to (a \to M\ b) \to M\ b \\
m \star k & = & \textbf{let}\ \ (x, a) = m\ \ \textbf{in} \\
 & & \textbf{let}\ \ (y, b) = k\ a\ \ \textbf{in} \\
 & & (x +\!\!+\ y, b) \\
\\
out & :: & Output \to M\ () \\
out\ x & = & (x, ())
\end{array}
$$

The call $unit\ a$ returns no output paired with a. The call $m \star k$ extracts output x and value a from computation m, then extracts output y and value b from computation $k\ a$, and returns the output formed by concatenating x and y paired with the value b. The call $out\ x$ returns the computation with output x and empty value $()$.

To add execution traces to the monadic evaluator, take the monad as above. Then in the clause for $Con\ a$ replace $unit\ a$ by

$$out\ (line\ (Con\ a)\ a) \star \lambda().\ unit\ a$$

and in the clause for $Div\ t\ u$ replace $unit\ (a \div b)$ by

$$out\ (line\ (Div\ t\ u)\ (a \div b)) \star \lambda().\ unit\ (a \div b)$$

Yet again, this is commensurate with change required in an impure language. Simplifying yields the evaluator with output of Section 2.4.

To get the output in the reverse order, all that is required is to change the definition of $\star$, replacing $x +\!\!+\ y$ by $y +\!\!+\ x$. This is commensurate with the change required in the pure program, and rather simpler than the change required in an impure program.

You might think that one difference between the pure and impure versions is that the impure version displays output as it computes, while the pure version will display nothing until the entire computation completes. In fact, if the pure language is lazy then output will be displayed in an incremental fashion as the computation occurs. Furthermore, this will also happen if the order of output is reversed, which is much more difficult to arrange in an impure language. Indeed, the easiest way to arrange it may be to simulate lazy evaluation.

3 Monad laws

The operations of a monad satisfy three laws.

- *Left unit.* Compute the value a, bind b to the result, and compute n. The result is the same as n with value a substituted for variable b.

$$\mathit{unit}\ a \star \lambda b.\ n = n[a/b].$$

- *Right unit.* Compute m, bind the result to a, and return a. The result is the same as m.

$$m \star \lambda a.\ \mathit{unit}\ a = m.$$

- *Associative.* Compute m, bind the result to a, compute n, bind the result to b, compute o. The order of parentheses in such a computation is irrelevant.

$$m \star (\lambda a.\ n \star \lambda b.\ o) = (m \star \lambda a.\ n) \star \lambda b.\ o.$$

  The scope of the variable a includes o on the left but excludes o on the right, so this law is valid only when a does not appear free in o.

A binary operation with left and right unit that is associative is called a *monoid.* A monad differs from a monoid in that the right operand involves a binding operation.

To demonstrate the utility of these laws, we prove that addition is associative. Consider a variant of the evaluator based on addition rather than division.

$$\begin{array}{lll}
\textbf{data}\ \ \mathit{Term} & = & \mathit{Con}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Term}\ \mathit{Term} \\[4pt]
\mathit{eval} & :: & \mathit{Term} \to M\ \mathit{Int} \\
\mathit{eval}\,(\mathit{Con}\ a) & = & \mathit{unit}\ a \\
\mathit{eval}\,(\mathit{Add}\ t\ u) & = & \mathit{eval}\ t \star \lambda a.\ \mathit{eval}\ u \star \lambda b.\ \mathit{unit}\,(a \div b)
\end{array}$$

We show that evaluation of

$$\mathit{Add}\ t\,(\mathit{Add}\ u\ v) \quad \text{and} \quad \mathit{Add}\,(\mathit{Add}\ t\ u)\ v,$$

both compute the same result. Simplify the left term:

$$\begin{array}{ll}
 & \mathit{eval}\,(\mathit{Add}\ t\,(\mathit{Add}\ u\ v)) \\
= & \{\ \text{def'n}\ \mathit{eval}\ \} \\
 & \mathit{eval}\ t \star \lambda a.\ \mathit{eval}\,(\mathit{Add}\ u\ v) \star \lambda x.\ \mathit{unit}\,(a + x) \\
= & \{\ \text{def'n}\ \mathit{eval}\ \} \\
 & \mathit{eval}\ t \star \lambda a.\ (\mathit{eval}\ u \star \lambda b.\ \mathit{eval}\ v \star \lambda c.\ \mathit{unit}\,(b + c)) \star \lambda x.\ \mathit{unit}\,(a + x) \\
= & \{\ \text{associative}\ \} \\
 & \mathit{eval}\ t \star \lambda a.\ \mathit{eval}\ u \star \lambda b.\ \mathit{eval}\ v \star \lambda c.\ \mathit{unit}\,(b + c) \star \lambda x.\ \mathit{unit}\,(a + x) \\
= & \{\ \text{left unit}\ \} \\
 & \mathit{eval}\ t \star \lambda a.\ \mathit{eval}\ u \star \lambda b.\ \mathit{eval}\ v \star \lambda c.\ \mathit{unit}\,(a + (b + c))
\end{array}$$

Simplify the right term similarly:

$$\begin{array}{ll}
 & \mathit{eval}\,(\mathit{Add}\,(\mathit{Add}\ t\ u)\ v) \\
= & \{\ \text{as before}\ \} \\
 & \mathit{eval}\ t \star \lambda a.\ \mathit{eval}\ u \star \lambda b.\ \mathit{eval}\ v \star \lambda c.\ \mathit{unit}\,((a + b) + c)
\end{array}$$

The result follows by the associativity of addition. This proof is trivial; without the monad laws, it would be impossible.

The proof works in any monad: exception, state, output. This assumes that the code is as above: if it is modified then the proof also must be modified. Section 2.3 modified the program by adding calls to *tick*. In this case, associativity still holds, as can be demonstrated using the law

$$tick \star \lambda().\, m = m \star \lambda().\, tick$$

which holds so long at *tick* is the only action on state within m. Section 2.4 modified the program by adding calls to *line*. In this case, the addition is no longer associative, in the sense that changing parentheses will change the trace, though the computations will still yield the same value.

As another example, note that for each monad we can define the following operations.

$$\begin{aligned}
&map && :: (a \to b) \to (M\ a \to M\ b) \\
&map\, f\ m && = m \star \lambda a.\, unit\, (f\ a) \\[4pt]
&join && :: M\ (M\ a) \to M\ a \\
&join\ z && = z \star \lambda m.\, m
\end{aligned}$$

The *map* operation simply applies a function to the result yielded by a computation. To compute $map\, f\ m$, first compute m, bind a to the result, and then return $f\ a$. The *join* operation is trickier. Let z be a computation that *itself* yields a computation. To compute $join\ z$, first compute z, binds m to the result, and then behaves as computation m. Thus, *join* flattens a mind-boggling double layer of computation into a run-of-the-mill single layer of computation. As we will see in Section 5.1, lists form a monad, and for this monad *map* applies a function to each element of a list, and *join* concatenates a list of lists.

Using *id* for the identity function ($id\ x = x$), and $(\cdot)$ for function composition ($((f \cdot g)\, x = f\ (g\ x))$), one can then formulate a number of laws.

$$\begin{aligned}
&map\ id && = id \\
&map\ (f \cdot g) && = map\, f \cdot map\ g \\[4pt]
&map\, f \cdot unit && = unit \cdot f \\
&map\, f \cdot join && = join \cdot map\, (map\, f) \\[4pt]
&join \cdot unit && = id \\
&join \cdot map\ unit && = id \\
&join \cdot map\ join && = join \cdot join \\[4pt]
&m \star k && = join\, (map\ k\ m)
\end{aligned}$$

The proof of each is a simple consequence of the definitions of *map* and *join* and the three monad laws.

Often, monads are defined not in terms of *unit* and $\star$, but rather in terms of *unit*, *join*, and *map* [10, 13]. The three monad laws are replaced by the first seven of the eight laws above. If one defines $\star$ by the eighth law, then the three monad laws follow. Hence the two definitions are equivalent.

4 State

Arrays play a central role in computing because they closely match current architectures. Programs are littered with array lookups such as $x[i]$ and array updates such as $x[i] := v$. These operations are popular because array lookup is implemented by a single indexed fetch instruction, and array update by a single indexed store.

It is easy to add arrays to a functional language, and easy to provide efficient array lookup. How to provide efficient array update, on the other hand, is a question with a long history. Monads provide a new answer to this old question.

Another question with a long history is whether it is *desirable* to base programs on array update. Since so much effort has gone into developing algorithms and architectures based on arrays, we will sidestep this debate and simply assume the answer is yes.

There is an important difference between the way monads are used in the previous section and the way monads are used here. The previous section showed monads help to use existing language features more effectively; this section shows how monads can help define new language features. No change to the programming language is required, but the implementation must provide a new abstract data type, perhaps as part of the standard prelude.

Here monads are used to manipulate state internal to the program, but the same techniques can be use to manipulate extenal state: to perform input/output, or to communicate with other programming languages. The Glasgow implementation of Haskell uses a design based on monads to provide input/output and interlanguage working with the imperative programming language C [15]. This design has been adopted for version 1.3 of the Haskell standard.

4.1 Arrays

Let Arr be the type of arrays taking indexes of type Ix and yielding values of type Val. The key operations on this type are

$$\begin{aligned}
&newarray :: Val \to Arr, \\
&index \quad\;\; :: Ix \to Arr \to Val, \\
&update \quad :: Ix \to Val \to Arr \to Arr.
\end{aligned}$$

The call $newarray\ v$ returns an array with all entries set to v; the call $index\ i\ x$ returns the value at index i in array x; and the call $update\ i\ v\ x$ returns an array where index i has value v and the remainder is identical to x. The behaviour of these operations is specified by the laws

$$\begin{aligned}
index\ i\ (newarray\ v) &= v, \\
index\ i\ (update\ i\ v\ x) &= v, \\
index\ i\ (update\ j\ v\ x) &= index\ i\ x, \text{ if } i \neq j.
\end{aligned}$$

In practice, these operations would be more complex; one needs a way to specify the index bounds, for instance. But the above suffices to explicate the main points.

The efficient way to implement the update operation is to overwrite the specified entry of the array, but in a pure functional language this is only safe if there are no other pointers to the array extant when the update operation is performed. An array satisfying this property is called *single threaded*, following Schmidt [18].

Consider building an interpreter for a simple imperative language. The abstract syntax for this language is represented by the following data types.

$$\begin{array}{lll}
\mathbf{data} & \mathit{Term} & = \mathit{Var}\ \mathit{Id} \mid \mathit{Con}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Term}\ \mathit{Term} \\
\mathbf{data} & \mathit{Comm} & = \mathit{Asgn}\ \mathit{Id}\ \mathit{Term} \mid \mathit{Seq}\ \mathit{Comm}\ \mathit{Comm} \mid \mathit{If}\ \mathit{Term}\ \mathit{Comm}\ \mathit{Comm} \\
\mathbf{data} & \mathit{Prog} & = \mathit{Prog}\ \mathit{Comm}\ \mathit{Term}
\end{array}$$

Here $\mathit{Id} : \mathit{baastad.tex}, \mathit{v1.1.1.11996/02/2915} : \mathit{17} : \mathit{01wadlerExp}$ is an unspecified type of identifiers. A term is a variable, a constant, or the sum of two terms; a command is an assignment, a sequence of two commands, or a conditional; and a program consists of a command followed by a term.

The current state of execution will be modelled by an array where the indexes are identifiers and the corresponding values are integers.

$$\begin{array}{lll}
\mathbf{type} & \mathit{State} & = \mathit{Arr} \\
\mathbf{type} & \mathit{Ix} & = \mathit{Id} \\
\mathbf{type} & \mathit{Val} & = \mathit{Int}
\end{array}$$

Here is the interpreter.

$$\begin{array}{ll}
\mathit{eval} & :: \mathit{Term} \to \mathit{State} \to \mathit{Int} \\
\mathit{eval}\,(\mathit{Var}\ i)\ x & = \mathit{index}\ i\ x \\
\mathit{eval}\,(\mathit{Con}\ a)\ x & = a \\
\mathit{eval}\,(\mathit{Add}\ t\ u)\ x & = \mathit{eval}\ t\ x + \mathit{eval}\ u\ x \\
\\
\mathit{exec} & :: \mathit{Comm} \to \mathit{State} \to \mathit{State} \\
\mathit{exec}\,(\mathit{Asgn}\ i\ t)\ x & = \mathit{update}\ i\,(\mathit{eval}\ t\ x)\ x \\
\mathit{exec}\,(\mathit{Seq}\ c\ d)\ x & = \mathit{exec}\ d\,(\mathit{exec}\ c\ x) \\
\mathit{exec}\,(\mathit{If}\ t\ c\ d)\ x & = \mathbf{if}\ \ \mathit{eval}\ t\ x == \mathit{0}\ \ \mathbf{then}\ \ \mathit{exec}\ c\ x\ \ \mathbf{else}\ \ \mathit{exec}\ d\ x \\
\\
\mathit{elab} & :: \mathit{Prog} \to \mathit{Int} \\
\mathit{elab}\,(\mathit{Prog}\ c\ t) & = \mathit{eval}\ t\,(\mathit{exec}\ c\,(\mathit{newarray}\ \mathit{0}))
\end{array}$$

This closely resembles a denotational semantics. The evaluator for terms takes a term and a state and returns an integer. Evaluation of a variable is implemented by indexing the state. The executor for commands takes a command and the initial state and returns the final state. Assignment is implemented by updating the state. The elaborator for programs takes a program and returns an integer. It executes the command in an initial state where all identifiers map to $\mathit{0}$, then evaluates the given expression in the resulting state and returns its value.

The state in this interpreter is single threaded: at any moment of execution there is only one pointer to the state, so it is safe to update the state in place. In order for this to work, the update operation must evaluate the new value before placing it in the array. Otherwise, the array may contain a closure that itself contains a pointer to the array, violating the single threading property. In semantic terms, one says that *update* is strict in all three of its arguments.

A number of researchers have proposed analyses that determine whether a given functional program uses an array in a single threaded manner, with the intent of incorporating such an analysis into an optimising compiler. Most of these analyses are based on abstract interpretation [1]. Although there has been some success in this area, the analyses tend to be so expensive as to be intractable [2, 7].

Even if such analyses were practicable, their use may be unwise. Optimising update can affect a program's time and space usage by an order of magnitude or more. The programmer must be assured that such an optimisation will occur in order to know that the program will run adequately fast and within the available space. It may be better for the programmer to indicate explicitly that an array should be single threaded, rather than leave it to the vagaries of an optimising compiler.

Again, a number of researchers have proposed techniques for indicating that an array is single threaded. Most of these techniques are based on type systems [6, 19, 22]. This area seems promising, although the complexities of these type systems remain formidable.

The following section presents another way of indicating explicitly the intention that an array be single threaded. Naturally, it is based on monads. The advantage of this method is that it works with existing type systems, using only the idea of an abstract data type.

4.2 Array transformers

The monad of array transformers is simply the monad of state transformers, with the state taken to be an array. The definitions of M, *unit*, $\star$ are as before.

$$
\begin{array}{lll}
\textbf{type}\ \ M\ a & = & \mathit{State} \to (a, \mathit{State}) \\
\textbf{type}\ \ \mathit{State} & = & \mathit{Arr} \\[4pt]
\mathit{unit} & :: & a \to M\ a \\
\mathit{unit}\ a & = & \lambda x.\, (a, x) \\[4pt]
(\star) & :: & M\ a \to (a \to M\ b) \to M\ b \\
m \star k & = & \lambda x.\, \textbf{let}\ \ (a, y) = m\ x\ \ \textbf{in} \\
 & & \textbf{let}\ \ (b, z) = k\ a\ y\ \ \textbf{in} \\
 & & (b, z)
\end{array}
$$

Previously, our state was an integer and we had an additional operation *tick* acting upon the state. Now our state is an array, and we have additional operations corresponding to array creation, indexing, and update.

$$
\begin{array}{lcl}
\mathit{block} & :: & \mathit{Val} \to M\ a \to a \\
\mathit{block}\ v\ m & = & \mathbf{let}\ \ (a, x) = m\,(\mathit{newarray}\ v)\ \ \mathbf{in}\ \ a \\[4pt]
\mathit{fetch} & :: & \mathit{Ix} \to M\ \mathit{Val} \\
\mathit{fetch}\ i & = & \lambda x.\,(\mathit{index}\ i\ x, x) \\[4pt]
\mathit{assign} & :: & \mathit{Ix} \to \mathit{Val} \to M\ () \\
\mathit{assign}\ i\ v & = & \lambda x.\,((), \mathit{update}\ i\ v\ x)
\end{array}
$$

The call $\mathit{block}\ v\ m$ creates a new array with all locations initialised to v, applies monad m to this initial state to yield value a and final state x, deallocates the array, and returns a. The call $\mathit{fetch}\ i$ returns the value at index i in the current state, and leaves the state unchanged. The call $\mathit{assign}\ i\ v$ returns the empty value $()$, and updates the state so that index i contains value v.

A little thought shows that these operations are indeed single threaded. The only operation that could duplicate the array is *fetch*, but this may be implemented as follows: first fetch the entry at the given index in the array, and then return the pair consisting of this value and the pointer to the array. In semantic terms, *fetch* is strict in the array and the index, but not in the value located at the index, and *assign* is strict in the array and the index, but not the value assigned.

(This differs from the previous section, where in order for the interpreter to be single threaded it was necessary for *update* to be strict in the given value. In this section, as we shall see, this strictness is removed but a spurious sequencing is introduced for evaluation of terms. In the following section, the spurious sequencing is removed, but the strictness will be reintroduced.)

We may now make M into an *abstract data type* supporting the five operations *unit*, $\star$, *block*, *fetch*, and *assign*. The operation *block* plays a central role, as it is the only one that does not have M in its result type. Without *block* there would be no way to write a program using M that did not have M in its output type.

Making M into an abstract data type guarantees that single threading is preserved, and hence it is safe to implement assignment with an in-place update. The use of data abstraction is essential for this purpose. Otherwise, one could write programs such as

$$\lambda x.\,(\mathit{assign}\ i\ v\ x, \mathit{assign}\ i\ w\ x)$$

that violate the single threading property.

The interpreter may now be rewritten as follows.

$$
\begin{array}{lll}
eval & :: & Term \to M\ Int \\
eval\ (Var\ i) & = & fetch\ i \\
eval\ (Con\ a) & = & unit\ a \\
eval\ (Add\ t\ u) & = & eval\ t \star \lambda a.\ eval\ u \star \lambda b.\ unit\ (a + b) \\
 & & \\
exec & :: & Comm \to M\ () \\
exec\ (Asgn\ i\ t) & = & eval\ t \star \lambda a.\ assign\ i\ a \\
exec\ (Seq\ c\ d) & = & exec\ c \star \lambda().\ exec\ d \star \lambda().\ unit\ () \\
exec\ (If\ t\ c\ d) & = & eval\ t \star \lambda a. \\
 & & \quad \mathbf{if}\ \ a = 0\ \ \mathbf{then}\ \ exec\ c\ \ \mathbf{else}\ \ exec\ d \\
 & & \\
elab & :: & Prog \to Int \\
elab\ (Prog\ c\ t) & = & block\ 0\ (exec\ c \star \lambda().\ eval\ t \star \lambda a.\ unit\ a)
\end{array}
$$

The types show that evaluation of a term returns an integer and may access or modify the state, and that execution of a term returns nothing and may access or modify the state. In fact, evaluation only accesses the state and never alters it — we will consider shortly a more refined system that allows us to indicate this.

The abstract data type for M guarantees that it is safe to perform updates in place – no special analysis technique is required. It is easy to see how the monad interpreter can be derived from the original, and (using the definitions given earlier) the proof of their equivalence is straightforward.

The rewritten interpreter is slightly longer than the previous version, but perhaps slightly easier to read. For instance, execution of $(Seq\ c\ d)$ can be read: compute the execution of c, then compute the execution of d, then return nothing. Compare this with the previous version, which has the unnerving property that $exec\ d$ appears to the left of $exec\ c$.

One drawback of this program is that it introduces too much sequencing. Evaluation of $(Add\ t\ u)$ can be read: compute the evaluation of t, bind a to the result, then compute the evaluation of u, bind b to the result, then return $a + b$. This is unfortunate, in that it imposes a spurious ordering on the evaluation of t and u that was not present in the original program. The order does not matter because although $eval$ depends on the state, it does not change it. To remedy this we will augment the monad of state transformers M with a second monad M' of state readers.

4.3 Array readers

Recall that the monad of array transformers takes an initial array and returns a value and a final array.

$$
\begin{array}{lll}
\mathbf{type}\ \ M\ a & = & State \to (a, State) \\
\mathbf{type}\ \ State & = & Arr
\end{array}
$$

The corresponding monad of array readers takes an array and returns a value. No array is returned because it is assumed identical to the original array.

$$
\begin{array}{lll}
\textbf{type}\;\; M'\,a = \mathit{State} \to a & & \\
\mathit{unit}' & :: & a \to M'\,a \\
\mathit{unit}'\,a & = & \lambda x.\,a \\
(\star') & :: & M'\,a \to (a \to M'\,b) \to M'\,b \\
m \star' k & = & \lambda x.\,\textbf{let}\;\; a = m\,x \;\;\textbf{in}\;\; k\,a\,x \\
\mathit{fetch}' & :: & \mathit{Ix} \to M'\,\mathit{Val} \\
\mathit{fetch}'\,i & = & \lambda x.\,\mathit{index}\,i\,x
\end{array}
$$

The call $\mathit{unit}'\,a$ ignores the given state x and returns a. The call $m \star' k$ performs computation m in the given state x, yielding value a, then performs computation $k\,a$ in the same state x. Thus, unit' discards the state and $\star'$ duplicates it. The call $\mathit{fetch}'\,i$ returns the value in the given state x at index i.

Clearly, computations that only read the state are a subset of the computations that may read and write the state. Hence there should be a way to coerce a computation in monad M' into one in monad M.

$$
\begin{array}{lll}
\mathit{coerce} & :: & M'\,a \to M\,a \\
\mathit{coerce}\,m & = & \lambda x.\,\textbf{let}\;\; a = m\,x \;\;\textbf{in}\;\; (a, x)
\end{array}
$$

The call $\mathit{coerce}\,m$ performs computation m in the initial state x, yielding a, and returns a paired with state x. The function coerce enjoys a number of mathematical properties to be discussed shortly.

Again, these operations maintain single threading if suitably implemented. The definitions of $\star'$ and coerce must both be strict in the intermediate value a. This guarantees that when $\mathit{coerce}\,m$ is performed in state x, the computation of $m\,x$ will reduce to a form a that contains no extant pointers to the state x before the pair (a, x) is returned. Hence there will be only one pointer extant to the state whenever it is updated.

A monad is *commutative* if it satisfies the law

$$m \star \lambda a.\, n \star \lambda b.\, o = n \star \lambda b.\, m \star \lambda a.\, o.$$

The scope of a includes n on the right and not on the left, so this law is valid only when a does not appear free in n. Similarly, b must not appear free in m. In a commutative monad the order of computation does not matter.

The state reader monad is commutative, while the state transfomer monad is not. So no spurious order is imposed on computations in the state reader monad. In particular, the call $m \star' k$ may safely be implemented so that m and $k\,a$ are computed in parallel. However, the final result must still be strict in a. For instance, with the annotations used in the GRIP processor, $\star'$ could be defined as follows.

$$
\begin{array}{l}
m \star' k = \lambda x.\,\textbf{let}\;\; a = m\,x \;\;\textbf{in} \\
\qquad\qquad\quad \textbf{let}\;\; b = k\,a\,x \;\;\textbf{in} \\
\qquad\qquad\quad \mathit{par}\,a\,(\mathit{par}\,b\,(\mathit{seq}\,a\,b))
\end{array}
$$

The two calls to *par* spark parallel computations of a and b, and the call to *seq* waits for a to reduce to a non-bottom value before returning b.

These operations may be packaged into two abstract data types, M and M', supporting the eight operations *unit*, $\star$, *unit'*, $\star'$, *block*, *assign*, *fetch'*, and *coerce*. The abstraction guarantees single threading, so *assign* may be implemented by an in-place update.

The interpreter may be rewritten again.

$$
\begin{array}{lcl}
\mathit{eval} & :: & \mathit{Term} \to M'\ \mathit{Int} \\
\mathit{eval}\ (\mathit{Var}\ i) & = & \mathit{fetch}'\ i \\
\mathit{eval}\ (\mathit{Con}\ a) & = & \mathit{unit}'\ a \\
\mathit{eval}\ (\mathit{Add}\ t\ u) & = & \mathit{eval}\ t \star' \lambda a.\ \mathit{eval}\ u \star' \lambda b.\ \mathit{unit}'\ (a+b) \\
 & & \\
\mathit{exec} & :: & \mathit{Comm} \to M\ () \\
\mathit{exec}\ (\mathit{Asgn}\ i\ t) & = & \mathit{coerce}\ (\mathit{eval}\ t) \star \lambda a.\ \mathit{assign}\ i\ a \\
\mathit{exec}\ (\mathit{Seq}\ c\ d) & = & \mathit{exec}\ c \star \lambda().\ \mathit{exec}\ d \star \lambda().\ \mathit{unit}\ () \\
\mathit{exec}\ (\mathit{If}\ t\ c\ d) & = & \mathit{coerce}\ (\mathit{eval}\ t) \star \lambda a. \\
 & & \quad \mathbf{if}\ \ a = 0\ \ \mathbf{then}\ \ \mathit{exec}\ c\ \ \mathbf{else}\ \ \mathit{exec}\ d \\
 & & \\
\mathit{elab} & :: & \mathit{Prog} \to \mathit{Int} \\
\mathit{elab}\ (\mathit{Prog}\ c\ t) & = & \mathit{block}\ 0\ (\mathit{exec}\ c \star \lambda().\ \mathit{coerce}\ (\mathit{eval}\ t) \star \lambda a.\ \mathit{unit}\ a)
\end{array}
$$

This differs from the previous version in that *eval* is written in terms of M' rather than M, and calls to *coerce* surround the calls of *eval* in the other two functions. The new types make it clear that *eval* depends upon the state but does not alter it, while *exec* may both depend upon and alter the state.

The excessive sequencing of the previous version has been eliminated. In the evaluation of $(\mathit{Add}\ t\ u)$ the two subexpressions may be evaluated in either order or concurrently.

A *monad morphism* from a monad M' to a monad M is a function $h :: M'\ a \to M\ a$ that preserves the monad structure:

$$
\begin{array}{lcl}
h\ (\mathit{unit}'\ a) & = & \mathit{unit}\ a, \\
h\ (m \star' \lambda a.\ n) & = & (h\ m) \star \lambda a.\ (h\ n).
\end{array}
$$

It often happens that one wishes to use a combination of monads to achieve a purpose, and monad morphisms play the key role of converting from one monad to another [9].

In particular, *coerce* is a monad morphism, and it follows immediately from this that the two versions of the interpreter are equivalent.

4.4 Conclusion

How a functional language may provide in-place array update is an old problem. This section has presented a new solution, consisting of two abstract data types with eight operations between them. No change to the programming language is required, other than to provide an implementation of these types, perhaps as part of the standard prelude. The discovery of such a simple solution comes as a surpise, considering the plethora of more elaborate solutions that have been proposed.

A different way of expressing the same solution, based on continuation passing style, has subsequently been proposed by Hudak [8]. But Hudak's solution was inspired by the monad solution, and the monad solution still appears to have some small advantages [15].

Why was this solution not discovered twenty years ago? One possible reason is that the data types involve higher-order functions in an essential way. The usual axiomatisation of arrays involves only first-order functions, and so perhaps it did not occur to anyone to search for an abstract data type based on higher-order functions. That monads led to the discovery of the solution must count as a point in their favour.

5 Parsers

Parsers are the great success story of theoretical computing. The BNF formalism provides a concise and precise way to describe the syntax of a programming language. Mathematical tests can determine if a BNF grammar is ambiguous or vacuous. Transformations can produce an equivalent grammar that is easier to parse. Compiler-compilers can turn a high-level specification of a grammar into an efficient program.

This section shows how monads provide a simple framework for constructing recursive descent parsers. This is of interest in its own right, and also because the basic structures of parsing – sequencing and alternation – are fundamental to all of computing. It also provides a demonstration of how monads can model backtracking (or angelic non-determinism).

5.1 Lists

Our representation of parsers depends upon lists. Lists are ubiquitous in functional programming, and it is surprising that we have managed to get by so far while barely mentioning them. Actually, they have appeared in disguise, as strings are simply lists of characters.

We review some notation. We write $[a]$ for the type of a list with elements all of type a, and : for 'cons'. Thus $[1, 2, 3] = 1 : 2 : 3 : []$, and both have type $[\mathit{Int}]$. Strings are lists of characters, so *String* and $[\mathit{Char}]$ are equivalent, and *"monad"* is just an abbreviation for $[\text{'}m\text{'}, \text{'}o\text{'}, \text{'}n\text{'}, \text{'}a\text{'}, \text{'}d\text{'}]$.

It is perhaps not suprising that lists form a monad.

$$
\begin{array}{ll}
\mathit{unit} & :: \; a \to [a] \\
\mathit{unit}\; a & = [a] \\
(\star) & :: \; [a] \to (a \to [b]) \to [b] \\
[\,] \star k & = [\,] \\
(a : x) \star k & = k\; a +\!\!+ (x \star k)
\end{array}
$$

The call $unit\ a$ simply forms the unit list containing a. The call $m \star k$ applies k to each element of the list m, and appends together the resulting lists.

If monads encapsulate effects and lists form a monad, do lists correspond to some effect? Indeed they do, and the effect they correspond to is choice. One can think of a computation of type $[a]$ as offering a choice of values, one for each element of the list. The monadic equivalent of a function of type $a \to b$ is a function of type $a \to [b]$. This offers a choice of results for each argument, and hence corresponds to a relation. The operation *unit* corresponds to the identity relation, which associates each argument only with itself. If $k :: a \to [b]$ and $h :: b \to [c]$, then

$$\lambda a.\, k\ a \star \lambda b.\, h\ b :: a \to [c]$$

corresponds to the relational composition of k and h.

The *list comprehension* notation provides a convenient way of manipulating lists. The behaviour is analogous to set comprehensions, except the order is significant. For example,

$$\begin{aligned}
[\, sqr\ a \mid a \leftarrow [1, 2, 3]\,] &= [1, 4, 9] \\
[\, (a, b) \mid a \leftarrow [1, 2], b \leftarrow \text{“}list\text{”}\,] &= [(1, \text{‘}l\text{’}), (1, \text{‘}i\text{’}), (1, \text{‘}s\text{’}), (1, \text{‘}t\text{’}), \\
&\quad\ (2, \text{‘}l\text{’}), (2, \text{‘}i\text{’}), (2, \text{‘}s\text{’}), (2, \text{‘}t\text{’})]
\end{aligned}$$

The list comprehension notation translates neatly into monad operations.

$$\begin{aligned}
[\, t \mid x \leftarrow u\,] &= u \star \lambda x.\, unit\ t \\
[\, t \mid x \leftarrow u,\ y \leftarrow v\,] &= u \star \lambda x.\, v \star \lambda y.\, unit\ t
\end{aligned}$$

Here t is an expression, x and y are variables (or more generally patterns), and u and v are expressions that evaluate to lists. Connections between comprehensions and monads have been described at length elsewhere [21].

5.2 Representing parsers

Parsers are represented in a way similar to state transformers.

$$\begin{aligned}
&\textbf{type}\ \ M\ a\ \ = State \to [(a, State)] \\
&\textbf{type}\ \ State = String
\end{aligned}$$

That is, the parser for type a takes a state representing a string to be parsed, and returns a *list* of containing the value of type a parsed from the string, and a state representing the remaining string yet to be parsed. The list represents all the alternative possible parses of the input state: it will be empty if there is no parse, have one element if there is one parse, have two elements if there are two different possible parses, and so on.

Consider a simple parser for arithmetic expressions, which returns a tree of the type considered previously.

$$\textbf{data}\ \ Term = Con\ Int \mid Div\ Term\ Term$$

Say we have a parser for such terms.

$$\mathit{term} :: M\ \mathit{Term}$$

Here are some examples of its use.

$$\begin{array}{ll}
\mathit{term}\ \text{``}\mathit{23}\text{''} & = [(\mathit{Con}\ \mathit{23},\ \text{``\,''})] \\
\mathit{term}\ \text{``}\mathit{23\ \ and\ more}\text{''} & = [(\mathit{Con}\ \mathit{23},\ \text{``}\ \mathit{and\ more}\text{''})] \\
\mathit{term}\ \text{``}\mathit{not\ a\ term}\text{''} & = [\,] \\
\mathit{term}\ \text{``}((\mathit{1972} \div \mathit{2}) \div \mathit{23})\text{''} & = [(\mathit{Div}\ (\mathit{Div}\ (\mathit{Con}\ \mathit{1972})\ (\mathit{Con}\ \mathit{2}))\ (\mathit{Con}\ \mathit{23})),\ \text{``\,''})]
\end{array}$$

A parser m is *unambiguous* if for every input x the list of possible parses $m\,x$ is either empty or has exactly one item. For instance, *term* is unambiguous. An ambiguous parser may return a list with two or more alternative parsings.

5.3 Parsing an item

The basic parser returns the first item of the input, and fails if the input is exhausted.

$$\begin{array}{ll}
\mathit{item} & :: M\ \mathit{Char} \\
\mathit{item}\ [\,] & = [\,] \\
\mathit{item}\ (a : x) & = [(a, x)]
\end{array}$$

Here are two examples.

$$\begin{array}{ll}
\mathit{item}\ \text{``\,''} & = [\,] \\
\mathit{item}\ \text{``}\mathit{monad}\text{''} & = [(\text{`}m\text{'},\ \text{``}\mathit{onad}\text{''})]
\end{array}$$

Clearly, *item* is unambiguous.

5.4 Sequencing

To form parsers into a monad, we require operations *unit* and $\star$.

$$\begin{array}{ll}
\mathit{unit} & :: a \to M\ a \\
\mathit{unit}\ a\ x & = [(a, x)] \\
(\star) & :: M\ a \to (a \to M\ b) \to M\ b \\
(m \star k)\ x & = [(b, z) \mid (a, y) \leftarrow m\,x,\ (b, z) \leftarrow k\ a\ y]
\end{array}$$

The parser $\mathit{unit}\ a$ accepts input x and yields one parse with value a paired with remaining input x. The parser $m \star k$ takes input x; parser m is applied to input x yielding for each parse a value a paired with remaining input y; then parser $k\ a$ is applied to input y, yielding for each parse a value b paired with final remaining output z.

Thus, *unit* corresponds to the empty parser, which consumes no input, and $\star$ corresponds to sequencing of parsers.

Two items may be parsed as follows.

$$\begin{array}{l}
\mathit{twoItems} :: M\ (\mathit{Char}, \mathit{Char}) \\
\mathit{twoItems} = \mathit{item} \star \lambda a.\ \mathit{item} \star \lambda b.\ \mathit{unit}\ (a, b)
\end{array}$$

Here are two examples.

$$\begin{aligned} &\mathit{twoItems}\ \text{“}m\text{”} &&= [\,] \\ &\mathit{twoItems}\ \text{“}monad\text{”} &&= [((\text{‘}m\text{’}, \text{‘}o\text{’}), \text{“}nad\text{”})] \end{aligned}$$

The parse succeeds only if there are at least two items in the list.

The three monad laws express that the empty parser is an identity for sequencing, and that sequencing is associative.

$$\begin{aligned} \mathit{unit}\ a \star \lambda b.\, n &= n[a/b] \\ m \star \lambda a.\, \mathit{unit}\ a &= m \\ m \star (\lambda a.\, n \star \lambda b.\, o) &= (m \star \lambda a.\, n) \star \lambda b.\, o \end{aligned}$$

If m is unambiguous and $k\,a$ is unambiguous for every a, then $m \star k$ is also unambiguous.

5.5 Alternation

Parsers may also be combined by alternation.

$$\begin{aligned} &\mathit{zero} &&:: M\ a \\ &\mathit{zero}\ x &&= [\,] \\ &(\oplus) &&:: M\ a \to M\ a \to M\ a \\ &(m \oplus n)\ x &&= m\,x \mathbin{+\!\!+} n\,x \end{aligned}$$

The parser *zero* takes input x and always fails. The parser $m \oplus n$ takes input x and yields all parses of m applied to input x and all parses of n applied to the same input x.

Here is a parser that parses one or two items from the input.

$$\begin{aligned} &\mathit{oneOrTwoItems} :: M\ \mathit{String} \\ &\mathit{oneOrTwoItems} = \quad\ (\mathit{item} \star \lambda a.\, \mathit{unit}\,[a]) \\ &\qquad\qquad\qquad\quad \oplus (\mathit{item} \star \lambda a.\, \mathit{item} \star \lambda b.\, \mathit{unit}\,[a, b]) \end{aligned}$$

Here are three examples.

$$\begin{aligned} &\mathit{oneOrTwoItems}\ \text{“ ”} &&= [\,] \\ &\mathit{oneOrTwoItems}\ \text{“}m\text{”} &&= [(\text{“}m\text{”}, \text{“ ”})] \\ &\mathit{oneOrTwoItems}\ \text{“}monad\text{”} &&= [(\text{“}m\text{”}, \text{“}onad\text{”}), (\text{“}mo\text{”}, \text{“}nad\text{”})] \end{aligned}$$

The last yields two alternative parses, showing that alternation can yield ambiguous parsers.

The parser that always fails is the identity for alternation, and alternation is associative.

$$\begin{aligned} \mathit{zero} \oplus n &= n \\ m \oplus \mathit{zero} &= m \\ m \oplus (n \oplus o) &= (m \oplus n) \oplus o \end{aligned}$$

Furthermore, *zero* is indeed a zero of $\star$, and $\star$ distributes through $\oplus$.

$$\begin{aligned} \mathit{zero} \star k &= \mathit{zero} \\ m \star \lambda a.\, \mathit{zero} &= \mathit{zero} \\ (m \oplus n) \star k &= (m \star k) \oplus (n \star k) \end{aligned}$$

It is *not* the case that $\star$ distributes rightward through $\oplus$ only because we are representing alternative parses by an ordered list; if we used an unordered bag, then $m \star \lambda a.\, (k\ a \oplus h\ a) = (m \star k) \oplus (m \star h)$ would also hold. An unambiguous parser yields a list of length at most one, so the order is irrelevant, and hence this law also holds whenever either side is unambiguous (which implies that both sides are).

5.6 Filtering

A parser may be filtered by combining it with a predicate.

$$\begin{aligned} (\triangleright) &:: M\ a \to (a \to \mathit{Bool}) \to M\ a \\ m \triangleright p &= m \star \lambda a.\, \mathbf{if}\ \ p\ a\ \ \mathbf{then}\ \ \mathit{unit}\ a\ \ \mathbf{else}\ \ \mathit{zero} \end{aligned}$$

Given a parser m and a predicate on values p, the parser $m \triangleright p$ applies parser m to yield a value a; if $p\ a$ holds it succeeds with value a, otherwise it fails. Note that filtering is written in terms of previously defined operators, and need not refer directly to the state.

Let *isLetter* and *isDigit* be the obvious predicates. Here are two parsers.

$$\begin{aligned} \mathit{letter} &:: M\ \mathit{Char} \\ \mathit{letter} &= \mathit{item} \triangleright \mathit{isLetter} \end{aligned}$$

$$\begin{aligned} \mathit{digit} &:: M\ \mathit{Int} \\ \mathit{digit} &= (\mathit{item} \triangleright \mathit{isDigit}) \star \lambda a.\, \mathit{unit}\,(\mathit{ord}\ a - \mathit{ord}\ \text{'}\mathit{0}\text{'}) \end{aligned}$$

The first succeeds only if the next input item is a letter, and the second succeeds only if it is a digit. The second also converts the digit to its corresponding value, using $\mathit{ord} :: \mathit{Char} \to \mathit{Int}$ to convert a character to its ASCII code. Assuming that $\triangleright$ has higher precedence than $\star$ would allow some parentheses to be dropped from the second definition.

A parser for a literal recognises a single specified character.

$$\begin{aligned} \mathit{lit} &:: \mathit{Char} \to M\ \mathit{Char} \\ \mathit{lit}\ c &= \mathit{item} \triangleright (\lambda a.\, a == c) \end{aligned}$$

The parser *lit c* succeeds if the input begins with character c, and fails otherwise.

$$\begin{aligned} \mathit{lit}\ \text{'}m\text{'}\ \text{“}\mathit{monad}\text{”} &= [(\text{'}m\text{'}, \text{“}\mathit{onad}\text{”})] \\ \mathit{lit}\ \text{'}m\text{'}\ \text{“}\mathit{parse}\text{”} &= [\,] \end{aligned}$$

From the previous laws, it follows that filtering preserves zero and distributes through alternation.

$$\begin{aligned} \mathit{zero} \triangleright p &= \mathit{zero} \\ (m \oplus n) \triangleright p &= (m \triangleright p) \oplus (n \triangleright p) \end{aligned}$$

If m is an unambiguous parser, so is $m \triangleright p$.

5.7 Iteration

A single parser may be iterated, yielding a list of parsed values.

$$
\begin{aligned}
&\mathit{iterate} && :: M\ a \rightarrow M\,[a] \\
&\mathit{iterate}\ m = && \quad (m \star \lambda a.\ \mathit{iterate}\ m \star \lambda x.\ \mathit{unit}\,(a : x)) \\
& && \oplus\ \mathit{unit}\,[\,]
\end{aligned}
$$

Given a parser m, the parser *iterate* m applies parser m in sequence zero or more times, returning a list of all the values parsed. In the list of alternative parses, the longest parse is returned first.

Here is an example.

$$
\begin{aligned}
\mathit{iterate\ digit}\ \text{“}\mathit{23\ \ and\ more}\text{”} = [&([\mathit{2}, \mathit{3}],\ \text{“}\ \mathit{and\ more}\text{”}), \\
&([\mathit{2}],\ \text{“}\mathit{3\ \ and\ more}\text{”}), \\
&([\,],\ \text{“}\mathit{23\ \ and\ more}\text{”})]
\end{aligned}
$$

Here is one way to parse a number.

$$
\begin{aligned}
&\mathit{number} :: M\ \mathit{Int} \\
&\mathit{number} = \mathit{digit} \star \lambda a.\ \mathit{iterate\ digit} \star \lambda x.\ \mathit{unit}\,(\mathit{asNumber}\,(a : x))
\end{aligned}
$$

Here *asNumber* takes a list of one or more digits and returns the corresponding number. Here is an example.

$$
\begin{aligned}
\mathit{number}\ \text{“}\mathit{23\ \ and\ more}\text{”} = [&(\mathit{23},\ \text{“}\ \mathit{and\ more}\text{”}), \\
&(\mathit{2},\ \text{“}\mathit{3\ \ and\ more}\text{”})]
\end{aligned}
$$

This supplies two possible parses, one which parses both digits, and one which parses only a single digit. A number is defined to contain at least one digit, so there is no parse with zero digits.

As this last example shows, often it is more natural to design an iterator to yield only the longest possible parse. The next section describes a way to achieve this.

5.8 Biased choice

Alternation, written $m \oplus n$, yields all parses yielded by m followed by all parses yielded by n. For some purposes, it is more sensible to choose one or the other: all parses by m if there are any, and all parses by n otherwise. This is called biased choice.

$$
\begin{aligned}
&(\oslash) && :: M\ a \rightarrow M\ a \rightarrow M\ a \\
&(m \oslash n)\ x = && \mathbf{if}\ \ m\,x \neq [\,]\ \ \mathbf{then}\ \ m\,x\ \ \mathbf{else}\ \ n\,x
\end{aligned}
$$

Biased choice, written $m \oslash n$, yields the same parses as m, unless m fails to yield any parse, in which case it yields the same parses as n.

Here is iteration, rewritten with biased choice.

$$\begin{aligned}
&\mathit{reiterate} &&:: M\ a \to M\,[a] \\
&\mathit{reiterate}\ m = && (m \star \lambda a.\, \mathit{reiterate}\ m \star \lambda x.\, \mathit{unit}\,(a : x)) \\
& && \oslash\ \mathit{unit}\,[\,]
\end{aligned}$$

The only difference is to replace $\oplus$ with $\oslash$. Instead of yielding a list of all possible parses with the longest first, this yields only the longest possible parse.

Here is the previous example revisited.

$$\mathit{reiterate\ digit\ “23\ \ and\ more”} = [([2, 3], \text{“ } \mathit{and\ more}\text{”})]$$

In what follows, *number* is taken to be rewritten with *reiterate*.

$$\begin{aligned}
&\mathit{number} :: M\ \mathit{Int} \\
&\mathit{number} = \mathit{digit} \star \lambda a.\, \mathit{reiterate\ digit} \star \lambda x.\, \mathit{unit}\,(\mathit{asNumber}\,(a : x))
\end{aligned}$$

Here is an example that reveals a little of how ambiguous parsers may be used to search a space of possibilities. We use *reiterate* to find all ways of taking one or two items from a string, zero or more times.

$$\begin{aligned}
\mathit{reiterate\ oneOrTwoItems\ “many”} = [&([\text{“}m\text{”}, \text{“}a\text{”}, \text{“}n\text{”}, \text{“}y\text{”}], \text{“ ”}), \\
&([\text{“}m\text{”}, \text{“}a\text{”}, \text{“}ny\text{”}], \text{“ ”}), \\
&([\text{“}m\text{”}, \text{“}an\text{”}, \text{“}y\text{”}], \text{“ ”}), \\
&([\text{“}ma\text{”}, \text{“}n\text{”}, \text{“}y\text{”}], \text{“ ”}), \\
&([\text{“}ma\text{”}, \text{“}ny\text{”}], \text{“ ”})]
\end{aligned}$$

This combines alternation (in *oneOrTwoItems*) with biased choice (in *reiterate*). There are several possible parses, but for each parse *oneOrTwoItems* has been applied until the entire input has been consumed. Although this example is somewhat fanciful, a similar technique could be used to find all ways of breaking a dollar into nickels, dimes, and quarters.

If m and n are unambiguous, then $m \oslash n$ and $\mathit{reiterate}\ m$ are also unambiguous. For unambiguous parsers, sequencing distributes right through biased choice:

$$(m \star k) \oslash (m \star h) = m \star \lambda a.\, k\ a \oslash h\ a$$

whenever m is unambiguous. Unlike with alternation, sequencing does not distribute left through biased choice, even for unambiguous parsers.

5.9 A parser for terms

It is now possible to write the parser for terms alluded to at the beginning. Here is a grammar for fully parenthesised terms, expressed in BNF.

$$\mathit{term} ::= \mathit{number} \mid \text{‘('}\ \mathit{term}\ \text{‘}\div\text{'}\ \mathit{term}\ \text{‘)'}$$

This translates directly into our notation as follows. Note that our notation, unlike BNF, specifies exactly how to construct the returned value.

$$
\begin{array}{lll}
\mathit{term} :: M\ \mathit{Term} & & \\
\mathit{term} = & (\mathit{number} & \star\ \lambda a. \\
 & \quad \mathit{unit}\,(\mathit{Con}\ a)) & \\
 & \oplus\ (\mathit{lit}\ \text{`('} & \star\ \lambda\_. \\
 & \quad \mathit{term} & \star\ \lambda t. \\
 & \quad \mathit{lit}\ \text{`}\div\text{'} & \star\ \lambda\_. \\
 & \quad \mathit{term} & \star\ \lambda u. \\
 & \quad \mathit{lit}\ \text{`)'} & \star\ \lambda\_. \\
 & \quad \mathit{unit}\,(\mathit{Div}\ t\ u)) &
\end{array}
$$

(Here $\lambda\_.\ e$ is equivalent to $\lambda x.\ e$ where x is some fresh variable that does not appear in e; it indicates that the value bound by the lambda expression is not of interest.) Examples of the use of this parser appeared earlier.

The above parser is written with alternation, but as it is unambiguous, it could just as well have been written with biased choice. The same is true for all the parsers in the next section.

5.10 Left recursion

The above parser works only for fully parenthesised terms. If we allow unparenthesised terms, then the operator $\div$ should associate to the left. The usual way to express such a grammar in BNF is as follows.

$$
\begin{array}{lcl}
\mathit{term} & ::= & \mathit{term}\ \text{`}\div\text{'}\,\mathit{factor} \mid \mathit{factor} \\
\mathit{factor} & ::= & \mathit{number} \mid \text{`('}\,\mathit{term}\ \text{`)'}
\end{array}
$$

This translates into our notation as follows.

$$
\begin{array}{llll}
\mathit{term} & :: & M\ \mathit{Term} & \\
\mathit{term} & = & (\mathit{term} & \star\ \lambda t. \\
 & & \quad \mathit{lit}\ \text{`}\div\text{'} & \star\ \lambda\_. \\
 & & \quad \mathit{factor} & \star\ \lambda u. \\
 & & \quad \mathit{unit}\,(\mathit{Div}\ t\ u)) & \\
 & & \oplus\ \mathit{factor} & \\[6pt]
\mathit{factor} & :: & M\ \mathit{Term} & \\
\mathit{factor} & = & (\mathit{number} & \star\ \lambda a. \\
 & & \quad \mathit{unit}\,(\mathit{Con}\ a)) & \\
 & & \oplus\ (\mathit{lit}\ \text{`('} & \star\ \lambda\_. \\
 & & \quad \mathit{term} & \star\ \lambda t. \\
 & & \quad \mathit{lit}\ \text{`)'} & \star\ \lambda\_. \\
 & & \quad \mathit{unit}\ t) &
\end{array}
$$

There is no problem with *factor*, but any attempt to apply *term* results in an infinite loop. The problem is that the first step of *term* is to apply *term*, leading to an infinite regress. This is called the *left recursion problem.* It is a difficulty for all recursive descent parsers, functional or otherwise.

The solution is to rewrite the grammar for *term* in the following equivalent form.

$$\begin{array}{lcl} \mathit{term} & ::= & \mathit{factor}\ \mathit{term}' \\ \mathit{term}' & ::= & \text{'}\div\text{'}\mathit{factor}\ \mathit{term}' \mid \mathit{unit} \end{array}$$

where as usual *unit* denotes the empty parser. This then translates directly into our notation.

$$\begin{array}{lcl} \mathit{term} & :: & M\ \mathit{Term} \\ \mathit{term} & = & \mathit{factor} \star \lambda t.\ \mathit{term}'\ t \end{array}$$

$$\begin{array}{lcll} \mathit{term}' & :: & \mathit{Term} \rightarrow M\ \mathit{Term} & \\ \mathit{term}'\ t & = & \quad (\mathit{lit}\ \text{'}\div\text{'} & \star\ \lambda\_. \\ & & \quad\ \mathit{factor} & \star\ \lambda u. \\ & & \quad\ \mathit{term}'\ (\mathit{Div}\ t\ u)) & \\ & & \oplus\ \ \mathit{unit}\ t & \end{array}$$

Here term' parses the remainder of a term; it takes an argument corresponding to the term parsed so far.

This has the desired effect.

$$\begin{array}{lcl} \mathit{term}\ \text{“}1972 \div 2 \div 23\text{”} & = & [((\mathit{Div}\ (\mathit{Div}\ (\mathit{Con}\ 1972)\ (\mathit{Con}\ 2))\ (\mathit{Con}\ 23)),\ \text{“ ”})] \\ \mathit{term}\ \text{“}1972 \div (2 \div 23)\text{”} & = & [((\mathit{Div}\ (\mathit{Con}\ 1972)\ (\mathit{Div}\ (\mathit{Con}\ 2)\ (\mathit{Con}\ 23))),\ \text{“ ”})] \end{array}$$

In general, the left-recursive definition

$$m = (m \star k) \oplus n$$

can be rewritten as

$$m = n \star (\mathit{closure}\ k)$$

where

$$\begin{array}{lcl} \mathit{closure} & :: & (a \rightarrow M\ a) \rightarrow (a \rightarrow M\ a) \\ \mathit{closure}\ k\ a & = & (k\ a \star \mathit{closure}\ k) \oplus \mathit{unit}\ a \end{array}$$

Here $m :: M\ a$, $n :: M\ a$, and $k :: a \rightarrow M\ a$.

5.11 Improving laziness

Typically, a program might be represented as a function from a list of characters – the input – to another list of characters – the output. Under lazy evaluation, usually only some of the input need be read before the first part of the output list is produced. This ‘on line’ behavior is essential for some purposes.

In general, it is unreasonable to expect such behaviour from a parser, since in general it cannot be known that the input will be successfully parsed until all of it is read. However, in certain special cases one may hope to do better.

Consider applying *reiterate* m to a string beginning with an instance of m. In this case, the parse cannot fail: regardless of the remainder of the string, one would expect the parse yielded to be a list beginning with the parsed value. Under lazy evaluation, one might expect to be able to generate output corresponding to the first digit before the remaining input has been read.

But this is not what happens: the parser reads the entire input before any output is generated. What is necessary is some way to encode that the parser *reiterate* m always succeeds. (Even if the beginning of the input does not match m, it will yield as a value the empty list.) This is provided by the function *guarantee*.

$$\begin{array}{lcl} \mathit{guarantee} & :: & M\ a \rightarrow M\ a \\ \mathit{guarantee}\ m\ x & = & \mathbf{let}\ \ u = m\ x\ \ \mathbf{in}\ \ (\mathit{fst}\,(\mathit{head}\ u), \mathit{snd}\,(\mathit{head}\ u)) : \mathit{tail}\ u \end{array}$$

Here $\mathit{fst}\,(a, b) = a$, $\mathit{snd}\,(a, b) = b$, $\mathit{head}\,(a : x) = a$, and $\mathit{tail}\,(a : x) = x$.

Here is *reiterate* with the guarantee added.

$$\begin{array}{lcl} \mathit{reiterate} & :: & M\ a \rightarrow M\,[a] \\ \mathit{reiterate}\ m & = & \mathit{guarantee}\,(\quad (m \star \lambda a.\, \mathit{reiterate}\ m \star \lambda x.\, \mathit{unit}\,(a : x)) \\ & & \qquad\qquad\quad \oslash\ \mathit{unit}\,[\,]) \end{array}$$

This ensures that *reiterate* m and all of its recursive calls return a list with at least one answer. As a result, the behaviour under lazy evaluation is much improved.

The preceding explanation is highly operational, and it is worth noting that denotational semantics provides a useful alternative approach. Let $\perp$ denote a program that does not terminate. One can verify that with the old definition

$$\mathit{reiterate}\ \mathit{digit}\,(\text{‘}\mathit{1}\text{’} : \perp) = \perp$$

while with the new definition

$$\mathit{reiterate}\ \mathit{digit}\,(\text{‘}\mathit{1}\text{’} : \perp) = ((\text{‘}\mathit{1}\text{’} : \perp), \perp) : \perp$$

Thus, given that the input begins with the character *‘1’* but that the remainder of the input is unknown, with the old definition nothing is known about the output, while with the new definition it is known that the output yields at least one parse, the value of which is a list which begins with the character *‘1’*.

Other parsers can also benefit from a judicious use of *guarantee*, and in particular *iterate* can be modified like *reiterate*.

5.12 Conclusion

We have seen that monads provide a useful framework for structuring recursive descent parsers. The empty parser and sequencing correspond directly to *unit* and $\star$, and the monads laws reflect that sequencing is associative and has the empty parser as a unit. The failing parser and alternation correspond to *zero* and $\oplus$, which satisfy laws reflecting that alternation is associative and has the failing parser as a unit, and that sequencing distributes through alternation.

Sequencing and alternation are fundamental not just to parsers but to much of computing. If monads capture sequencing, then it is reasonable to ask: what captures both sequencing and alternation? It may be that *unit*, $\star$, *zero*, and $\oplus$, together with the laws above, provide such a structure. Further experiments are needed. One hopeful indication is that a slight variation of the parser monad yields a plausible model of Dijkstra's guarded command language.

References

1. S. Abramsky and C. Hankin, *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.
2. A. Bloss, Update analysis and the efficient implementation of functional aggregates. In *4'th Symposium on Functional Programming Languages and Computer Architecture*, ACM, London, September 1989.
3. R. Bird and P. Wadler, *Introduction to Functional Programming.* Prentice Hall, 1987.
4. P. Hudak, S. Peyton Jones and P. Wadler, editors, *Report on the Programming Language Haskell: Version 1.1.* Technical report, Yale University and Glasgow University, August 1991.
5. J.-Y. Girard, Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
6. J. Guzmán and P. Hudak, Single-threaded polymorphic lambda calculus. In *IEEE Symposium on Logic in Computer Science*, Philadelphia, June 1990.
7. P. Hudak, A semantic model of reference counting and its abstraction (detailed summary). In *ACM Conference on Lisp and Functional Programming*, pp. 351–363, Cambridge, Massachusetts, August 1986.
8. P. Hudak, Continuation-based mutable abstract data types, or how to have your state and munge it too. Technical report YALEU/DCS/RR-914, Department of Computer Science, Yale University, July 1992.
9. D. King and P. Wadler, Combining monads. In *Glasgow Workshop on Functional Programming*, Ayr, July 1992. Workshops in Computing Series, Springer Verlag, to appear.
10. S. Mac Lane, *Categories for the Working Mathematician*, Springer-Verlag, 1971.
11. R. Milner, M. Tofte, and R. Harper, *The definition of Standard ML.* MIT Press, 1990.
12. L. C. Paulson, *ML for the Working Programmer.* Cambridge University Press, 1991.
13. E. Moggi, Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, Asilomar, California; IEEE, June 1989. (A longer version is available as a technical report from the University of Edinburgh.)
14. E. Moggi, An abstract view of programming languges. Course notes, University of Edinburgh.
15. S. L. Peyton Jones and P. Wadler, Imperative functional programming. In *20'th Symposium on Principles of Programming Languages*, Charleston, South Carolina; ACM, January 1993.
16. G. Plotkin, Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
17. J. Rees and W. Clinger (eds.), The revised^{3} report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, 1986.
18. D. Schmidt, Detecting global variables in denotational specifications. *ACM Trans. on Programming Languages and Systems*, 7:299–310, 1985.

19. V. Swarup, U. S. Reddy, and E. Ireland, Assignments for applicative languages. In *Conference on Functional Programming Languages and Computer Architecture*, Cambridge, Massachusetts; LNCS 523, Springer Verlag, August 1991.
20. D. A. Turner, An overview of Miranda. In D. A. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.
21. P. Wadler, Comprehending monads. In *Conference on Lisp and Functional Programming*, Nice, France; ACM, June 1990.
22. Is there a use for linear logic? *Conference on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, New Haven, Connecticut; ACM, June 1991.
23. P. Wadler, The essence of functional programming (invited talk). In *19'th Symposium on Principles of Programming Languages*, Albuquerque, New Mexico; ACM, January 1992.