Generalising Monads to Arrows

John Hughes

November 10, 1998

1 Introduction

One of the distinguishing features of functional programming is the widespread use of *combinators* to construct programs. A combinator is a function which builds program fragments from program fragments; in a sense the programmer using combinators constructs much of the desired program automatically, rather than writing every detail by hand. The freedom that functional languages provide to manipulate functions — program fragments — as first-class citizens supports combinator programming directly.

Some combinators, such as the well-known list-processing operators *map* and *filter*, encapsulate generally useful program constructions and may appear in almost any functional program. Others are tailored to particular application areas, and are often collected into libraries that enable applications in that area to be built quickly and easily. For example, *parsing* is an application area that has been extensively studied. Given an appropriate library of parsing combinators, a parser for the grammar

$$G ::= a\ G\ b \mid c$$

might be programmed in Haskell [Hud92, PH96] as

$$gram = symbol\ \text{“a”}\ \text{`cat`}\ gram\ \text{`cat`}\ symbol\ \text{“b”} +\!\!+\!\!+ symbol\ \text{“c”}$$

A note on syntax: in Haskell, function application is written without brackets, so *symbol* “a” denotes a call of the function *symbol* with argument “a”, and any function of two arguments may be used as an infix operator by enclosing it in back-quotes. In this example, *symbol* is a function which constructs a parser that accepts just the given token, `cat` is a binary operator which combines two parsers into a parser that runs both in sequence, +++ is a binary operator which combines two parsers into one which tries both as alternatives[1], and the entire declaration is a recursive definition of a parser *gram* which recognises the non-terminal G.

[1] We follow the fairly widespread convention that long operator names are typeset with their characters overlapping, so that they look like a single name. In reality, +++ is just `+++`.

Although the idea of programming with combinators is quite old, the design of combinator libraries has been profoundly influenced in recent years by Wadler's introduction of the concept of a *monad* into functional programming [Wad90, Wad92, Wad95]. We shall discuss monads much more fully in the next section, but for now, suffice it to say that a monad is a kind of standardised interface to an abstract data type of 'program fragments'. The monad interface has been found to be suitable for many combinator libraries, and is now extensively used. Numerous benefits flow from using a common interface: to take just one example, Haskell has been extended with special constructions to make the use of monads particularly convenient.

It is therefore a matter for some concern when libraries emerge which cannot, for fundamental reasons, use the monad interface. In particular, Swierstra and Duponcheel have developed a very interesting library for parsing LL-1 grammars [SD96], that avoids a well-known inefficiency in monadic parsing libraries by combining the construction of a parser with a 'static analysis' of the program so constructed. Yet Swierstra and Duponcheel's optimisation is incompatible with the monad interface. We believe that their library is not just an isolated example, but demonstrates a generally useful paradigm for combinator design that falls outside the world of monads. We shall look more closely at their idea in section 3.

Inspired by Swierstra and Duponcheel's library, I sought a generalisation of the monad concept that could also offer a standardised interface to libraries of this new type. My proposal, which I call *arrows*, is the subject of this paper. Pleasingly, the arrow interface turned out to be applicable to other kinds of non-monadic library also, for example the *fudgets* library for graphical user interfaces [CH93], and a new library for programming active web pages. These applications will be described in sections 6 and 9.

While arrows are a little less convenient to use than monads, they have significantly wider applicability. They can therefore be used to bring the benefits of monad-like programming to a much wider class of applications.

2 Background: Library Design Using Monads

What, then, is a monad? In Haskell, the monad interface can be defined as a *class*:

$$\begin{aligned}&\textbf{class } \mathit{Monad}\ m \textbf{ where}\\ &\quad \mathit{return} :: a \to m\ a\\ &\quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b\end{aligned}$$

Read this as follows: a parameterised type m is a monad if it supports the two operations *return* and $\gg\!=$ (pronounced 'bind') with the types given. Intuitively, we think of a value of type $m\ a$ as representing a *computation* with result of type a — a program fragment. The *return* operation constructs a trivial computation

that just delivers its argument as its result. The $\gg=$ operation combines two computations in sequence, passing the result of the first as an argument to the second — hence the type of the second argument of $\gg=$: it is a function that constructs the second computation, rather than just a computation.

2.1 An Example: A Monad to Manage Failures

For example, consider the type *Maybe* defined by

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}$$

(This declaration introduces a new parameterised type *Maybe* with two constructors, *Just* and *Nothing*. A value of type $\mathit{Maybe}\ a$ is either of the form $\mathit{Just}\ x$, where x is a component of type a, or of the form *Nothing*.) This type can be used to represent possible failure: a function which intuitively returns a result of type t, but may fail, can be defined to return a result of type $\mathit{Maybe}\ t$ instead, where *Nothing* represents failure. This idea can be used more conveniently if we define a combinator library to take care of failure handling.

To do so, we declare the type *Maybe* to be a monad; that is, we give implementations of *return* and $\gg=$ for this type. In Haskell, we write

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ \mathit{Maybe}\ \textbf{where}\\ \quad \mathit{return}\ a = \mathit{Just}\ a\\ \quad x \gg= f = \ \textbf{case}\ x\ \textbf{of}\\ \qquad\qquad\quad \mathit{Just}\ a \quad \to \quad f\ a\\ \qquad\qquad\quad \mathit{Nothing} \quad \to \quad \mathit{Nothing}\end{array}$$

where $x \gg= f$ fails immediately, without calling f, if its first argument x fails.

Using these combinators we can write functions which handle failure properly without any explicit tests for *Just* and *Nothing*. For example, the following function adds together two possibly-failing integers, failing itself if either argument does:

$$\begin{array}{l}\mathit{add} :: \mathit{Maybe}\ \mathit{Int} \to \mathit{Maybe}\ \mathit{Int} \to \mathit{Maybe}\ \mathit{Int}\\ \mathit{add}\ x\ y = \ x \gg= \lambda a \to\\ \qquad\qquad\quad\ y \gg= \lambda b \to\\ \qquad\qquad\quad\ \mathit{return}\ (a + b)\end{array}$$

(The layout here is well suited to monadic programs, but may be confusing at first: the body of the λ-expression $\lambda a \to \ldots$ extends to the *end* of the entire right hand side!)

To complete a useful library for failure handling we must add at least a combinator to cause a failure, for example

$$\begin{array}{l}\mathit{fail} :: \mathit{Maybe}\ a\\ \mathit{fail} = \mathit{Nothing}\end{array}$$

Now we can treat the *Maybe* type as abstract, and write programs that cause and propagate failures just using the operators *fail*, *return* and $\gg=$, without any explicit dependence on the way that failures are represented.

2.2 Another Example: A Monad to Manage State

As another example, an updateable state can be modelled in a purely functional language by passing each function the current contents of the state as an additional parameter, and returning the possibly modified state as a part of each function's result. To do so by hand is tedious and error-prone, but fortunately we can encapsulate the state passing mechanism in a combinator library by using a monad.

In this case we represent a computation with result type a and a state of type s by a value of the type

$$\textbf{newtype}\ \mathit{StateMonad}\ s\ a = \mathit{SM}\ (s \to (a, s))$$

(The Haskell **newtype** declaration introduces a new type isomorphic to an existing one, where the constructor names the isomorphism). For any state type s, the partially applied type $\mathit{StateMonad}\ s$ (which denotes a parameterised type with one remaining parameter) is a monad:

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ (\mathit{StateMonad}\ s)\ \textbf{where}\\ \quad \mathit{return}\ a = \mathit{SM}\ (\lambda s \to (a, s))\\ \quad x \gg= f = \mathit{SM}\ (\lambda s \to\ \textbf{let}\ \mathit{SM}\ x' = x\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad (a, s') = x'\ s\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{SM}\ f' = f\ a\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad (b, s'') = f'\ s'\\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (b, s''))\end{array}$$

With these definitions, we can write programs which pass around a state just in terms of *return* and $\gg=$; there is no need to manipulate the state explicitly. Notice that $\gg=$ must pass the modified state s' returned by its first argument to its second, rather than the original state, and must return the modified state returned by its second argument as part of its own result. If one attempts to pass a state around by hand, rather than by using combinators, then it is very easy to forget a $'$ somewhere, with strange bugs as a result.

To complete a library for state passing we must provide combinators for reading and modifying the state. For example,

$$\begin{array}{l}\mathit{fetch} :: \mathit{StateMonad}\ s\ s\\ \mathit{fetch} = \mathit{SM}\ (\lambda s \to (s, s))\\ \\ \mathit{store} :: s \to \mathit{StateMonad}\ s\ ()\\ \mathit{store}\ x = \mathit{SM}\ (\lambda s \to ((), x))\end{array}$$

Now the *StateMonad* type can be made abstract, and stateful programs can be written just in terms of the combinators. For example, a function to increment a counter:

$$\begin{aligned} &\mathit{tick} :: \mathit{StateMonad}\ \mathit{Int}\ \mathit{Int} \\ &\mathit{tick} = \ \mathit{fetch} \gg\!= \lambda n \rightarrow \\ &\qquad\quad \mathit{store}\ (n+1) \gg\!= \lambda() \rightarrow \\ &\qquad\quad \mathit{return}\ n \end{aligned}$$

2.3 Monadic Parsing Combinators

In practice, combinator libraries are usually based on monads providing a combination of features. For example, a parser for values of type a can be represented by the type

$$\textbf{newtype}\ \mathit{Parser}\ s\ a = P\ ([s] \rightarrow \mathit{Maybe}\ (a, [s]))$$

where s is the type used to represent symbols in the parser's input, and $[s]$ is Haskell's notation for the type *list-of-s*. Such a parser is invoked by applying its representation to a list of symbols to parse; its result indicates whether or not parsing was successful, and in the event of success contains both the value parsed and the remaining, unparsed input. For example, a parser which recognises a particular symbol can be defined by

$$\begin{aligned} &\mathit{symbol} :: s \rightarrow \mathit{Parser}\ s\ s \\ &\mathit{symbol}\ s = P\ (\lambda xs \rightarrow \ \textbf{case}\ xs\ \textbf{of} \\ &\qquad\qquad\qquad\qquad [\,] \rightarrow \mathit{Nothing} \\ &\qquad\qquad\qquad\qquad (x : xs') \rightarrow \textbf{if}\ x = s\ \textbf{then}\ \mathit{Just}\ (s, xs')\ \textbf{else}\ \mathit{Nothing}) \end{aligned}$$

This parser fails if the input is empty or begins with the wrong symbol, and succeeds with one symbol consumed from the input otherwise.

This representation of parsers supports a combination of failure handling and state passing, where the state is the unparsed input. It can be declared to be a monad just like the *Maybe* and *StateMonad* types above — see Wadler's articles for details. Further combinators can then be added to build up a complete library for parsing based on this monad.

2.4 Why Use Monads?

We have now seen that monads *can* be used as a basis for combinator libraries, but why *should* they be used? Why have monads become so ubiquitous in Haskell programs today?

One reason, of course, is that using monads simplifies code dramatically. It should be clear that writing a parser with explicit tests for failure and explicit passing of the input here and there, would be much more labour intensive than

writing one in terms of *symbol*, *return* and $\gg=$. However, this is an advantage of using *any* combinator library to encapsulate coding details, and does not argue for using monads in particular.

Another reason for using monads is that they offer a design guideline for combinator libraries: it is often a good start to begin by defining a suitable monad. For example, it is fairly clear that a library for parsing should include a combinator to invoke two parsers in sequence, but there are many possible ways in which such a combinator might handle the two parsers' results. In some early parsing libraries the two results were paired together, in others the sequencing combinator took an extra parameter, a function to combine the results. The monadic operator $\gg=$ is more general than either of these: both may be easily defined in terms of $\gg=$, but the converse is not true. By basing a parsing library on a monad, the designer gives the user more flexibility than these ad hoc alternatives. Indeed, we know from experience that the monadic interface gives the library user great power.

On the other hand, the monad interface also gives the implementor of a combinator library flexibility, because there are so many possible implementations. We have already seen three examples of monads; in fact, using monad transformers [KW92, LHJ95], we can systematically construct an infinite variety of monads. A systematic approach to monad design helps the implementor to find an appropriate type to base a combinator library on, but also helps to make the library 'future proof'. Namely, should a future extension of the library require a change in the representation of, say, parsers, then the implementor can rest assured that there are a myriad alternatives. To put it another way, the monad interface itself does not constrain the choice of monad type very much at all; it exposes very little of the internal workings of the library to the rest of the program. Consequently monads help the library maintainer to upgrade a combinator library without forcing changes in the code that uses it.

Finally, the fact that the monad operations *return* and $\gg=$ are overloaded in Haskell permits us to write *generic* monadic code, which can be used together with any library based on a monad. A growing collection of such functions are provided in the standard Haskell library. For example, we can generalise the *add* function above (for adding two possibly-failing integers) into a generic function which applies *any* binary operator to the results of two computations.

$$
\begin{array}{l}
\mathit{liftM2} :: \mathit{Monad}\ m \Rightarrow (a \to b \to c) \to m\ a \to m\ b \to m\ c \\
\mathit{liftM2}\ \mathit{op}\ x\ y = \quad x \gg= \lambda a \to \\
\qquad\qquad\qquad\qquad\ \ y \gg= \lambda b \to \\
\qquad\qquad\qquad\qquad\ \ \mathit{return}(x\ \text{`op`}\ y)
\end{array}
$$

(The $\mathit{Monad}\ m \Rightarrow$ in the type of *liftM2* is a *context*, and means that this function may be used for any monad type m). Now the `` `cat` `` operator on parsers that we saw in the introduction can be defined simply as

$$\mathit{cat} = \mathit{liftM2}\ (+\!\!+)$$

(where $+\!\!+$ is Haskell's concatenation operator for lists).

Generic code of this sort represents functionality that the designer of an individual combinator library no longer needs to provide: simply by basing the library on a monad, one gains access to a host of useful functions for free. This in turn may significantly reduce the work required to produce each new library.

Taken together, these arguments provide rather compelling reasons for using monads in combinator design; it is no wonder that they have become so ubiquitous.

2.5 Further Parsing Combinators

Let us pursue our example of combinators for parsing a little further. One of the things a parser can do is to fail; to enable us to express this we define a combinator which always fails. In fact, very many monads support a notion of failure, and so it is useful to overload the failure operator, just as we overloaded monadic *return* and $\gg\!=$. In Haskell this is done via a predefined class

$$\begin{array}{l}\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadZero}\ m\ \textbf{where}\\ \quad \mathit{zero} :: m\ a\end{array}$$

to be read as follows: a parameterised type m is a *MonadZero* if it is a *Monad*, and additionally supports the operation *zero*. The implementation of *zero* for parsers is then defined by

$$\begin{array}{l}\textbf{instance}\ \mathit{MonadZero}\ \mathit{Parser}\ \textbf{where}\\ \quad \mathit{zero} = P\ (\lambda s \rightarrow \mathit{Nothing})\end{array}$$

Moreover, many monads which support failure also support a *choice* combinator, which tries two alternative ways to perform a computation, using the second if the first one fails. Haskell defines a predefined class

$$\begin{array}{l}\textbf{class}\ \mathit{MonadZero}\ m \Rightarrow \mathit{MonadPlus}\ m\ \textbf{where}\\ \quad (+\!\!+) :: m\ a \rightarrow m\ a \rightarrow m\ a\end{array}$$

and the implementation for parsers is

$$\begin{array}{l}\textbf{instance}\ \mathit{MonadPlus}\ \mathit{Parser}\ \textbf{where}\\ \quad P\ a +\!\!+ P\ b = P\ (\lambda s \rightarrow\ \ \textbf{case}\ a\ s\ \textbf{of}\\ \qquad\qquad\qquad\qquad\qquad \mathit{Just}\ (x, s') \rightarrow \mathit{Just}\ (x, s')\\ \qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow b\ s)\end{array}$$

This is one of the fundamental building blocks of a parsing library: every interesting grammar defines some non-terminals via alternatives. But unfortunately, this definition contains a serious *space leak*. That is, it causes the retention of data by the garbage collector much longer than one would naively anticipate, with the result that parsers built with this operator use much more space than one would reasonably expect.

The problem is actually inherent to backtracking parsers. By inspection, the input to be parsed, s, cannot be garbage collected while the first parser a is running, because if a eventually fails, then s must be passed to b. In a lazy language such as Haskell, it is the very act of running parser a which forces the list of tokens s to be constructed, perhaps by reading from a file. Provided a fails quickly, without forcing the evaluation of many elements of s, then little space is used. But if a actually succeeds in parsing a large part of the input s, then a great deal of space may be used to hold these already-parsed tokens, just in case a eventually fails and b needs to be invoked. Ironically, in practice a and b usually recognise quite different syntactic constructs, so that if a succeeds in parsing many symbols then b will almost certainly fail as soon as it is invoked. Saving the input for b is costly only when it is unnecessary!

This problem has been known since combinator libraries for parsing were first proposed, and Wadler for example gives a partial solution in his 1985 paper [Wad85]. But the solutions known for monadic parser libraries are only partial, and depend on the programmer using an additional combinator similar to Prolog's 'cut' operator, to declare that a parser need never backtrack beyond a certain point. Although monadic parser libraries work quite well in practice, the fundamental problem remains unsolved, which is really rather unsatisfactory.

3 Swierstra and Duponcheel's Parsing Library

In 1996, Swierstra and Duponcheel found a different way to solve this problem. They restrict their attention to LL(1) parsers, in which choices between alternative parses can always be resolved by looking at the next token of the input. Their implementation of $a{+\!\!+}b$ can therefore choose between a and b immediately, and there is no need to save the input s in case the other alternative needs to be tried later. The space leak that other parsing libraries suffer from is completely cured.

To implement this idea, Swierstra and Duponcheel need to be able to tell, given a parser, which tokens it might accept as the first in the input (and also whether or not it can accept the empty sequence of tokens). This means that parsers can no longer be represented as functions, as they were in the previous section. Instead, they are represented as a combination of static information, which can be computed before parsing begins, and a parsing function, which can be optimised on the basis of the static information. Paraphrasing Swierstra and Duponcheel, we might define

```
data StaticParser s =  SP Bool [s]
newtype DynamicParser s a = DP ([s] → (a, [s]))
data Parser s a =  P (StaticParser s) (DynamicParser s a)
```

The first component of a parser tells us whether it matches the empty string, and which tokens it can accept first, while the second component is a function

which does the actual parsing. For example, the combinator which accepts a particular symbol can be defined as

$$
\begin{aligned}
&\mathit{symbol} :: s \to \mathit{Parser}\ s\ s \\
&\mathit{symbol}\ s = P\ (\mathit{SP}\ \mathit{False}\ [s])\ (\mathit{DP}\ (\lambda(x : xs) \to (s, xs)))
\end{aligned}
$$

The dynamic parsing function need not test for an empty input, or check that the first symbol is s, because it will be invoked only when the preconditions expressed by the static part are satisfied.

Now we can make use of the static information to define the choice combinator efficiently:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{MonadPlus}\ \mathit{Parser}\ \textbf{where} \\
&\quad P\ (\mathit{SP}\ \mathit{empty}_1\ \mathit{starters}_1)\ (\mathit{DP}\ p_1) +\!\!+ P\ (\mathit{SP}\ \mathit{empty}_2\ \mathit{starters}_2)\ (\mathit{DP}\ p_2) = \\
&\qquad P\ \ (\mathit{SP}\ (\mathit{empty}_1 \vee \mathit{empty}_2)\ (\mathit{starters}_1 +\!\!+ \mathit{starters}_2)) \\
&\qquad\quad (\mathit{DP}\ (\lambda xs \to \\
&\qquad\qquad \textbf{case}\ xs\ \textbf{of} \\
&\qquad\qquad\quad [] \qquad\ \to\ \ \textbf{if}\ \mathit{empty}_1\ \textbf{then}\ p_1\ []\ \textbf{else}\ p_2\ [] \\
&\qquad\qquad\quad x : xs' \ \to\ \ \textbf{if}\ x \in \mathit{starters}_1\ \textbf{then}\ p_1\ (x : xs')\ \textbf{else} \\
&\qquad\qquad\qquad\qquad\ \ \textbf{if}\ x \in \mathit{starters}_2\ \textbf{then}\ p_2\ (x : xs')\ \textbf{else} \\
&\qquad\qquad\qquad\qquad\ \ \textbf{if}\ \mathit{empty}_1\ \textbf{then}\ p_1\ (x : xs')\ \textbf{else}\ p_2\ (x : xs')))
\end{aligned}
$$

It is clear from this definition that the choice of whether to invoke p_1 or p_2 is made directly, and once made cannot be revised, so there is no need to retain a pointer to the input, and consequently no space leak[2].

Just as the $+\!\!+$ operator computes the starter symbols and potential emptiness of the parser it constructs, so must all of the other combinators. In most cases this is straightforward to do, but unfortunately in the case of $\gg\!=$ it turns out to be impossible! To see why, recall the type which $\gg\!=$ must have in this case:

$$(\gg\!=) :: \mathit{Parser}\ s\ a \to (a \to \mathit{Parser}\ s\ b) \to \mathit{Parser}\ s\ b$$

Now, the static properties of the result of $\gg\!=$ depend on the static properties of *both* the first and the second argument — for example, the combination can match the empty sequence only if both arguments can. Yet in the definition of $\gg\!=$, while we have access to the static properties of the first argument, we *cannot* obtain the static properties of the second one without applying it to a value of type a. Such values will be constructed only during parsing, but for Swierstra and Duponcheel's idea to be useful we must compute the static parts

[2]However, this definition is not completely realistic. It assumes that the user of the library really does write an LL(1) parser, so that $\mathit{starters}_1$ and $\mathit{starters}_2$ are disjoint. In a real implementation this would of course be checked. Moreover, the expensive tests of the form $x \in \mathit{starters}_1$ can be avoided by choosing a cleverer representation of parsers — see Swierstra and Duponcheel's article for details.

of parsers once and for all, before parsing begins. It is simply impossible to find a definition of $\gg\!=$ which does this.

Swierstra and Duponcheel's solution to this problem was to abandon the use of a monad: instead of $\gg\!=$ they defined a different sequencing operator with the type

$$(\lessdot\!\!\gtrdot) :: \mathit{Parser}\ s\ (a \to b) \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b$$

This operator is perfectly adequate for expressing parsers, and poses no problem as far as computing static properties in advance of parsing is concerned. Nevertheless, the need to abandon the monad signature is worrying, for the reasons we discussed above. Useful as it is, Swierstra and Duponcheel's parsing library stands alone; it cannot, for example, be used with generic monadic functions.

If this were an isolated case we might simply ignore it. But Swierstra and Duponcheel's idea is clearly much more widely applicable: to optimise a combinator library, redefine the combinators to collect static properties of the computations they construct, and then use those static properties to optimise the dynamic computations. If we think of a library as defining a domain specific 'language', whose constructions are represented as combinators, then Swierstra and Duponcheel's idea is to implement the language via a combination of a static analysis and an optimised dynamic semantics. We may clearly wish to do this very often indeed. But every time we do, the type of $\gg\!=$ will make it impossible to use a monadic interface!

It is this observation that motivated us to search for a generalisation of monads, a generic interface for combinator libraries that fits a much wider class of applications. We will introduce the generalisation we found in the next section.

3.1 On Category Theory

Before we do so, we make a short digression on the subject of category theory. The concept of a monad was developed by category theorists long before it eventually found an application in functional programming. Some might find it surprising that something so abstract as category theory should turn out to be useful for something so concrete as programming. After all, category theory is, in a sense, so abstract as to be rather unsatisfying: it is 'all definitions and no theorems', almost everything turns out to be a category if you look at it long enough, to say something is a category is actually to say very little about it. The same is true of most categorical concepts: they have very many possible instantiations, and so to say that something is, for example, a monad, is to say very little. This extreme generality is one reason why it is hard for the beginner to develop good intuitions about category theory, but it is hardly surprising: category theory was, after all, developed to be a 'theory of everything', a framework into which very many different mathematical structures would fit. But why should a theory so abstract be of any use for programming?

The answer is simple: as computer scientists, *we value abstraction*! When we design the interface to a software component, we *want* it to reveal as little as possible about the implementation. We want to be able to replace the implementation with many alternatives, many other 'instances' of the same 'concept'. When we design a generic interface to many program libraries, it is even more important that the interface we choose have a wide variety of implementations. It is the very generality of the monad concept which we value so highly, it is *because* category theory is so abstract that its concepts are so useful for programming.

It is hardly surprising, then, that the generalisation of monads that we present below also has a close connection to category theory. But we stress that our purpose is very practical: it is not to 'implement category theory', it is to find a more general way to structure combinator libraries. It is simply our good fortune that mathematicians have already done much of the work for us!

4 Arrows

Returning to our problem, recall that Swierstra and Duponcheel were unable to implement

$$(\gg\!=) :: \mathit{Parser}\ s\ a \to (a \to \mathit{Parser}\ s\ b) \to \mathit{Parser}\ s\ b$$

because its second argument is a function, and the only thing one can do with a function is apply it. Lacking a suitable value of type a to apply it to, they could not extract any static information from it, and therefore could not construct the static part of $\gg\!=$'s result.

Our solution is simply to *change the representation* of this argument. Rather than a function of type $a \to \mathit{Parser}\ s\ b$ we will use an abstract type, which we will call an *arrow* from a to b. We solve Swierstra and Duponcheel's problem by choosing a representation for arrows which makes static properties immediately accessible.

In fact there is no need to work with two abstract types, a monad type and an arrow type. Instead we will work purely with arrows. In general, an arrow type will be a parameterised type with *two* parameters, supporting operations analogous to *return* and $\gg\!=$. Just as we think of a monadic type $m\ a$ as representing a 'computation delivering an a', so we think of an arrow type $a\ b\ c$ (that is, the application of the parameterised type a to the two parameters b and c) as representing a 'computation with input of type b delivering a c'; arrows make the dependence on input explicit.

Just as Haskell defines a *Monad* class, so we shall define an *Arrow* class with analogous operators. But we must make dependence on an input explicit. Thus while the *return* operator, with type $a \to m\ a$, merely converts a value into a computation, its analogue for arrows, with type $(b \to c) \to a\ b\ c$, converts a

function from input to output into a computation. The analogue of $\gg=$ is just composition of arrows. We define

$$\begin{aligned}&\textbf{class } \mathit{Arrow}\ a \textbf{ where}\\ &\quad \mathit{arr} :: (b \to c) \to a\ b\ c\\ &\quad (\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d\end{aligned}$$

For any monad m, functions of type $a \to m\ b$ are potential arrows. If we give this type a name,

$$\textbf{newtype } \mathit{Kleisli}\ m\ a\ b = K\ (a \to m\ b)$$

then we can implement the arrow operations as follows:

$$\begin{aligned}&\textbf{instance } \mathit{Monad}\ m \Rightarrow \mathit{Arrow}\ (\mathit{Kleisli}\ m) \textbf{ where}\\ &\quad \mathit{arr}\ f = K\ (\lambda b \to \mathit{return}\ (f\ b))\\ &\quad K\ f \ggg K\ g = K\ (\lambda b \to f\ b \gg= g)\end{aligned}$$

This shows that arrows do indeed generalise monads; for every monad type, there is a corresponding arrow type. (Categorically speaking, we just constructed the Kleisli category of the monad m). Of course, we will see later that there are also many other, non-monadic implementations of the arrow signature.

4.1 Arrows and Pairs

However, even though in the case of monads the operators *return* and $\gg=$ are all we need to begin writing useful code, for arrows the analogous operators *arr* and $\ggg$ are not sufficient. Even the simple monadic addition function that we saw earlier

$$\begin{aligned}&\mathit{add} :: \mathit{Monad}\ m \Rightarrow m\ \mathit{Int} \to m\ \mathit{Int} \to m\ \mathit{Int}\\ &\mathit{add}\ x\ y = x \gg= \lambda u \to y \gg= \lambda v \to \mathit{return}\ (u + v)\end{aligned}$$

cannot yet be expressed in an arrow form. Making dependence on an input explicit, we see that an analogous definition should take the form

$$\begin{aligned}&\mathit{add} :: \mathit{Arrow}\ a \Rightarrow (a\ b\ \mathit{Int}) \to (a\ b\ \mathit{Int}) \to (a\ b\ \mathit{Int})\\ &\mathit{add}\ f\ g = \ldots\end{aligned}$$

where we must combine f and g in sequence. The only sequencing operator available is $\ggg$, but f and g do not have the right types to be composed. Indeed, the *add* function needs to *save the input* of type b across the computation of f, so as to be able to supply the same input to g. Likewise the result of f must be saved across the computation of g, so that the two results can eventually be added together and returned. The arrow combinators so far introduced give us no way to save a value across another computation, and so we have no alternative but to introduce another combinator.

We extend the definition of the *Arrow* class as follows:

```
class Arrow a where
    arr :: (a → b) → a b c
    (>>>) :: a b c → a c d → a b d
    first :: a b c → a (b,d) (c,d)
```

The new operator *first* converts an arrow from b to c into an arrow on pairs, that applies its argument to the first component and leaves the second component untouched, thus saving its value across a computation. Once again, we can implement *first* for any Kleisli arrow:

```
instance Monad m ⇒ Arrow (Kleisli m) where
  ...
  first (K f) = K (λ(b,d) → f b >>= λc → return(c,d))
```

Given *first*, we can define a combinator that applies its argument to the second component instead,

```
second :: Arrow a ⇒ a b c → a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
  where swap (x,y) = (y,x)
```

a combinator which processes both components of a pair,

```
(***) :: Arrow a ⇒ a b c → a d e → a (b,d) (c,e)
f *** g = first f >>> second g
```

and a combinator which builds a pair from the results of two arrows,

```
(&&&) :: Arrow a ⇒ a b c → a b d → a b (c,d)
f &&& g = arr (λb → (b,b)) >>> (f *** g)
```

With these definitions the *add* function is easily completed:

```
add :: Arrow a ⇒ (a b Int) → (a b Int) → (a b Int)
add f g = (f &&& g) >>> arr (λ(u,v) → u + v)
```

Just as we abstracted the idea of applying a binary operator to the results of two monadic computations, by going on to define *liftM2*, so we can generalise the arrow version likewise:

```
liftA2 :: Arrow a ⇒ (b → c → d) → a e b → a e c → a e d
liftA2 op f g = (f &&& g) >>> arr (λ(b,c) → b `op` c)
```

By this point the reader with a categorical background may have formed the impression that arrows with the extended interface implement a category with products. After all, we can construct arrows into a pair type using &&&, and we can construct projection arrows as *arr fst* and *arr snd*. Beware! In fact, there is no reason to expect Haskell's pair type to be a categorical product in the

category of arrows, or indeed to expect any categorical product to exist. This would require properties such as

$$(f \mathbin{\&\&\&} g) \ggg arr\ fst = f$$

to hold, and in general, since our arrows usually represent computations with some sort of effect, laws of this sort are simply false. In this case, the side-effects of g are lost on the right hand side.

The reader may also wonder why we chose to take *first* as primitive, rather than (say) &&& which resembles a well-known categorical operator. There are two main reasons for our choice.

- Firstly, since in general our arrows represent computations with effects, evaluation order makes a difference. The definition of $f \mathbin{\&\&\&} g$ above is explicit about this: the effects of f are composed with the effects of g *in that order*, that is evaluation is left-to-right. The definitions of &&& and *** above can be used as algebraic laws by the programmer, laws which capture evaluation order. In contrast, had we taken &&& as primitive, then the designer of each arrow-based library would have had to choose either left-to-right or right-to-left evaluation, with the result that evaluation order would probably differ from case to case. This would make the behaviour of arrow-based libraries less predictable, and reduce the number of useful laws that arrow combinators satisfy.

- Secondly, *first* is a simpler operator than &&&, and in general its implementation is around half the size of that of the latter. In practice the implementations of arrow combinators can be quite complex, and by making the choice we did we reduce the work required to build a new arrow-based library appreciably.

4.2 Arrows and Interpreters

How awkward is it to program with arrow combinators instead of monadic ones? And how expressive are the combinators in each case — are there some kinds of program which can be expressed using *return* and $\gg\!=$, but cannot be written at all in terms of *arr*, $\ggg$ and *first*? We can begin to answer both questions by looking at (fragments of) an interpreter based on arrows *vs.* one based on monads. If we can write an interpreter in which program fragments in a certain language are interpreted as arrows, then we know that any kind of program expressible in the interpreted language can also be expressed in terms of the arrow combinators.

To begin with, we shall consider a tiny language with only variables and addition. We represent expressions by the datatype

$$\textbf{data}\ Exp = \quad Var\ String \mid Add\ Exp\ Exp$$

The value of such an expression is always an integer, but in anticipation of making extensions we introduce a separate type of values anyway:

$$\textbf{data}\ \mathit{Val} = \ \mathit{Num}\ \mathit{Int}$$

We will also require a type for environments:

$$\textbf{type}\ \mathit{Env} = [(\mathit{String}, \mathit{Val})]$$

Now, a monadic interpreter maps expressions to computations, represented using a monad M. To do so, we introduce an evaluation function

$$\mathit{eval} :: \mathit{Exp} \to \mathit{Env} \to M\ \ \mathit{Val}$$

which we can define by

$$\begin{array}{l}\mathit{eval}\ (\mathit{Var}\ s)\ \ \mathit{env} = \mathit{return}\ (\mathit{lookup}\ s\ \ \mathit{env}) \\ \mathit{eval}\ (\mathit{Add}\ e_1\ \ e_2)\ \ \mathit{env} = \mathit{liftM2}\ \ \mathit{add}\ (\mathit{eval}\ e_1\ \ \mathit{env})\ (\mathit{eval}\ e_2\ \ \mathit{env}) \\ \quad \textbf{where}\ \ \mathit{add}\ (\mathit{Num}\ u)\ (\mathit{Num}\ v) = \mathit{Num}\ (u + v)\end{array}$$

An arrow interpreter, on the other hand, maps expressions to computations represented as arrows. But what should the input of an arrow denoting an expression be? By analogy with the monadic case, it is natural to take the input of an expression to be the environment. In an arrow interpreter based on arrow type A, we therefore give *eval* the type

$$\mathit{eval} :: \mathit{Exp} \to A\ \ \mathit{Env}\ \ \mathit{Val}$$

We can define *eval* as follows:

$$\begin{array}{l}\mathit{eval}\ (\mathit{Var}\ s) = \mathit{arr}\ (\mathit{lookup}\ s) \\ \mathit{eval}\ (\mathit{Add}\ e_1\ \ e_2) = \mathit{liftA2}\ \ \mathit{add}\ (\mathit{eval}\ e_1)\ (\mathit{eval}\ e_2) \\ \quad \textbf{where}\ \ \mathit{add}\ (\mathit{Num}\ u)\ (\mathit{Num}\ v) = \mathit{Num}\ (u + v)\end{array}$$

As we can see, at least in this small example, the arrow code is by no means more awkward than the monadic code. Indeed, often the user of a monadic combinator library works more with derived operators such as *liftM2* than with the operators in the monad signature themselves. Where analogous operators can be defined on arrows, arrow programs are essentially the same as monadic ones.

4.2.1 Interpreting Conditionals

Let us pursue the interpreter example a little further, and add a conditional expression to the interpreted language. We extend the expression and value types as follows:

$$\begin{array}{l}\textbf{data}\ \mathit{Exp} = \ \ldots \mid \mathit{If}\ \mathit{Exp}\ \mathit{Exp}\ \mathit{Exp} \\ \textbf{data}\ \mathit{Val} = \ \ldots \mid \mathit{Bl}\ \mathit{Bool}\end{array}$$

The monadic interpreter is easy to extend; we add a new case

$$\begin{aligned} \mathit{eval}\ (\mathit{If}\ e_1\ e_2\ e_3)\ \mathit{env} = \ & \mathit{eval}\ e_1\ \mathit{env} \gg\!= \lambda(\mathit{Bl}\ b) \rightarrow \\ & \mathbf{if}\ b\ \mathbf{then}\ \mathit{eval}\ e_2\ \mathit{env}\ \mathbf{else}\ \mathit{eval}\ e_3\ \mathit{env} \end{aligned}$$

But the arrow interpreter is more difficult. Certainly we could define

$$\begin{aligned} \mathit{eval}\ (\mathit{If}\ e_1\ e_2\ e_3) = \ & (\mathit{eval}\ e_1 \mathbin{\&\!\!\&\!\!\&} \mathit{eval}\ e_2 \mathbin{\&\!\!\&\!\!\&} \mathit{eval}\ e_3) \ggg \\ & \mathit{arr}\ (\lambda(\mathit{Bl}\ b, (v_1, v_2)) \rightarrow \mathbf{if}\ b\ \mathbf{then}\ v_1\ \mathbf{else}\ v_2) \end{aligned}$$

but this doesn't properly capture the meaning of a conditional expression: *both* branches are evaluated, and we just choose between the results. Of course the intention is to evaluate just *one* branch, depending on the value of the boolean.

And this is the crux of the problem: the arrow combinators provide no way to choose between two arrows on the basis of an input. To do so, we are obliged to add a new combinator. But this time, we choose to define a new class *ArrowChoice* rather than enlarge the existing *Arrow* class further. By doing so we retain the freedom to define arrow types which *do not* support a dynamic choice combinator; they will simply fail to be instances of our new class.

The new combinator we want will choose between two arrows on the basis of the input, and it makes sense therefore for the input to be of Haskell's sum type

$$\mathbf{data}\ \mathit{Either}\ a\ b = \ \mathit{Left}\ a \mid \mathit{Right}\ b$$

We will define $(f \mid\mid\mid g)$ to pass *Left* inputs to f and *Right* inputs to g, so the type of $\mid\mid\mid$ will be

$$(\mid\mid\mid) :: \mathit{ArrowChoice}\ a \Rightarrow a\ b\ d \rightarrow a\ c\ d \rightarrow a\ (\mathit{Either}\ b\ c)\ d$$

However, just as we chose to define *first* as an arrow primitive rather than &&&, so we choose a simpler operator than $\mid\mid\mid$ as the primitive method in the *ArrowChoice* class. We define

$$\begin{aligned} & \mathbf{class}\ \mathit{Arrow}\ a \Rightarrow \mathit{ArrowChoice}\ a\ \mathbf{where} \\ & \quad \mathit{left} :: a\ b\ c \rightarrow a\ (\mathit{Either}\ b\ d)\ (\mathit{Either}\ c\ d) \end{aligned}$$

where *left* f invokes f only on *Left* inputs, and leaves *Right* inputs unchanged. As usual, we check that we can implement *left* for Kleisli arrows:

$$\begin{aligned} & \mathbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{ArrowChoice}\ (\mathit{Kleisli}\ m)\ \mathbf{where} \\ & \quad \mathit{left}\ (K\ f) = K\ (\lambda x \rightarrow\ \ \mathbf{case}\ x\ \mathbf{of} \\ & \qquad\qquad\qquad\qquad \mathit{Left}\ b \rightarrow f\ b \gg\!= \lambda c \rightarrow \mathit{return}\ (\mathit{Left}\ c) \\ & \qquad\qquad\qquad\qquad \mathit{Right}\ d \rightarrow \mathit{return}\ (\mathit{Right}\ d)) \end{aligned}$$

Once we have introduced *left*, we can define

$$
\begin{array}{l}
right\ f = arr\ mirror \ggg left\ f \ggg arr\ mirror \\
\quad \textbf{where} \quad mirror\ (Left\ x) = Right\ x \\
\qquad\qquad\ \ mirror\ (Right\ x) = Left\ x
\end{array}
$$

$$
f <+> g = left\ f \ggg right\ g
$$

$$
\begin{array}{l}
f \mid\mid\mid g = (f <+> g) \ggg arr\ untag \\
\textbf{where} \quad untag\ (Left\ x) = x \\
\qquad\qquad\ \ untag\ (Right\ y) = y
\end{array}
$$

Now returning to our interpreter, we can at last define the interpretation of conditionals:

$$
\begin{array}{ll}
eval\ (If\ e_1\ e_2\ e_3) = & (eval\ e_1 \mathbin{\&\&\&} arr\ id) \ggg \\
& arr(\lambda(Bl\ b, env) \to \textbf{if}\ b\ \textbf{then}\ Left\ env\ \textbf{else}\ Right\ env) \ggg \\
& (eval\ e_2 \mid\mid\mid eval\ e_3)
\end{array}
$$

This is a little more awkward than the monadic code, but would be much simplified by introducing a combinator especially for testing predicates:

$$
\begin{array}{l}
test :: Arrow\ a \Rightarrow a\ b\ Bool \to a\ b\ (Either\ b\ b) \\
test\ f = (f \mathbin{\&\&\&} arr\ id) \ggg arr\ (\lambda(b, x) \to \textbf{if}\ b\ \textbf{then}\ Left\ x\ \textbf{else}\ Right\ x)
\end{array}
$$

Such a combinator is sufficiently useful that it is reasonable to include it in the arrow library, whereupon this case of our interpreter becomes no more complicated than the monadic version:

$$
eval\ (If\ e_1\ e_2\ e_3) = test\ (eval\ e_1 \ggg arr(\lambda(Bl\ b) \to b))\ (eval\ e_2 \mid\mid\mid eval\ e_3)
$$

4.2.2 Interpreting λ-Calculus

Using the combinators we have now introduced, we could go on to write an arrow interpreter for a complete first-order functional language. But can we interpret higher-order functions? Let us consider adding λ-expressions and (call-by-value) application to the interpreted language. We extend the type of expressions as follows:

$$
\textbf{data}\ Exp = \ \ldots \mid Lam\ String\ Exp \mid App\ Exp\ Exp
$$

Before we can extend the type *Val*, we must decide how to represent function values. Since calling a function may have an effect, we cannot interpret functions as values of type $Val \to Val$. In the monadic interpreter, we can use functions whose result is a computation,

$$
\textbf{data}\ Val = \ \ldots \mid Fun\ (Val \to M\ \ Val)
$$

while in the arrow interpreter, we naturally represent functions by arrows:

$$\textbf{data}\ \mathit{Val} = \ \ldots \mid \mathit{Fun}\ (A\ \mathit{Val}\ \mathit{Val})$$

The monadic *eval* function is easily extended to handle the new cases:

$$\begin{aligned}&\mathit{eval}\ (\mathit{Lam}\ x\ e)\ \mathit{env} = \mathit{return}\ (\mathit{Fun}\ (\lambda v \to \mathit{eval}\ e\ ((x, v) : \mathit{env})))\\ &\mathit{eval}\ (\mathit{App}\ e_1\ e_2)\ \mathit{env} = \mathit{eval}\ e_1\ \mathit{env} \gg\!= \lambda f \to \mathit{eval}\ e_2\ \mathit{env} \gg\!= \lambda v \to f\ v\end{aligned}$$

But the arrow version proves more difficult. Interpreting λ-expressions is unproblematic,

$$\mathit{eval}\ (\mathit{Lam}\ x\ e) = \mathit{arr}\ (\lambda \mathit{env} \to \mathit{Fun}\ (\mathit{arr}\ (\lambda v \to (x, v) : \mathit{env}) \ggg \mathit{eval}\ e))$$

but application is much harder. If we try to define

$$\mathit{eval}\ (\mathit{App}\ e_1\ e_2) = ((\mathit{eval}\ e_1 \ggg \mathit{arr}\ (\lambda(\mathit{Fun}\ f) \to f))\ \&\&\&\ \mathit{eval}\ e_2) \ggg \mathit{app}$$

for some suitable definition of *app*, then we find that *app* must invoke an arrow *which it receives as an input*, and there is no way to do so using the combinators so far introduced. There is nothing for it but to introduce another new class:

$$\begin{aligned}&\textbf{class}\ \mathit{Arrow}\ a \Rightarrow \mathit{ArrowApply}\ a\ \textbf{where}\\ &\quad \mathit{app} :: a\ (a\ b\ c, b)\ c\end{aligned}$$

whereupon the definition of *eval* above works. So, given an implementation of *app*, we can write an interpreter for the λ-calculus, and so we can also express other arrow programs in a higher-order style. Once more, it is easy to implement *app* for Kleisli arrows:

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{ArrowApply}\ (\mathit{Kleisli}\ m)\ \textbf{where}\\ &\quad \mathit{app} = K\ (\lambda(K\ f, x) \to f\ x)\end{aligned}$$

We have now seen that, given a monad m, we can define a corresponding arrow type *Kleisli m* which moreover supports all the other combinators we have introduced so far. Conversely, it turns out that, given an arrow type a which also supports *app*, we can define a corresponding monad type *ArrowMonad a*. The definition is simply

$$\textbf{newtype}\ \mathit{ArrowApply}\ a \Rightarrow \mathit{ArrowMonad}\ a\ b = M\ (a\ \mathit{Void}\ b)$$

where *Void* is Haskell's one-point type, whose only element is undefined. That is, a 'monadic' computation based on a is simply an arrow which ignores its input. We can now define the monad operations on *ArrowMonad a*:

$$\begin{aligned}&\textbf{instance}\ \mathit{ArrowApply}\ a \Rightarrow \mathit{Monad}\ (\mathit{ArrowMonad}\ a)\ \textbf{where}\\ &\quad \mathit{return}\ x = M\ (\mathit{arr}\ (\lambda z \to x))\\ &\quad M\ m \gg\!= f = M\ (\ m \ggg\\ &\qquad\qquad \mathit{arr}\ (\lambda x \to \textbf{let}\ M\ h = f\ x\ \textbf{in}\ (h, \mathit{undefined})) \ggg\\ &\qquad\qquad \mathit{app})\end{aligned}$$

We need *app* in order to invoke the arrow that the second argument of $\gg\!=$ produces.

One conclusion we can draw from this is that arrow types which support *app* are just as expressive as monads. In principle one might eliminate the concept of a monad from Haskell altogether, and replace it with arrows supporting *app*. But another conclusion to draw is that arrows supporting *app* are really of little interest to us here. Our motivation, after all, is to find a generic interface for combinator libraries which *cannot* be based on a monad. But clearly, any library which supports an arrow type with *app* could equally well be given a monadic interface. In the rest of the paper, therefore, we will be most interested in arrow types which *cannot* be made instances of *ArrowApply*.

5 Swierstra and Duponcheel's Parsers as Arrows

Now that we have introduced a number of arrow classes, let us return to Swierstra and Duponcheel's parsers. Recall that we defined their parser type as

```
data StaticParser s =  SP Bool [s]
newtype DynamicParser s a = DP ([s] → (a, [s]))
data Parser s a =  P (StaticParser s) (DynamicParser s a)
```

We were unable to make *Parser* into a monad, but can we make it into an arrow type?

To do so, we will need to add an extra type parameter, since arrow types take two parameters, whereas monad types take only one. Our intention is that the *static properties* of a parser should not depend on parse-time inputs, so let us change only the type of the dynamic parsing function:

```
newtype DynamicParser s a b = DP ((a, [s]) → (b, [s]))
data Parser s a b =  P (StaticParser s) (DynamicParser s a b)
```

Implementing the arrow combinators for this type is now straightforward:

```
instance Arrow (Parser s) where

  arr f = P (SP True []) (DP (λ(b, s) → (f b, s)))

  P (SP empty1 starters1) (DP p1) ⋙ P (SP empty2 starters2) (DP p2) =
    P (SP (empty1 ∧ empty2)
          (starters1 `union` if empty1 then starters2 else []))
      (DP (p2 ∘ p1))

  first (P sp (DP p)) =
    P sp (λ((b, d), s) → let (c, s') = p (b, s) in ((c, d), s'))
```

It is easy to modify the definitions from section 3 of *symbol*, the failure operator *zero*, and the choice combinator $+\!\!+$, to handle the arrows' input appropriately. Of course, since *zero* and $+\!\!+$ are overloaded names for monad operators, then we cannot use the same names for the corresponding operators on arrows. We therefore introduce two further arrow classes,

$$\begin{aligned}&\textbf{class } \mathit{Arrow}\ a \Rightarrow \mathit{ArrowZero}\ a \textbf{ where}\\ &\quad \mathit{zeroArrow} :: a\ b\ c\end{aligned}$$

$$\begin{aligned}&\textbf{class } \mathit{ArrowZero}\ a \Rightarrow \mathit{ArrowPlus}\ a \textbf{ where}\\ &\quad (+\!\!+) :: a\ b\ c \to a\ b\ c \to a\ b\ c\end{aligned}$$

and declare *Parser s* to be an instance of these classes instead. Having done so, we can go on to define all the operators in the interface that Swierstra and Duponcheel use, in terms of the arrow operations already introduced. For example, their sequencing operator is definable by

$$\begin{aligned}&(\lessdot\!\!\circledast\!\!\gtrdot) :: \mathit{Parser}\ s\ a\ (b \to c) \to \mathit{Parser}\ s\ a\ b \to \mathit{Parser}\ s\ a\ c\\ &(\lessdot\!\!\circledast\!\!\gtrdot) = \mathit{liftA2}\ (\lambda f\, x \to f\, x)\end{aligned}$$

So the user of an arrow-based parsing library can use it in exactly the same way as Swierstra and Duponcheel's original library, but in addition can combine parsers with generic arrow code.

What, then, of the other arrow classes, *ArrowChoice* and *ArrowApply*? A moment's thought shows that parsers *cannot* support these signatures. The choice operator $f \,|||\, g$ is supposed to make a dynamic choice between two arrows on the basis of the input, which implies that the possible starting symbols of $f \,|||\, g$ would depend on the arrow's input. But we have deliberately designed the *Parser* type so that the value of the input *cannot* affect the static component. It follows that $|||$ is unimplementable. A similar argument shows that *app* is also unimplementable (indeed, any arrow type which supports *app* can also support choice; to see this, give a definition of *left* in terms of *app*). Luckily this does not matter: it is rare that we *want* to write a parser which decides on the grammar to accept on the basis of previously parsed values.

What we see here is that the arrow interface lets the programmer make finer distinctions than the monad interface does; we can distinguish between types of computations that permit dynamic choices and calls of dynamic functions, and types of computations that do not. Swierstra and Duponcheel parsers do not. In contrast, once we declare a type to be a monad, we open the possibility of doing everything with it. And this is why the monadic interface is too restrictive.

6 Stream Processors: Processes as Arrows

We have already seen that any monad gives rise to a corresponding arrow type in a natural way, and that Swierstra and Duponcheel's parsers (or more generally,

combinators which collect static information about computations) can also be represented as arrows. In this section we will show that yet another 'non-monadic' notion of computation, namely that of a process, fits naturally into the arrow framework.

We concern ourselves for the time being with processes that have one input channel and one output channel. Such processes can be modelled in a purely functional language by *stream processors*. A stream processor maps a stream of input messages into a stream of output messages, but is represented by an abstract data type. Let $SP\ a\ b$ be the type of stream processors with inputs of type a and outputs of type b. Stream processors are then constructed using the operators

$$put :: b \to SP\ a\ b \to SP\ a\ b$$

which constructs a stream processor which outputs the b and then behaves like the second argument, and

$$get :: (a \to SP\ a\ b) \to SP\ a\ b$$

which constructs a stream processor which waits for an input, passes it to its function argument, and then behaves like the result. For simplicity we shall only consider non-terminating (recursively defined) stream processors; otherwise we would add another operator to construct a stream processor which halts.

Stream processors can be represented in several different ways, but quite a good choice is as a datatype with *put* and *get* as constructors:

$$\begin{aligned}&\textbf{data}\ SP\ a\ b = \ \ Put\ b\ (SP\ a\ b) \mid Get\ (a \to SP\ a\ b)\\ &put = Put\\ &get = Get\end{aligned}$$

Now we can write single processes using *put* and *get*, but to put processes together we need further combinators.

The arrow combinators turn out to represent very natural operations on processes! For readability we present them separately rather than as one large instance definition. The *arr* operator builds a stateless process that just applies a given function to its inputs to produce its outputs.

$$arr\ f = Get\ (\lambda x \to Put\ (f\ x)\ (arr\ f))$$

The $\ggg$ operator connects two processes in series:

$$\begin{aligned}&sp_1 \ggg Put\ c\ sp_2 = Put\ c\ (sp_1 \ggg sp_2)\\ &Put\ b\ sp_1 \ggg Get\ f = sp_1 \ggg f\ b\\ &Get\ f_1 \ggg Get\ f_2 = Get\ (\lambda a \to f_1\ a \ggg Get\ f_2)\end{aligned}$$

Notice that we define process composition *lazily*: the composition blocks waiting for an input only if *both* its constituent processes do.

Finally the *first* operator builds a process that feeds the first components of its inputs through its argument process, while the second components bypass the argument process and are recombined with its outputs. But what if the argument process does not produce one output per input? Our solution is to buffer the unconsumed inputs until corresponding outputs are produced. The function *bypass* takes as an additional argument the queue of second components waiting to bypass f:

$$
\begin{aligned}
&\mathit{first}\ f = \mathit{bypass}\ []\ f \\
\\
&\mathit{bypass}\ ds\ (\mathit{Get}\ f) = \mathit{Get}\ (\lambda(b,d) \to \mathit{bypass}\ (ds +\!\!+ [d])\ (f\ b)) \\
&\mathit{bypass}\ (d : ds)\ (\mathit{Put}\ c\ sp) = \mathit{Put}\ (c,d)\ (\mathit{bypass}\ ds\ sp) \\
&\mathit{bypass}\ []\ (\mathit{Put}\ c\ sp) = \mathit{Get}\ (\lambda(b,d) \to \mathit{Put}\ (c,d)\ (\mathit{bypass}\ []\ sp))
\end{aligned}
$$

With this definition, $f \mathbin{\&\&\&} g$ combines f and g in parallel, synchronising their output streams to produce a stream of pairs (and also synchronising their joint output with the input stream).

We can now use generic arrow combinators to write down stream processors. For example, the following stream processor outputs Fibonacci numbers:

$$
\begin{aligned}
&\mathit{fibs} = \mathit{put}\ 0\ \mathit{fibs}' \\
&\mathit{fibs}' = \mathit{put}\ 1\ (\mathit{liftA2}\ (+)\ \mathit{fibs}\ \mathit{fibs}')
\end{aligned}
$$

Stream processors also support a natural notion of failure: a failing process simply never produces more output. We can therefore define a *zeroArrow* as

$$
\begin{aligned}
&\textbf{instance}\ \mathit{ArrowZero}\ \mathit{SP}\ \textbf{where} \\
&\quad \mathit{zeroArrow} = \mathit{Get}\ (\lambda x \to \mathit{zeroArrow})
\end{aligned}
$$

We define $p +\!\!\!+\!\!\!+ q$ to run p and q in parallel, merging their outputs.

$$
\begin{aligned}
&\textbf{instance}\ \mathit{ArrowPlus}\ \mathit{SP}\ \textbf{where} \\
&\quad \mathit{Put}\ b\ sp_1 +\!\!\!+\!\!\!+ sp_2 = \mathit{Put}\ b\ (sp_1 +\!\!\!+\!\!\!+ sp_2) \\
&\quad sp_1 +\!\!\!+\!\!\!+ \mathit{Put}\ b\ sp_2 = \mathit{Put}\ b\ (sp_1 +\!\!\!+\!\!\!+ sp_2) \\
&\quad \mathit{Get}\ f_1 +\!\!\!+\!\!\!+ \mathit{Get}\ f_2 = \mathit{Get}\ (\lambda a \to f_1\ a +\!\!\!+\!\!\!+ f_2\ a)
\end{aligned}
$$

We take care to define parallel composition lazily also, so that $p +\!\!\!+\!\!\!+ q$ blocks waiting for input only if *both* p and q do.

These definitions satisfy the laws

$$
\begin{aligned}
\mathit{zeroArrow} +\!\!\!+\!\!\!+ q &= q \\
p +\!\!\!+\!\!\!+ \mathit{zeroArrow} &= p \\
(p +\!\!\!+\!\!\!+ q) +\!\!\!+\!\!\!+ r &= p +\!\!\!+\!\!\!+ (q +\!\!\!+\!\!\!+ r)
\end{aligned}
$$

which is a strong indication that they are reasonable.

Stream processors can also support dynamic choice. The stream processor *left sp* simply passes messages tagged *Left* through *sp*, while others are passed on directly.

```
instance ArrowChoice SP where
    left (Put c sp) = Put (Left c) (left sp)
    left (Get f) = Get (λz → case z of
                                 Left a   → left (f a)
                                 Right b  → Put (Right b) (left (Get f)))
```

With this definition, then $f \,|||\, g$ can be regarded as yet another kind of parallel composition, which routes inputs tagged *Left* to f and inputs tagged *Right* to g.

In fact, although stream processors have only one input and one output channel, we can model processes with many of each by *multiplexing* several channels onto one. For example, we can regard a channel carrying messages of type *Either a b* as a representation for *two* channels, one carrying as and the other carrying bs. With this viewpoint, $f \,|||\, g$ combines f and g in parallel to yield a stream processor with two input channels (multiplexed onto one), and merges the output channels onto one. Should we wish to combine f and g without merging their outputs, we can instead use $f \mathbin{<+>} g$. We can copy an input channel to two output channels using *arr Left* $\mathbin{+\!\!+\!\!+}$ *arr Right*, and so we can define a parallel combination of f and g with two output channels, but which copies one input channel to both processes by

$$f \,|\&|\, g = (arr\ Left \mathbin{+\!\!+\!\!+} arr\ Right) \ggg (f \mathbin{<+>} g)$$

We can write a stream processor with two input channels and one output, that just copies the first input channel and discards the second, or vice versa, as

$$\begin{aligned} justLeft &= arr\ id \,|||\, zeroArrow \\ justRight &= zeroArrow \,|||\, arr\ id \end{aligned}$$

Not surprisingly, combining two processes and then discarding the output channel from one of them is equivalent to the other:

$$\begin{aligned} (f \,|\&|\, g) \ggg justLeft &= f \\ (f \,|\&|\, g) \ggg justRight &= g \end{aligned}$$

But these properties have a categorical interpretation: they tell us that the *Either* type is a *weak categorical product* in the category of stream processors! (Only weak, because there is more than one way to define $|\&|$ so that these equations hold; our definition favours g over f in case both produce outputs simultaneously). In a deep sense, then, the *Either* type behaves more like a product than the pair type does, when we work with stream processors. And indeed, a channel carrying a sum type corresponds much more closely to a pair of channels than does a channel carrying pairs.

The only arrow class we have not yet shown how to implement is *ArrowApply*. But it turns out that there is no sensible definition of

$$app :: SP\ (SP\ a\ b, a)\ b$$

Since *app* would receive a new stream processor to invoke with every input, there is no real sense in which the stream processors it is passed would receive a *stream* of inputs; we could supply them with only one input each. This would really be very unnatural. Since stream processors do not support a natural definition of *app*, they cannot either be fitted into the monadic framework. They thus give us our second example of a useful kind of computation which cannot be represented as a monad.

However, recalling that *Either* may play the rôle of a product type for stream processors, we might instead of *app* consider looking for a function of type

$$dyn :: SP\ (Either\ (SP\ a\ b)\ a)\ b$$

There is actually a very natural definition with this type: the 'dynamic stream processor' *dyn* receives stream processors on its first input channel, and then passes inputs from its second input channel through the stream processor received, until it receives another stream processor to replace the first. We implement it as

```
dyn = dynloop zeroArrow
  where dynloop (Put b sp) = Put b (dynloop sp)
        dynloop (Get f) = Get (λz →
         case z of
           Right a  →  dynloop (f a)
           Left sp  →  dynloop sp)
```

Stream processors are not just amusing toys: they are at the heart of the *fudgets* combinator library for programming graphical user interfaces [CH93]. A fudget from a to b is like a stream processor with two extra hidden communication channels, to and from the window manager. A fudget can therefore exchange high-level messages with other fudgets, but can also manage a part of the screen. Thus a fudget has both an appearance and a behaviour, which makes them useful for structuring complex user interfaces.

The fudget type $F\ a\ b$ is actually implemented as a stream processor in which the high and low level communication channels are multiplexed onto one, in just the way we described. Since fudgets are just stream processors, they can also be declared to be arrows, supporting the same operations. Interestingly, almost all the operations we discussed in this section do indeed appear in the fudgets library — even *dyn* — although of course, they appear with different names, and not as instances of a general framework.

7 Functors: New Arrows from Old

One of the attractive features of monads is that they can be designed systematically, using so-called *monad transformers* [LHJ95]. A monad transformer is a monad parameterised on another monad, such that computations over the parameter monad can be 'lifted' to computations over the new one.

For example, the state monad of section 2.2 can be generalised to a monad transformer:

$$\textbf{newtype}\ \mathit{StateMonadT}\ s\ m\ a = \mathit{SM}\ (s \to m\ (a, s))$$

In general the monad operators on the new type must be defined in terms of the monad operators on the parameter monad, as in this case:

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (\mathit{StateMonadT}\ s\ m)\ \textbf{where}\\ \quad \mathit{return}\ a = \mathit{SM}\ (\lambda s \to \mathit{return}\ (a, s))\\ \quad x \gg\!= f = \mathit{SM}\ (\lambda s \to\ \ \textbf{let}\ \mathit{SM}\ x' = x\ \textbf{in}\\ \qquad\qquad\qquad\qquad\quad x'\ s \gg\!= \lambda(a, s') \to\\ \qquad\qquad\qquad\qquad\quad \textbf{let}\ \mathit{SM}\ f' = f\ a\ \textbf{in}\\ \qquad\qquad\qquad\qquad\quad f'\ s')\end{array}$$

Lifting of computations is defined by passing the state through unchanged:

$$\begin{array}{l}\mathit{liftState} :: \mathit{Monad}\ m \Rightarrow m\ a \to \mathit{StateMonadT}\ s\ m\ a\\ \mathit{liftState}\ x = \mathit{SM}\ (\lambda s \to x \gg\!= \lambda a \to \mathit{return}\ (a, s))\end{array}$$

Finally, the new monad supports fetch and store operations, just like the original state monad:

$$\begin{array}{l}\mathit{fetch} :: \mathit{Monad}\ m \Rightarrow \mathit{StateMonad}\ s\ m\ s\\ \mathit{fetch} = \mathit{SM}\ (\lambda s \to \mathit{return}\ (s, s))\\ \\ \mathit{store} :: \mathit{Monad}\ m \Rightarrow s \to \mathit{StateMonad}\ s\ m\ ()\\ \mathit{store}\ x = \mathit{SM}\ (\lambda s \to \mathit{return}\ ((), x))\end{array}$$

The new monad thus supports all the computations of the parameter monad (by lifting), and in addition manages a state. By composing monad transformers together, one can build up a monad providing any desired combination of features. For example, if we want a monad which manages a state and handles failures, we can use the type $\mathit{StateMonadT}\ s\ \mathit{Maybe}$.

In this section we show that arrows have the same property: we can define 'arrow transformers' which map simpler arrow types to more complex ones. The most important monad transformers have arrow transformer counterparts, and we will describe those for handling failures, state, and continuations. An arrow transformer is, by analogy with a monad transformer, just an arrow type parameterised on another arrow type, such that arrows of the second type can be mapped into arrows of the first. But in fact, this corresponds closely to the

standard categorical notion of a *functor*, and so from now on we shall use the word functor instead of arrow transformer.

We note briefly that the concepts of monad transformers and functors can be formalised as classes, thus overloading the lifting operations, but that this requires a much more powerful class system than Haskell currently supports. We therefore refrain from doing so.

7.1 The Maybe Functor

Any arrow type can be lifted to an arrow type supporting failures by the functor

$$\textbf{newtype}\ \mathit{MaybeFunctor}\ a\ b\ c = \mathit{MF}\ (a\ b\ (\mathit{Maybe}\ c))$$

That is, we use arrows whose result can indicate failure. We can lift arrows to this type using

$$\begin{aligned}&\mathit{liftMaybe} :: \mathit{Arrow}\ a \Rightarrow a\ b\ c \to \mathit{MaybeFunctor}\ a\ b\ c\\ &\mathit{liftMaybe}\ f = \mathit{MF}\ (f \ggg \mathit{arr}\ \mathit{Just})\end{aligned}$$

The arrow operations need to handle failures, which means they need to make dynamic decisions. We therefore must require that the parameter arrow type supports choice:

$$\begin{aligned}&\textbf{instance}\ \mathit{ArrowChoice}\ a \Rightarrow \mathit{Arrow}\ (\mathit{MaybeFunctor}\ a)\ \textbf{where}\\ &\quad \mathit{arr}\ f = \mathit{liftMaybe}\ (\mathit{arr}\ f)\\ &\quad \mathit{MF}\ f \ggg \mathit{MF}\ g = \mathit{MF}\ (\ f \ggg\\ &\qquad\qquad \mathit{arr}\ (\lambda z \to\ \textbf{case}\ z\ \textbf{of}\\ &\qquad\qquad\qquad\qquad \mathit{Just}\ c \to \mathit{Left}\ c\\ &\qquad\qquad\qquad\qquad \mathit{Nothing} \to \mathit{Right}\ \mathit{Nothing}) \ggg\\ &\qquad\qquad (g\ |||\ \mathit{arr}\ \mathit{id}))\\ &\quad \mathit{first}\ (\mathit{MF}\ f) = \mathit{MF}\ (\ \mathit{first}\ f \ggg\\ &\qquad\qquad \mathit{arr}\ (\lambda(c', d) \to\ \textbf{case}\ c'\ \textbf{of}\\ &\qquad\qquad\qquad\qquad \mathit{Just}\ c \to \mathit{Just}\ (c, d)\\ &\qquad\qquad\qquad\qquad \mathit{Nothing} \to \mathit{Nothing}))\end{aligned}$$

Arrows formed by *MaybeFunctor* support failure and failure handling, of course:

```
instance ArrowChoice a ⇒ ArrowZero (MaybeFunctor a) where
    zeroArrow = MF (arr (λz → Nothing))

instance ArrowChoice a ⇒ ArrowPlus (MaybeFunctor a) where
    MF f ⫫ MF g = MF (  (f &&& arr id) ⋙
                        arr (λ(c', b) →  case c' of
                                           Just c    →  Left c'
                                           Nothing   →  Right b) ⋙
                        (arr id ||| g))
```

and they also, not surprisingly, support choice:

```
instance ArrowChoice a ⇒ ArrowChoice (MaybeFunctor a) where
    MF f ||| MF g = MF (f ||| g)
```

Finally, if the underlying arrows support application, then so do the arrows produced by *MaybeFunctor*:

```
instance (ArrowChoice a, ArrowApply a) ⇒ ArrowApply (MaybeFunctor a) where
    app = MF (arr (λ(MF f, b) → (f, b)) ⋙ app)
```

7.2 The State Functor

Any arrow type can be lifted to an arrow type supporting state passing by the functor

```
newtype StateFunctor s a b c = SF (a (b, s) (c, s))
```

We can lift arrows to this type using

```
liftState :: Arrow a ⇒ a b c → StateFunctor s a b c
liftState f = SF (first f)
```

The arrow operations just pass the state along as one would expect:

```
instance Arrow a ⇒ Arrow (StateFunctor s a) where
    arr f = liftState (arr f)
    SF f ⋙ SF g = SF (f ⋙ g)
    first (SF f) = SF (  arr (λ((b, d), s) → ((b, s), d)) ⋙
                         first f ⋙
                         arr (λ((c, s), d) → ((c, d), s)))
```

Of course, the arrows produced by the *StateFunctor* support fetch and store operations:

```
fetch :: Arrow a ⇒ StateFunctor s a b s
fetch = SF (arr (λ(b, s) → (s, s)))

store :: Arrow a ⇒ StateFunctor s a s ()
store = SF (arr (λ(x, s) → ((), x)))
```

Stateful arrows inherit the ability to support dynamic choice, failure, and failure handling from the parameter arrow:

```
instance ArrowChoice a ⇒ ArrowChoice (StateFunctor s a) where
    left (SF f) = SF (  arr (λ(z, s) →  case z of
                                          Left b → Left (b, s)
                                          Right c → Right (c, s)) ⋙
                        ((f ⋙ first (arr Left)) ||| first (arr Right)))

instance ArrowZero a ⇒ ArrowZero (StateFunctor s a) where
    zeroArrow = SF zeroArrow

instance ArrowPlus a ⇒ ArrowPlus (StateFunctor s a) where
    SF f ⩲ SF g = SF (f ⩲ g)
```

Finally, if the underlying arrow type supports application, then so do stateful arrows based on it:

```
instance ArrowApply a ⇒ ArrowApply (StateFunctor s a) where
    app = SF (arr (λ((SF f, b), s) → (f, (b, s))) ⋙ app)
```

The state functor we have defined is of course closely related to the state monad transformer, but the advantage of defining functors on arrows, rather than transformers on monads, is that we can apply them to arrow types that do not correspond to any monad. As an example, the reader is invited to work out the behaviour of arrows of type *StateFunctor s SP*, derived by adding state passing to stream processors.

7.3 The CPS Functor

A third well-known monad transformer adds continuation passing to any monad. In the monadic world, we can define

```
newtype CPS ans m a = CPS ((a → m ans) → m ans)
```

so that a computation is represented by a function from a *continuation* for its result (a monadic function into an answer type) to the computation of the answer. In the world of arrows, we can represent a continuation by an arrow, rather than a function, and a continuation-passing arrow from b to c as a function from the continuation of the result to the continuation of the argument:

```
newtype CPSFunctor ans a b c = CPS ((a c ans) → (a b ans))
```

Lifting an arrow to the CPS type is straightforward:

$$\begin{aligned}&\mathit{liftCPS} :: \mathit{Arrow}\ a \Rightarrow a\ b\ c \to \mathit{CPSFunctor}\ \mathit{ans}\ a\ b\ c\\&\mathit{liftCPS}\ f = \mathit{CPS}\ (\lambda k \to f \ggg k)\end{aligned}$$

But now, in order to define the basic arrow operations on CPS arrows, we find we already need to use application at the underlying arrow type!

$$\begin{aligned}&\textbf{instance}\ \mathit{ArrowApply}\ a \Rightarrow \mathit{Arrow}\ (\mathit{CPSFunctor}\ \mathit{ans}\ a)\ \textbf{where}\\&\quad \mathit{arr}\ f = \mathit{liftCPS}\ (\mathit{arr}\ f)\\&\quad \mathit{CPS}\ f \ggg \mathit{CPS}\ g = \mathit{CPS}\ (\lambda k \to f\ (g\ k))\\&\quad \mathit{first}\ (\mathit{CPS}\ f) =\\&\qquad \mathit{CPS}\ (\lambda k \to \mathit{arr}\ (\lambda(b,d) \to (f\ (\mathit{arr}\ (\lambda c \to (c,d)) \ggg k), b)) \ggg \mathit{app})\end{aligned}$$

To define $\mathit{first}\ (\mathit{CPS}\ f)$ we must invoke f with a continuation which recombines its result with the second component of the argument. This we can do, but only in the scope of an $\mathit{arr}\ (\lambda(b,d) \to \ldots)$ which binds a name to that second component. We can only construct the arrow representing f's continuation within another arrow, and so we can only contruct the continuation of f's argument within an arrow, which forces us to use *app* to invoke it. In a way, since continuation passing is the epitomy of higher-order programming, this is not really surprising.

CPS arrows inherit the ability to support failures and failure handling from the underlying arrow type, and can of course support dynamic choice and application. We will not give the definition here, however. What we will do is show how to define a *jump* operator, which invokes a continuation supplied as its input

$$\begin{aligned}&\mathit{jump} :: \mathit{ArrowApply}\ a \Rightarrow \mathit{CPSFunctor}\ \mathit{ans}\ a\ (a\ c\ \mathit{ans}, c)\ z\\&\mathit{jump} = \mathit{CPS}\ (\lambda k \to \mathit{app})\end{aligned}$$

and a combinator *callcc*, which passes the current continuation to its argument arrow:

$$\begin{aligned}&\mathit{callcc} :: \mathit{ArrowApply}\ a \Rightarrow\\&\qquad (a\ c\ \mathit{ans} \to \mathit{CPSFunctor}\ \mathit{ans}\ a\ b\ c) \to \mathit{CPSFunctor}\ \mathit{ans}\ a\ b\ c\\&\mathit{callcc}\ f = \mathit{CPS}\ (\lambda k \to \textbf{let}\ \mathit{CPS}\ g = f\ k\ \textbf{in}\ g\ k)\end{aligned}$$

As we have seen, continuation passing arrows always support application, and must be based on an underlying arrow type which also supports application. Thus both the argument and the resulting arrow types correspond to monads. Our CPS functor is therefore no more general than the CPS monad transformer, but nonetheless, what we have shown is that we can work entirely with arrows even if we want to use continuation passing style.

8 Arrow Laws

Up to this point we have ignored the matter of laws. In fact the presentation of monads in section 2 was a little oversimplified: an implementation of *return* and $\gg\!=$ constitutes a monad only if the so-called *monad laws* are satisfied:

$$\begin{aligned} \mathit{return}\ x \gg\!= f &= f\ x \\ m \gg\!= \mathit{return} &= m \\ (m \gg\!= f) \gg\!= g &= m \gg\!= (\lambda x \to f\ x \gg\!= g) \end{aligned}$$

These laws state in essense that sequential composition is associative, and *return* is its unit, although they are complicated slightly by the need to pass values from one computation to the next. The programmer relies implicitly on the monad laws every time he or she uses a monad based library without worrying about how to bracket sequential compositions.

We will place similar requirements on the implementations of the arrow combinators. But since there are many more arrow combinators than monadic ones, we will require a larger collection of laws. All of the laws that we state in this section are satisfied by Kleisli arrows.

We can simplify the statements of the laws a little by noting that the ordinary function type can be declared to be an arrow:

$$\begin{array}{l} \textbf{instance}\ \mathit{Arrow}\ (\to)\ \textbf{where} \\ \quad \mathit{arr}\ f = f \\ \quad f \ggg g = g \circ f \\ \quad \mathit{first}\ f = \lambda(b, c) \to (f\ b, c) \end{array}$$

$$\begin{array}{l} \textbf{instance}\ \mathit{ArrowChoice}\ (\to)\ \textbf{where} \\ \quad \mathit{left}\ f\ (\mathit{Left}\ b) = \mathit{Left}\ (f\ b) \\ \quad \mathit{left}\ f\ (\mathit{Right}\ d) = \mathit{Right}\ d \end{array}$$

$$\begin{array}{l} \textbf{instance}\ \mathit{ArrowApply}\ (\to)\ \textbf{where} \\ \quad \mathit{app} = \lambda(f, x) \to f\ x \end{array}$$

Of course, we will require composition to be associative, and moreover to be preserved by *arr*:

$$\begin{aligned} (f \ggg g) \ggg h &= f \ggg (g \ggg h) \\ \mathit{arr}\ (f \ggg g) &= \mathit{arr}\ f \ggg \mathit{arr}\ g \end{aligned}$$

We will require an extensionality principle for arrows, that arrows which 'behave the same' for all inputs really are equal. We can formulate this as a law as follows:

$$\left.\begin{array}{c} \mathit{arr}\ h \ggg f = \mathit{arr}\ h \ggg g \\ h \text{ is onto} \end{array}\right\} \Longrightarrow f = g$$

Dually

$$\left.\begin{array}{c} f \ggg arr\ h = g \ggg arr\ h \\ h \text{ is one-to-one} \end{array}\right\} \Longrightarrow f = g$$

It follows that

$$arr\ id \ggg f = f = f \ggg arr\ id$$

(by composing on each side with *arr id*, since *id* is both one-to-one and onto). Categorically speaking, we now know that arrows form a category, and that *arr* is a functor from the category of Haskell functions to the category of arrows.

These laws correspond in some sense to the monad laws, but now we must go on to state the laws that the other arrow combinators are required to satisfy. Let us call an arrow *pure* if it is equal to *arr f* for some *f*; a pure arrow 'has no side-effects'. We shall require that all combinators behave for pure arrows as they do for functions; that is:

$$\begin{array}{rcl} first\ (arr\ f) & = & arr\ (first\ f) \\ left\ (arr\ f) & = & arr\ (left\ f) \end{array}$$

Furthermore we require that our combinators preserve composition:

$$\begin{array}{rcl} first\ (f \ggg g) & = & first\ f \ggg first\ g \\ left\ (f \ggg g) & = & left\ f \ggg left\ g \end{array}$$

Similar properties for *second* and *right* follow as easy consequences.

Notice, though, that it does *not* follow that

$$(f \mathbin{***} g) \ggg (h \mathbin{***} k) = (f \ggg h) \mathbin{***} (g \ggg k)$$

since the order of *g* and *h* differs on the two sides. This is another reason to favour *first* and *left* as primitives over their more usual binary counterparts: the laws they must satisfy become much simpler to state.

We formalise the property that *first f* depends only on first components of pairs as follows:

$$first\ f \ggg arr\ fst \quad = \quad arr\ fst \ggg f$$

but it is *not* in general true that

$$first\ f \ggg arr\ snd = arr\ snd$$

since, on the right hand side, the side-effects of *f* are lost. Instead we formalise the intuition that the second component of a pair is unaffected by *first f* as a law that allows a function of that second component to be moved across the use of *first*. We have to require that the function be pure, to avoid potentially changing the order in which side-effects occur. Thus the law becomes

$$first\ f \ggg second\ (arr\ g) = second\ (arr\ g) \ggg first\ f$$

Once again, the dual statement, in which *first* and *second* are interchanged, follows as an easy corollary.

We note in passing that many categorical properties of products fail in the presence of side effects. For example, the reader might expect that

$$f \ggg (g \mathbin{\&\&\&} h) = (f \ggg h) \mathbin{\&\&\&} (f \ggg h)$$

but this is not true (unless f is pure) because the side-effects of f are duplicated on the right.

The laws for *first* serve as models for the laws for *left*; we require that

$$\begin{aligned} arr\ Left \ggg left\ f &= f \ggg arr\ Left \\ right\ (arr\ g) \ggg left\ f &= left\ f \ggg right\ (arr\ g) \end{aligned}$$

Note here also that we cannot change the order of *left* f and *right* g unless we know that one of f or g is pure, because we might change the order of side-effects.

For arrows supporting application, we require firstly that 'currying' and then applying the identity arrow is equivalent to the identity (on pairs):

$$first\ (arr\ (\lambda x \to arr\ (\lambda y \to (x, y)))) \ggg app = arr\ id$$

Secondly, we require a kind of parametricity property for *app*, which permits operations to be moved in or out of the applied arrow:

$$\begin{aligned} first\ (arr\ (g \ggg)) \ggg app &= second\ g \ggg app \\ first\ (arr\ (\ggg h)) \ggg app &= app \ggg h \end{aligned}$$

From these laws we can prove an analogue of η-conversion, that applying a constant arrow using *app* is equivalent to the arrow itself:

$$arr\ (\lambda x \to (f, x)) \ggg app = f$$

Moreover, currying and then applying any arrow is equivalent to the arrow:

$$first\ (arr\ (\lambda x \to arr\ (\lambda y \to (x, y)) \ggg f)) \ggg app = f$$

Finally, we can prove that the monad laws hold for the *ArrowMonad* defined in section 4.2.2.

For the remaining arrow classes, *ArrowZero* and *ArrowPlus*, we just require that $\mid\mid\mid$ is associative, and *zeroArrow* is its unit. Stronger conditions, such as for example

$$zeroArrow \ggg f = zeroArrow$$

would be overly restrictive: this property fails for stream processors, for example, since f may very well produce outputs independently of its input.

In general, there is something of a conflict between the desire on the one hand to state many laws, thus making it possible to prove strong properties generically, for every kind of arrow, and the wish on the other hand to leave open the possibility of very many different implementations of the arrow signature. We believe that the laws we have stated in this section are a rather minimal set, which every reasonable arrow type should satisfy.

9 Active Web Pages: CGI Programs as Arrows

So for in this paper we have shown how the arrow interface can generalise a variety of existing combinator libraries. In this section we shall discuss a library we are currently developing, which was inspired by the concept of arrows.

The application that this library addresses is that of constructing active web pages, that is, pages that may appear differently each time they are visited. Active web pages are represented by programs, which may run either in the client browser (applets) or on the web server. Quite different technologies are used in each case; we concern ourselves here with programs which run on a web server. Such programs can query a database held on the server, allow clients to upload new data, and so on. Even rather simple programs can be very useful: for example, those which enable students to book meetings with a teacher, or researchers to submit articles to conferences.

Active web pages of this sort are implemented by so-called *CGI programs* stored on the server. When a client accesses the URL of the program, then it is run on the server, and the output from the program (usually HTML) is sent back to the client browser. There are a couple of different mechanisms for sending data from the client to the CGI program; the one we will consider sends an encoding of the fields of an HTML form to the web server, along with the request to run the program. CGI stands for *Common Gateway Interface*, the protocol governing the form in which data is sent to and fro between the client and the server.

Unfortunately, this mechanism is awkward to use in practice. Normally, the implementor of a CGI program wishes to lead the remote client through a series of interactions, for example first asking a student to identify him or herself, then offering a choice of meeting times, then confirming that a time has been booked. But interactions with the client can only take place in between runs of CGI programs. To ask the client a question, a CGI program must output the question as an HTML form, and terminate. When the client answers the question by filling in and submitting the form, then in general a different CGI program is run to accept and process the answer. This leads to poor modularity, because the format of the form (field names, etc) must be known both to the program which creates it, and to the program which interprets its contents. But a more severe problem is that the *state* of the CGI program is lost across the interaction.

It is therefore necessary to save the state of the CGI program *explicitly* across each interaction. *This cannot be done on the server!* It is by no means certain that the client ever will submit a reply, so that if the state were saved on the server then it might remain there for ever, waiting for a reply that never came. On the other hand, the client might submit a reply, then use the 'Back' button in the browser, and reply to the same question again! If second and subsequent replies are to be handled properly, then the state cannot be discarded even once a reply has been received.

The solution is to store the state of the CGI program *on the client*, along with the question. When the client submits an answer, then the state is returned along with it, permitting the CGI program to pick up from the same point that it left off. One can think of this state as a kind of continuation: when a CGI program wishes to ask the client something, it captures its current continuation and sends it along with the question to the client, and when the client replies then the continuation is returned to the server, and can be invoked to handle the reply. HTML provides a mechanism for handling such data: an HTML form can contain 'hidden fields' whose contents are returned unchanged to the server when the form is filled in and submitted. Unfortunately, though, HTML fields cannot contain function values, and so we must find a different way to represent continuations if we are to use this idea.

The combinator library I am developing takes care of suspension of computations, saving of state, and restart from the same point. It lets the CGI programmer view interaction with the client as a procedure call; there is an arrow

$$ask :: \mathit{CGI\ String\ String}$$

which maps a question to the client's answer. Thus programs which conduct a series of interactions can be implemented very simply. For example, consider a program which asks the client

> What is your question?

expecting to recieve a reply such as

> How old are you?

and then asks the client the same question, taking the client's answer, and then finally sending the client a result such as

> The answer to "How old are you?" is 40.

Such a program can be implemented by the arrow

$$\begin{array}{l} arr\ (\lambda z \rightarrow \text{“What is your question?”}) \ggg ask \ggg \\ (arr\ id \mathbin{\&\&\&} ask) \ggg \\ arr\ (\lambda(q,a) \rightarrow \text{“The answer to \textbackslash “”} +\!\!+ q +\!\!+ \text{“\textbackslash ” is ”} +\!\!+ a) \end{array}$$

Why choose the arrow interface rather than the monad interface for this problem? The key observation guiding the choice was that the combinators need to save the entire state of the program at an *ask* operation, which is difficult because a part of the program state may be held in free variables. We need only be concerned here with variables bound to the results of computations, since it is only these that may have a different value the next time the program is run. The monadic interface permits such variables to scope over computations, and in particular over *ask* operations, which means that their values must be part of the saved state. But the arrow interface does not permit this: the only way to bind a variable to the result of a computation is with the *arr* combinator, but then the scope of the variable cannot extend over an *ask* operation.

How, then, can CGI arrows be represented? When such an arrow is invoked, it may either terminate normally, producing a result, or it may suspend at an *ask* operation. On suspension, an arrow must produce a state to save, and a question to ask. A CGI arrow can also be entered in two different ways: it may either be entered normally, with an argument, or it may be resumed from an *ask*. In the latter case we must supply a state to resume from, and the answer to the question. A natural representation for CGI arrows might therefore be

$$\textbf{newtype}\ \mathit{CGI}\ b\ c = \mathit{CGI}\ (\mathit{Either}\ b\ (\mathit{State}, \mathit{String}) \to \mathit{Either}\ c\ (\mathit{State}, \mathit{String}))$$

However, in general a CGI program may have side-effects on the server, which this type does not allow for. So we shall instead represent CGI arrows as *arrows* between these two types, which in practice will be arrows which can perform I/O. We shall parameterise our definitions on the underlying arrow type, and so define a CGI functor:

$$\begin{aligned}&\textbf{newtype}\ \mathit{CGIFunctor}\ a\ b\ c = \\ &\quad \mathit{CGI}\ (a\ (\mathit{Either}\ b\ (\mathit{State}, \mathit{String}))\ (\mathit{Either}\ c\ (\mathit{State}, \mathit{String})))\end{aligned}$$

Now we can define

$$\textbf{type}\ \mathit{CGI}\ b\ c = \mathit{CGIFunctor}\ (\mathit{Kleisli}\ \mathit{IO})\ b\ c$$

With this definition, the *ask* operation is easily defined: it suspends when entered normally, and delivers the answer as its result when it is resumed. No state is needed to resume the *ask* operator itself, so we assume that the *State* type includes a constructor *Empty*:

$$\textbf{data}\ \mathit{State} = \ \mathit{Empty} \mid \ldots$$

We define *ask* as follows:

$$\begin{aligned}&\mathit{ask} :: \mathit{ArrowChoice}\ a \Rightarrow \mathit{CGIFunctor}\ a\ \mathit{String}\ \mathit{String} \\ &\mathit{ask} = \mathit{CGI}\ (\mathit{arr}\ (\lambda q \to \mathit{Right}\ (\mathit{Empty}, q)) \mid\mid\mid \mathit{arr}\ (\lambda(\mathit{Empty}, a) \to \mathit{Left}\ a))\end{aligned}$$

The first alternative here handles a normal entry, and suspends to ask the question q, while the second alternative handles a resumption, and delivers the answer a as the arrow's result.

The *arr* operator is also easily defined: a pure arrow can never suspend, and therefore can never be resumed either, so we need consider only the *Left* summands here.

$$\begin{aligned}&arr :: Arrow\ a \Rightarrow (b \to c) \to CGIFunctor\ a\ b\ c\\ &arr\ f = CGI\ (arr\ (\lambda(Left\ b) \to Left\ (f\ b)))\end{aligned}$$

It is when we define arrow composition that we first need to make use of the state. A composition of arrows may suspend either in the first arrow, or in the second, and the state that we save must record which case applied. Similarly, when we resume a composition of arrows, then we need to know which arrow to resume. We shall therefore extend the *State* type to record this information:

$$\mathbf{data}\ State = \ Empty \mid InLeft\ State \mid InRight\ State \mid \ldots$$

The definition of composition then becomes

$$\begin{aligned}&(\ggg) :: ArrowChoice\ a \Rightarrow\\ &\quad CGIFunctor\ a\ b\ c \to CGIFunctor\ a\ c\ d \to CGIFunctor\ a\ b\ d\\ &CGI\ f \ggg CGI\ g = CGI\ (\ (arr\ Left \ggg enterf) \mid\mid\mid\\ &\qquad\qquad (\ (arr\ (\lambda(s,a) \to\\ &\qquad\qquad\quad \mathbf{case}\ s\ \mathbf{of}\\ &\qquad\qquad\qquad InLeft\ s' \ \to\ Left\ (Right\ (s',a))\\ &\qquad\qquad\qquad InRight\ s' \ \to\ Right\ (Right\ (s',a)))) \ggg\\ &\qquad\qquad (enterf \mid\mid\mid enterg)))\\ &\quad \mathbf{where}\ \ enterf = f \ggg (\ (arr\ Left \ggg enterg) \mid\mid\mid\\ &\qquad\qquad\qquad\qquad (arr\ (\lambda(s,q) \to Right\ (InLeft\ s,q))))\\ &\qquad\quad enterg = g \ggg (arr\ Left \mid\mid\mid arr\ (\lambda(s,q) \to Right\ (InRight\ s,q)))\end{aligned}$$

The first case in $\ggg$ handles initial entry to the composition, and just makes an initial entry to f. The second case handles resumption: it tests to see which of f and g should be resumed, and sends a resumption state to the appropriate one. Arrow *enterf* invokes f, and if f terminates normally, makes an initial entry to g. If f suspends, on the other hand, then *enterf* records that the suspension occurred in the left operand of $\ggg$. Arrow *enterg* similarly records that a suspension in g occurred in the right operand of $\ggg$. Thus we always record in which arrow a suspension occurred, and on resumption we return to the same point.

When we define the *first* combinator, we need to use the state in a different way. There is no need to record where a suspension occurred: when *first* f suspends, it must be in the arrow f. However, since *first* f must preserve the second component of its input unchanged, then when we resume after a suspension, we need to know what the value of this second component was. We

therefore have to save it in the state. One difficulty is that the values to be saved can have many different types, at different occurrences of *first*. We shall convert them all to the *same* type before saving them; since states must eventually be embedded in HTML fields, it is convenient to convert them to strings, using Haskell's standard function *show*. When we resume from such a state, we can convert the saved value back to its original type using the standard function *read*, which satisfies $read \circ show = id$.

We shall therefore extend the *State* type again:

$$\textbf{data}\ State = \ Empty \mid InLeft\ State \mid InRight\ State \mid Save\ String\ State$$

and define *first* as

```
first :: (Arrow a, Show d, Read d) ⇒
      CGIFunctor a b c → CGIFunctor a (b,d) (c,d)
first (CGI f) =
   CGI ( arr (λx →  case x of
                       Left (b,d) → (Left b,d))
                       Right (Save v s,a) → (Right (s,a), read v))) ⋙
         first f ⋙
         arr (λ(x,d) →  case x of
                           Left c      →  Left (c,d)
                           Right (s,q) →  Right (Save (show d) s,q)))
```

On an initial entry to *first* f, we just pass the first component of the input to f; on a resumption we reconstruct the saved second component from the state. On final termination of f, we just pair its output with the second component d, but on suspension we save d in the state. Notice that the type d must support *read* and *show* operations, which not all types do. This is recorded in the type signature of *first*, which requires d to be an instance of the classes *Read* and *Show*.

CGI arrows also permit dynamic choices. Implementing *left* turns out to be particularly simple, because *left* f can suspend only if the input was of the form *Left* b; we therefore don't need to record any additional information in the state to allow us to decide whether or not to invoke f on a resumption.

```
left :: ArrowChoice a ⇒
        CGIFunctor a b c → CGIFunctor a (Either b d) (Either c d)
left (CGI f) =
    CGI ( arr (λx →  case x of
                       Left (Left b)    →  Left (Left b)
                       Left (Right d)   →  Right d
                       Right (s, a)     →  Left (Right (s, a))) ⋙
          left f ⋙
          arr (λx →  case x of
                       Left (Left c)        →  Left (Left c)
                       Left (Right (s, q))  →  Right (s, q)
                       Right d              →  Left (Right d)))
```

On an initial entry to *left* f, we pass inputs tagged *Left* to f, and those tagged *Right* are passed through unchanged. On a resumption of *left* f, we just resume f. When f terminates normally, or the input was tagged *Right*, then *left* f terminates. When f suspends then so does *left* f, in the same state.

It is also possible to give an appealing interpretation of *zeroArrow* and $+\!\!+\!\!+$ for CGI arrows: $f +\!\!+\!\!+ g$ creates two *threads* which run in parallel, and *zeroArrow* terminates a thread. We use this mechanism to enable a CGI arrow to ask several questions in one interaction (if both f and g suspend). We omit the details here.

It is not possible, however, to implement *app*. The difficulty here is that the types that CGI arrows operate over must support *read* and *show*, so that intermediate values can be saved on the client. CGI arrows themselves are implemented in terms of functions, and so cannot be read and written. Therefore a CGI arrow cannot take another CGI arrow in its input, and *app* cannot be defined.

The library I am developing is based on the ideas in this section, but is necessarily a little more complicated. It is an oversimplification to consider the communication with the client to consist of a single question and answer, or even multiple questions and answers. In reality the client is sent an HTML page containing one or more HTML forms, each of which may contain many fields. The full-scale library includes combinators for generating various HTML elements, and for putting parts of forms together into larger forms. There is also a 'top-level' function

```
serveCGI :: CGI a b → IO ()
```

which takes an arrow and 'runs it', taking care of encoding states in hidden fields, decoding the data returning from the client, and so on.

One major irritation which we have so far glossed over is that CGI arrows cannot actually be made an instance of the *Arrow* class defined in this paper! The problem lies in the types of the arrow methods given in this section. Look back at the type of *first*: it requires that the type of the value to be saved be an

instance of the *Read* and *Show* classes. The type given for *first* in the definition of the *Arrow* class makes no such restriction. Therefore this implementation of *first* cannot be declared to be an instance of the generic one — it is less general.

We might attempt to solve this problem by moving the restriction to a different place. Let us define the CGI arrow type so that it is only applicable to types in these classes:

```
newtype (Read b, Show b, Read c, Show c) ⇒
    CGIFunctor a b c =
        CGI (a (Either b (State, String)) (Either c (State, String)))
```

In categorical terms, we define a new category whose arrows are CGI arrows, and whose objects are a *subset* of the Haskell types, namely those supporting *read* and *show*. Now, since the implementation of *first* given in this section constructs a CGI arrow from (b, d) to (c, d), then it is evident that the type d must support *read* and *show*, and there is no need to explicitly require that in the type of *first*. As a result, it should now be possible to declare CGI arrows an instance of the generic arrow class.

Unfortunately, this does not work. The Haskell type system requires the restrictions on d in the type of *first*, even if we declare that they are satisfied for all CGI arrows. Haskell does not infer from the occurrence of a type *CGI b c*, that b and c must be instances of *Read* and *Show* — and indeed, this is not even true, because of the way that type restrictions on datatype definitions are interpreted. I consider this to be a defect of the Haskell type system, which hopefully can be corrected in a future version of the language.

In the absence of such a correction, we are obliged to make a copy of the arrow library, and all the generic code that uses it, with the only difference that the type assigned to *first* in the *Arrow* class is the one required for the CGI instance. By doing so we can still benefit from using a standard arrow interface to the CGI library — we can still combine CGI arrows with other arrow code — but any program which uses the CGI library must import a special definition of the arrow class, which restricts *all* arrows in the entire program to work over types supporting *read* and *show*. This is frustrating indeed.

Finally, we note with hindsight that a monadic interface *could* be used instead here. We could define a monad whose computations can be suspended and resumed, in an analogous way to CGI arrows. However, the definition of $m \gg\!= f$ would need to record not only which of m or f suspended, but also the *value* that m delivered, if suspension occurred in f. Concretely, the '*InRight*' form of *State* would need to carry an extra component, namely the value of m. Thus the problem of recording free variables is solved: every free variable of an *ask* operation *which is bound to the result of a computation*, is bound by an occurrence of $\gg\!=$, and we can make that occurrence of $\gg\!=$ responsible for saving the value.

However, even if a monadic interface would be possible, we believe it would make for less efficient CGI programs. The monadic library we suggest would

need to save *every* previously delivered value, whereas the arrow library saves only those which are still needed. Thus the monadic library would tend to send more information to and from the client. Of course, such a monadic library would also fall foul of the typing problem just discussed, so that a CGI monad could not be declared to be an instance of Haskell's *Monad* class. Consequently it could not be used together with standard monadic functions, or Haskell's monadic **do** syntax.

10 Conclusions

This paper proposes the replacement of monads as a structuring tool for combinator libraries, by arrows. We have seen that any monadic library can be given an arrow interface instead (via *Kleisli* arrows), and so the arrow interface is strictly more general. We have seen that many monadic programming techniques have analogues in the world of arrows: monad transformers become functors, standard monad constructions for exceptions, state passing and continuations carry over to arrows, even generic monadic functions often have an arrow analogue. But basing an interface on arrows instead of monads permits finer distinctions to be made: we can distinguish between kinds of computation which permit dynamic choices to be made, or dynamic computations to be invoked, and those which do not.

The advantage of the arrow interface is that it has a wider class of implementations than the monad interface does; it is more general. Thus some libraries based on abstract data types which simply are not monads, can nonetheless be given an arrow interface. Such libraries include those for processes modelled by stream processors or fudgets, libraries for efficient parsing, or in general any library which computes static properties of computations in advance of running them. So this category includes a number of libraries which are highly useful in practice. By giving them an arrow interface, we make it possible to use them together with generic arrow code.

Moreover, some existing *monadic* libraries might benefit by replacing the monads with arrows. One motivation might be in order to introduce the same kind of optimisation which Swierstra and Duponcheel used. We believe this may be the case for Conal Elliot's animation library [EH97], and for Bjesse et als library for hardware design [BCSS98].

In short, we believe that arrows offer a useful extension to the generality of generic library interfaces.

References

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In *International Conference on Functional*

Programming, Baltimore, 1998. ACM.

[CH93] M. Carlsson and T. Hallgren. FUDGETS - A Graphical User Interface in a Lazy Functional Language. In *FPCA '93 - Conference on Functional Programming Languages and Computer Architecture*, pages 321–330. ACM Press, June 1993.

[EH97] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *International Conference on Functional Programming*. ACM SIGPLAN, 1997.

[Hud92] Paul Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[KW92] David King and Phil Wadler. Combining Monads. In *Glasgow Workshop on Functional Programming*, Ayr, July 1992. Springer-Verlag.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995.

[PH96] J. Peterson and K. Hammond. The Haskell 1.3 Report. Technical Report YALEU/DCS/RR-1106, Yale University, 1996.

[SD96] S.D. Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.

[Wad85] P. Wadler. How to Replace Failure by a List of Successes. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, Nancy, France, 1985.

[Wad90] P. Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–77, Nice, France, 1990.

[Wad92] Phil Wadler. The essence of functional programming. In *Proceedings 1992 Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, 1992.

[Wad95] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in LNCS, pages 24–52. Springer Verlag, May 1995.