Composing monads*

Mark P. Jones
Yale University
New Haven, CT, U.S.A.
jones-mark@cs.yale.edu

Luc Duponcheel
Alcatel Bell Telephone
Antwerp, Belgium
ldup@ra.alcbel.be

Research Report YALEU/DCS/RR-1004, December 1993

Abstract

Monads are becoming an increasingly important tool for functional programming. Different monads can be used to model a wide range of programming language features. However, real programs typically require a combination of different features, so it is important to have techniques for combining several features in a single monad. In practice, it is usually possible to construct a monad that supports some specific combination of features. However, the techniques used are typically ad-hoc and it is very difficult to find general techniques for combining arbitrary monads.

This report gives three general constructions for the composition of monads, each of which depends on the existence of an auxiliary function linking the monad structures of the components. In each case, we establish a set of laws that the auxiliary function must satisfy to ensure that the composition is itself a monad.

Using the notation of constructor classes, we describe some specific applications of these constructions. These results are used in the development of a simple expression evaluator that combines exceptions, output and an environment of variable bindings using a composition of three corresponding monads.

1 Introduction

In recent years, the concept of a monad – an idea that was originally motivated by high-level abstract algebra – has become an important and practical tool for functional programmers. The reason for this is that monads provide a uniform framework for describing a wide range of programming language features including, for example, state, I/O, continuations, exceptions, parsing and non-determinism, without leaving the framework of a purely functional language. Many of these techniques were already familiar to functional programmers, but there are many new insights when they are reinterpreted as specific instances of a more general concept.

*A Gofer script containing executable versions of the programs described in this report is currently available by anonymous ftp from `nebula.cs.yale.edu` in the file `pub/yale-fp/reports/RR-1004.gs`.

Much of the initial interest in monads has been motivated by the work of Wadler [14, 15] who, in turn, drew inspiration from the work of Moggi [9] and Spivey [12]. Monads are already widely used in both small and large programs (for example, the Glasgow Haskell compiler, the largest Haskell program known to us at the time of writing, makes substantial use of monads [2]). New approaches to old problems have been proposed, relying heavily on the use of monads. For example, the I/O monad proposed in [11] is already widely used and may soon be included as part of the definition of Haskell. Monads are even influencing the design of programming languages. For example, monads provide an important motivating example for the system of constructor classes presented in [5].

With the exception of [8], questions about how monads can be combined have, so far, received surprisingly little attention. This is an important topic because many real programs require a combination of features, for example, state, I/O and exceptions. In practice, it is usually possible to construct a suitable monad that supports the desired combination of features, but the methods used are typically ad-hoc and monolithic.

The goal of this report is to investigate some techniques for combining monads by a process of *composition* and to illustrate how these constructions can be used in practice. The conditions required to build a composite monad are quite complex, but we hope that this work will provide a step towards a more modular approach to the use of monads.

One of the nicest features about monads is that, with a little practice (and perhaps, some carefully chosen syntax), it is actually quite easy to write programs in a monadic style without any knowledge of the abstract theoretical underpinnings. In the same spirit, this report is directed at functional programmers with an interest in using monads as part of a practical programming project, rather than the underlying category theory. To this end, in a number of places, we have intentionally chosen to give definitions or results without reference to the corresponding categorical concepts. Readers with an interest in a more technical, category theoretic presentation of the ideas described in this report are referred to [1].

We will assume some familiarity with the motivation for monads and their use in structuring functional programs; Wadler [14, 15] provides an excellent introduction to these topics. Programming examples will be written using the syntax of Gofer, a small, experimental, purely functional language based closely on the definition of Haskell [3]. This enables us to use constructor classes [5] to show how our results can be expressed in a concrete programming language.

For completeness, we have included detailed proofs for many of our results. In keeping with our aim to avoid unnecessary technical details, most of the proofs are constructed from first principles using simple equational reasoning. Working in this manner has lead to some surprising insights. For example, one of our results in Section 3 corresponds closely to a result stated in [8], but a careful study of the laws that are actually used in the proof has allowed us to weaken the hypotheses and state the result in a slightly more general form. However, recognizing that some readers may prefer to omit such details, most of the proofs are presented in boxed figures that are easily identified and skipped.

The remainder of this report is organized as follows. Section 2 gives a definition of the algebraic properties of a monad, and these are then used to describe the construction of compositions of monads in Section 3 and their converses in Section 4. Moving on to practical applications, Section 5 describes how these results about composition of monads can be expressed using the notation of constructor classes, with several examples in Section 6. A simple application, building on this framework, is included in Section 7. Finally, Section 8 illustrates that there are other ways of combining certain monads, setting a direction for future work.

2 Monads for functional programming

Wadler [14] defines a *monad* as a unary type constructor M together with three functions *map*, *unit* and *join* whose types are given by:

$$\begin{array}{lcl} map & :: & (a \to b) \to (M\ a \to M\ b) \\ unit & :: & a \to M\ a \\ join & :: & M\ (M\ a) \to M\ a \end{array}$$

In addition, these functions are required to satisfy a collection of algebraic laws:

$$\begin{array}{lcll} map\ id & = & id & (1) \\ map\ f\ .\ map\ g & = & map\ (f\ .\ g) & (2) \\[1ex] unit\ .\ f & = & map\ f\ .\ unit & (3) \\[1ex] join\ .\ map\ (map\ f) & = & map\ f\ .\ join & (4) \\ join\ .\ unit & = & id & (5) \\ join\ .\ map\ unit & = & id & (6) \\ join\ .\ map\ join & = & join\ .\ join & (7) \end{array}$$

(We use the standard infix period notation for function composition, $(f\ .\ g)\ x = f\ (g\ x)$, and the symbol id to represent the identity function $id\ x = x$. It is well known that composition of functions is associative with id as both a left and right identity; in symbols, $f\ .\ (g\ .\ h) = (f\ .\ g)\ .\ h$ and $f\ .\ id = f = id\ .\ f$ for all f, g and h of appropriate types.)

For the purposes of this report, it will be convenient to break this down into stages; if M is a unary type constructor, then we will say that:

- M is a *functor* if there is a function $map :: (a \to b) \to (M\ a \to M\ b)$ satisfying laws (1) and (2) above.

- M is a *premonad* if it is a functor with a function $unit :: a \to M\ a$ satisfying law (3) above.

- M is a *monad* if it is a premonad with a function $join :: M\ (M\ a) \to M\ a$ satisfying laws (4), (5), (6) and (7) above.

It should be mentioned that there are several other (equivalent) ways to define the concept of a monad, each of which comes with its own collection of monad operators and equational laws. For example, Wadler [15] describes how a monad can be characterized using just the *unit* function together with an operator

$$bind \quad :: \quad M\ \ a \rightarrow (a \rightarrow M\ \ b) \rightarrow M\ \ b$$

Both the *map* and *join* operators that we have described can be defined using *bind* and *unit*. The *bind* operator is particularly useful in practical work with monads and as a means of translating the notation of monad comprehensions [14, 15, 5]. Furthermore, only three laws are needed to specify the properties of a monad in this case. Nevertheless, we have chosen to work with the *map*, *unit*, *join* formulation of monads outlined above. One reason for this decision is that, in our opinion, many of the proofs are easier to express in this framework. In addition, we will see that the ability to distinguish between monads and premonads will also be quite useful in the following work.

3 Conditions for composition

Suppose that M and N are functors. To avoid confusion, we will write map_M and map_N for the corresponding *map* functions in each case. How is it possible to compose these functors in some sensible way? Certainly, we can think of a composition of M and N as a type constructor that takes any type a to the type $M\ \ (N\ \ a)$, but we also need to be able to give a definition for

$$map \quad :: \quad (a \rightarrow b) \rightarrow (M\ \ (N\ \ a) \rightarrow M\ \ (N\ \ b))$$

satisfying the functor laws (1) and (2). Fortunately, this is quite easy! If $f :: a \rightarrow b$, then we can apply map_N to obtain $map_N\ f :: N\ \ a \rightarrow N\ \ b$ and then apply map_M to obtain $map_M\ \ (map_N\ f) :: M\ \ (N\ \ a) \rightarrow M\ \ (N\ \ b)$. This gives the definition

$$map \quad = \quad map_M\ .\ map_N$$

and we can use the proofs in Figure 1 to show that this does satisfy the necessary laws.

These proofs use standard techniques of equational reasoning, writing the justification for each step in the right hand column. In many cases, this is just a reference to the law that has been used, with the convention that, for example, (1M) is the version of law (1) for the constructor M. Where a step follows directly from the definition of a function (either by folding or unfolding the definition), we will simply write the name of the function involved as justification.

Composition of premonads is similar. Most of the work (the definition of the composed type constructor) has already been dealt with in the composition of functors. A suitable *unit* function for the composition is:

$$\begin{array}{lcl} unit & :: & a \rightarrow M\ \ (N\ \ a) \\ unit & = & unit_M\ .\ unit_N \end{array}$$

$$
\begin{array}{llll}
map\ id & = & map_M\ (map_N\ id) & map \\
 & = & map_M\ id & (1N) \\
 & = & id & (1M) \\
 & & & \\
map\ f\ .\ map\ g & = & map_M\ (map_N\ f)\ .\ map_M\ (map_N\ g) & map,\ map \\
 & = & map_M\ (map_N\ f\ .\ map_N\ g) & (2M) \\
 & = & map_M\ (map_N\ (f\ .\ g)) & (2N) \\
 & = & map\ (f\ .\ g) & map
\end{array}
$$

Figure 1: Composition of functors, laws (1)–(2)

($unit_M$ and $unit_N$ being the unit functions for M and N respectively) and the proof in Figure 2 demonstrates that this does indeed satisfy law (3).

$$
\begin{array}{llll}
unit\ .\ f & = & unit_M\ .\ unit_N\ .\ f & unit \\
 & = & unit_M\ .\ map_N\ f\ .\ unit_N & (3N) \\
 & = & map_M\ (map_N\ f)\ .\ unit_M\ .\ unit_N & (3M) \\
 & = & map\ f\ .\ unit & map,\ unit
\end{array}
$$

Figure 2: Composition of premonads, law (3)

Unfortunately, our real goal, composition of monads, is rather more difficult. Recall that, to define the composition of two monads M and N, we need to find a function:

$$join \ :: \ M\ (N\ (M\ (N\ a))) \rightarrow M\ (N\ a)$$

that satisfies the monad laws (4)–(7). As a first guess, and following the pattern of the previous examples, we might consider the function $join_M\ .\ join_N$ where $join_M$ and $join_N$ are the *join* functions in the component monads. But this is, in general, not even type correct! If $x :: N\ (N\ a)$ (for some a), then the result of $join_N\ x$ will have type $N\ a$ while $join_M$ expects an argument with a type of the form $M\ (M\ b)$.

In fact, we can actually prove that, in a certain sense, there is no way to construct a *join* function with the type above using only the operations of the two monads (see the appendix for an outline of the proof). It follows that the only way that we might hope to form a composition is if there are some additional constructions linking the two components. In this report we will concentrate on four methods for constructing a composite monad, described in the following sections.

3.1 The trivial case, by definition

The composite of two premonads M and N is a monad if there is a polymorphic function:

$$join \;\; :: \;\; M\ (N\ (M\ (N\ a))) \to M\ (N\ a)$$

satisfying the laws (4), (5), (6) and (7). This, of course, follows directly from the definitions above.

3.2 The *prod* construction

The composition of a monad M with a premonad N is itself a monad if there is a polymorphic function:

$$prod \;\; :: \;\; N\ (M\ (N\ a)) \to M\ (N\ a)$$

with the *join* function defined by:

$$join \;\; = \;\; join_M\ .\ map_M\ prod$$

satisfying the following four laws:

$$\begin{array}{lcll}
prod\ .\ map_N\ (map\ f) & = & map\ f\ .\ prod & \mathrm{P}(1) \\
prod\ .\ unit_N & = & id & \mathrm{P}(2) \\
prod\ .\ map_N\ unit & = & unit_M & \mathrm{P}(3) \\
prod\ .\ map_N\ join & = & join\ .\ prod & \mathrm{P}(4)
\end{array}$$

The proofs for this are given in Figure 3. One interesting point is that, when we started this work, we had assumed that it would be necessary to require that both M and N be monads. In fact, from the proofs, we see that we did not actually need any of the monad laws for N at all! As a result, we can construct a 'composition of monads' under somewhat weaker conditions than originally expected.

3.3 The *dorp* construction

The composition of a premonad M with a monad N is itself a monad if there is a polymorphic function:

$$dorp \;\; :: \;\; M\ (N\ (M\ a)) \to M\ (N\ a)$$

with the *join* function defined by:

$$join \;\; = \;\; map_M\ join_N\ .\ dorp$$

such that the following laws hold:

$$\begin{array}{lcll}
dorp\ .\ map\ (map_M\ f) & = & map\ f\ .\ dorp & \mathrm{D}(1) \\
dorp\ .\ unit & = & map_M\ unit_N & \mathrm{D}(2) \\
dorp\ .\ map\ unit_M & = & id & \mathrm{D}(3) \\
dorp\ .\ join & = & join\ .\ map\ dorp & \mathrm{D}(4)
\end{array}$$

Full proofs for this are given in Figure 4. In this case, we use the fact that N is a monad, although M can be an arbitrary premonad.

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathit{join}\ .\ \mathit{map}\ (\mathit{map}\ f)} & \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ \mathit{prod}\ .\ \mathit{map}_M\ (\mathit{map}_N\ (\mathit{map}\ f)) & \mathit{join},\ \mathit{map} \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ (\mathit{prod}\ .\ \mathit{map}_N\ (\mathit{map}\ f)) & \text{(2M)} \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ (\mathit{map}\ f\ .\ \mathit{prod}) & \text{P(1)} \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ (\mathit{map}\ f)\ .\ \mathit{map}_M\ \mathit{prod} & \text{(2M)} \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ (\mathit{map}_M\ (\mathit{map}_N\ f))\ .\ \mathit{map}_M\ \mathit{prod} & \mathit{map} \\
\quad = & \mathit{map}_M\ (\mathit{map}_N\ f)\ .\ \mathit{join}_M\ .\ \mathit{map}_M\ \mathit{prod} & \text{(4M)} \\
\quad = & \mathit{map}\ f\ .\ \mathit{join} & \mathit{map},\ \mathit{join} \\
& & \\
\multicolumn{2}{l}{\mathit{join}\ .\ \mathit{unit}} & \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ \mathit{prod}\ .\ \mathit{unit}_M\ .\ \mathit{unit}_N & \mathit{join},\ \mathit{unit} \\
\quad = & \mathit{join}_M\ .\ \mathit{unit}_M\ .\ \mathit{prod}\ .\ \mathit{unit}_N & \text{(3M)} \\
\quad = & \mathit{prod}\ .\ \mathit{unit}_N & \text{(5M)} \\
\quad = & \mathit{id} & \text{P(2)} \\
& & \\
\multicolumn{2}{l}{\mathit{join}\ .\ \mathit{map}\ \mathit{unit}} & \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ \mathit{prod}\ .\ \mathit{map}_M\ (\mathit{map}_N\ \mathit{unit}) & \mathit{join},\ \mathit{map} \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ (\mathit{prod}\ .\ \mathit{map}_M\ \mathit{unit}) & \text{(2M)} \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ \mathit{unit}_M & \text{P(3)} \\
\quad = & \mathit{id} & \text{(6M)} \\
& & \\
\multicolumn{2}{l}{\mathit{join}\ .\ \mathit{map}\ \mathit{join}} & \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ \mathit{prod}\ .\ \mathit{map}_M\ (\mathit{map}_N\ \mathit{join}) & \mathit{join},\ \mathit{map} \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ (\mathit{prod}\ .\ \mathit{map}_N\ \mathit{join}) & \text{(2M)} \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ (\mathit{join}\ .\ \mathit{prod}) & \text{P(4)} \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ (\mathit{join}_M\ .\ \mathit{map}_M\ \mathit{prod}\ .\ \mathit{prod}) & \mathit{join} \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ \mathit{join}_M\ .\ \mathit{map}_M\ (\mathit{map}_M\ \mathit{prod})\ .\ \mathit{map}_M\ \mathit{prod} & \text{(2M)} \\
\quad = & \mathit{join}_M\ .\ \mathit{join}_M\ .\ \mathit{map}_M\ (\mathit{map}_M\ \mathit{prod})\ .\ \mathit{map}_M\ \mathit{prod} & \text{(7M)} \\
\quad = & \mathit{join}_M\ .\ \mathit{map}_M\ \mathit{prod}\ .\ \mathit{join}_M\ .\ \mathit{map}_M\ \mathit{prod} & \text{(4M)} \\
\quad = & \mathit{join}\ .\ \mathit{join} & \mathit{join},\ \mathit{join}
\end{array}
$$

Figure 3: Composition of monads, laws (4)–(7), with $\mathit{join} = \mathit{join}_M\ .\ \mathit{map}_M\ \mathit{prod}$

$$
\begin{array}{lll}
\multicolumn{2}{l}{join\ .\ map\ (map\ f)} & \\
\quad = & map_M\ join_N\ .\ dorp\ .\ map\ (map_M\ (map_N\ f)) & join,\ map \\
\quad = & map_M\ join_N\ .\ map\ (map_N\ f)\ .\ dorp & \mathrm{D}(1) \\
\quad = & map_M\ join_N\ .\ map_M\ (map_N\ (map_N\ f))\ .\ dorp & map \\
\quad = & map_M\ (join_N\ .\ map_N\ (map_N\ f))\ .\ dorp & (2\mathrm{M}) \\
\quad = & map_M\ (map_N\ f\ .\ join_N)\ .\ dorp & (4\mathrm{N}) \\
\quad = & map_M\ (map_N\ f)\ .\ map_M\ join_N\ .\ dorp & (2\mathrm{M}) \\
\quad = & map\ f\ .\ join & map,\ join \\
 & & \\
\multicolumn{2}{l}{join\ .\ unit} & \\
\quad = & map_M\ join_N\ .\ dorp\ .\ unit & join \\
\quad = & map_M\ join_N\ .\ map_M\ unit_N & \mathrm{D}(2) \\
\quad = & map_M\ (join_N\ .\ unit_N) & (2\mathrm{M}) \\
\quad = & map_M\ id & (5\mathrm{N}) \\
\quad = & id & (1\mathrm{M}) \\
 & & \\
\multicolumn{2}{l}{join\ .\ map\ unit} & \\
\quad = & map_M\ join_N\ .\ dorp\ .\ map\ (unit_M\ .\ unit_N) & join,\ unit \\
\quad = & map_M\ join_N\ .\ dorp\ .\ map\ unit_M\ .\ map\ unit_N & (2) \\
\quad = & map_M\ join_N\ .\ map\ unit_N & \mathrm{D}(3) \\
\quad = & map_M\ join_N\ .\ map_M\ (map_N\ unit_N) & map \\
\quad = & map_M\ (join_N\ .\ map_N\ unit_N) & (2\mathrm{M}) \\
\quad = & map_M\ id & (6\mathrm{N}) \\
\quad = & id & (1\mathrm{M}) \\
 & & \\
\multicolumn{2}{l}{join\ .\ map\ join} & \\
\quad = & map_M\ join_N\ .\ dorp\ .\ map\ (map_M\ join_N\ .\ dorp) & join,\ join \\
\quad = & map_M\ join_N\ .\ dorp\ .\ map\ (map_M\ join_N)\ .\ map\ dorp & (2) \\
\quad = & map_M\ join_N\ .\ map\ join_N\ .\ dorp\ .\ map\ dorp & \mathrm{D}(1) \\
\quad = & map_M\ join_N\ .\ map_M\ (map_N\ join_N)\ .\ dorp\ .\ map\ dorp & map \\
\quad = & map_M\ (join_N\ .\ map_N\ join_N)\ .\ dorp\ .\ map\ dorp & (2\mathrm{M}) \\
\quad = & map_M\ (join_N\ .\ join_N)\ .\ dorp\ .\ map\ dorp & (7\mathrm{N}) \\
\quad = & map_M\ join_N\ .\ map_M\ join_N\ .\ dorp\ .\ map\ dorp & (2\mathrm{M}) \\
\quad = & map_M\ join_N\ .\ join\ .\ map\ dorp & join \\
\quad = & map_M\ join_N\ .\ dorp\ .\ join & \mathrm{D}(4) \\
\quad = & join\ .\ join & join
\end{array}
$$

Figure 4: Composition of monads, laws (4)–(7), with $join = map_M\ join_N\ .\ dorp$

3.4 The *swap* construction

We can also define the composition of two monads M and N in terms of a polymorphic function:

$$swap \ :: \ N\ (M\ a) \rightarrow M\ (N\ a)$$

with the *join* function defined by:

$$join \ = \ map_M\ join_N \ .\ join_M \ .\ map_M\ swap$$

satisfying a collection of laws given below. Note that this is equivalent to:

$$join \ = \ join_M \ .\ map_M\ (map_M\ join_N \ .\ swap)$$

since:

$$\begin{array}{lll} & map_M\ join_N \ .\ join_M \ .\ map_M\ swap & \\ = & join_M \ .\ map_M\ (map_M\ join_N)\ .\ map_M\ swap & \text{(4M)} \\ = & join_M \ .\ map_M\ (map_M\ join_N \ .\ swap). & \text{(2M)} \end{array}$$

In order to state the laws for *swap*, we define two (familiarly named) functions as abbreviations for expressions involving *swap*:

$$\begin{array}{lcl} prod & = & map_M\ join_N \ .\ swap \\ dorp & = & join_M \ .\ map_M\ swap \end{array}$$

We will justify the use of these names below, but first we state the laws that the *swap* function must satisfy:

$$\begin{array}{lcll} swap \ .\ map_N\ (map_M\ f) & = & map_M\ (map_N\ f)\ .\ swap & \text{S(1)} \\ swap \ .\ unit_N & = & map_M\ unit_N & \text{S(2)} \\ swap \ .\ map_N\ unit_M & = & unit_M & \text{S(3)} \\ prod \ .\ map_N\ dorp & = & dorp \ .\ prod & \text{S(4)} \end{array}$$

It is possible to prove the existence of the composite monad using these definitions alone. However, it is easier, and perhaps more instructive, to do this indirectly by exploring the relationship between *swap*, *prod* and *dorp*.

First of all, note that the definition of *join* coincides with the *join* that would be obtained using the *prod* construction (with the above definition of *prod*):

$$\begin{array}{lcll} join_M \ .\ map_M\ prod & = & join_M \ .\ map_M\ (map_M\ join_N \ .\ swap) & prod \\ & = & join & join \end{array}$$

Furthermore, the definition of *join* also coincides with the *join* function that we would obtain using the *dorp* construction (with the above definition of *dorp*):

$$\begin{array}{lcll} map_M\ join_N \ .\ dorp & = & map_M\ join_N \ .\ (join_M \ .\ map_M\ swap) & dorp \\ & = & join & join \end{array}$$

Assuming that laws S(1)–S(4) are satisfied, the proofs in Figures 5 and 6 show that *prod* and *dorp* satisfy laws P(1)–P(4) and D(1)–D(4), respectively. Note that, in each case, both M and N must be monads if the composite is also to be a monad. For example, the proof of P(1)–P(4) requires the monad laws for N, while the *prod* construction requires that M should also be a monad.

$$
\begin{array}{lll}
\multicolumn{2}{l}{prod \ . \ map_N \ (map \ f)} & \\
\quad = & map_M \ join_N \ . \ swap \ . \ map_N \ (map_M \ (map_N \ f)) & prod, \ map \\
\quad = & map_M \ join_N \ . \ map_M \ (map_N \ (map_N \ f)) \ . \ swap & \mathrm{S}(1) \\
\quad = & map_M \ (join_N \ . \ map_N \ (map_N \ f)) \ . \ swap & (2\mathrm{M}) \\
\quad = & map_M \ (map_N \ f \ . \ join_N) \ . \ swap & (4\mathrm{N}) \\
\quad = & map_M \ (map_N \ f) \ . \ map_M \ join_N \ . \ swap & (2\mathrm{M}) \\
\quad = & map \ f \ . \ prod & map, \ prod \\
\\
\multicolumn{2}{l}{prod \ . \ unit_N} & \\
\quad = & map_M \ join_N \ . \ swap \ . \ unit_N & prod \\
\quad = & map_M \ join_N \ . \ map_M \ unit_N & \mathrm{S}(2) \\
\quad = & map_M \ (join_N \ . \ unit_N) & (2\mathrm{M}) \\
\quad = & map_M \ id & (5\mathrm{N}) \\
\quad = & id & (1\mathrm{M}) \\
\\
\multicolumn{2}{l}{prod \ . \ map_N \ unit} & \\
\quad = & map_M \ join_N \ . \ swap \ . \ map_N \ (unit_M \ . \ unit_N) & prod, \ unit \\
\quad = & map_M \ join_N \ . \ swap \ . \ map_N \ unit_M \ . \ map_N \ unit_N & (2\mathrm{N}) \\
\quad = & map_M \ join_N \ . \ unit_M \ . \ map_N \ unit_N & \mathrm{S}(3) \\
\quad = & unit_M \ . \ join_N \ . \ map_N \ unit_N & (3\mathrm{M}) \\
\quad = & unit_M & (6\mathrm{N}) \\
\\
\multicolumn{2}{l}{prod \ . \ map_N \ join} & \\
\quad = & prod \ . \ map_N \ (map_M \ join_N \ . \ dorp) & join \\
\quad = & map_M \ join_N \ . \ swap \ . \ map_N \ (map_M \ join_N) \ . \ map_N \ dorp & prod, \ (2\mathrm{N}) \\
\quad = & map_M \ join_N \ . \ map_M \ (map_N \ join_N) \ . \ swap \ . \ map_N \ dorp & \mathrm{S}(1) \\
\quad = & map_M \ (join_N \ . \ map_N \ join_N) \ . \ swap \ . \ map_N \ dorp & (2\mathrm{M}) \\
\quad = & map_M \ (join_N \ . \ join_N) \ . \ swap \ . \ map_N \ dorp & (7\mathrm{N}) \\
\quad = & map_M \ join_N \ . \ map_M \ join_N \ . \ swap \ . \ map_N \ dorp & (2\mathrm{M}) \\
\quad = & map_M \ join_N \ . \ prod \ . \ map_N \ dorp & prod \\
\quad = & map_M \ join_N \ . \ dorp \ . \ prod & \mathrm{S}(4) \\
\quad = & join \ . \ prod & join
\end{array}
$$

Figure 5: Proof of P(1)–P(4) from S(1)–S(4) with $prod = map_M \ join_N \ . \ swap$

$$
\begin{array}{lll}
\multicolumn{2}{l}{dorp\ .\ map\ (map_M\ f)} & \\
\quad = & join_M\ .\ map_M\ swap\ .\ map_M\ (map_N\ (map_M\ f)) & dorp,\ map \\
\quad = & join_M\ .\ map_M\ (swap\ .\ map_N\ (map_M\ f)) & \text{(2M)} \\
\quad = & join_M\ .\ map_M\ (map_M\ (map_N\ f)\ .\ swap) & \text{S(1)} \\
\quad = & join_M\ .\ map_M\ (map_M\ (map_N\ f))\ .\ map_M\ swap & \text{(2M)} \\
\quad = & map_M\ (map_N\ f)\ .\ join_M\ .\ map_M\ swap & \text{(4M)} \\
\quad = & map\ f\ .\ dorp & map,\ dorp \\
 & & \\
\multicolumn{2}{l}{dorp\ .\ unit} & \\
\quad = & join_M\ .\ map_M\ swap\ .\ unit_M\ .\ unit_N & dorp,\ unit \\
\quad = & join_M\ .\ unit_M\ .\ swap\ .\ unit_N & \text{(3M)} \\
\quad = & swap\ .\ unit_N & \text{(5M)} \\
\quad = & map_M\ unit_N & \text{S(2)} \\
 & & \\
\multicolumn{2}{l}{dorp\ .\ map\ unit_M} & \\
\quad = & join_M\ .\ map_M\ swap\ .\ map_M\ (map_N\ unit_M) & dorp,\ map \\
\quad = & join_M\ .\ map_M\ (swap\ .\ map_N\ unit_M) & \text{(2M)} \\
\quad = & join_M\ .\ map_M\ unit_M & \text{S(3)} \\
\quad = & id & \text{(6M)} \\
 & & \\
\multicolumn{2}{l}{dorp\ .\ join} & \\
\quad = & join_M\ .\ map_M\ swap\ .\ join_M\ .\ map_M\ prod & dorp,\ join \\
\quad = & join_M\ .\ join_M\ .\ map_M\ (map_M\ swap)\ .\ map_M\ prod & \text{(4M)} \\
\quad = & join_M\ .\ map_M\ join_M\ .\ map_M\ (map_M\ swap)\ .\ map_M\ prod & \text{(7M)} \\
\quad = & join_M\ .\ map_M\ (join_M\ .\ map_M\ swap\ .\ prod) & \text{(2M)} \\
\quad = & join_M\ .\ map_M\ (dorp\ .\ prod) & dorp \\
\quad = & join_M\ .\ map_M\ (prod\ .\ map_N\ dorp) & \text{S(4)} \\
\quad = & join_M\ .\ map_M\ prod\ .\ map\ dorp & \text{(2M)},\ map \\
\quad = & join\ .\ map\ dorp & join
\end{array}
$$

Figure 6: Proof of D(1)–D(4) from S(1)–S(4) with $dorp = join_M\ .\ map_M\ swap$

3.5 Summary

At first glance, the constructions in the previous sections may seem rather mysterious; in each case, we gave a type for some polymorphic function, stated some laws that it should satisfy … and 'presto!' we have another way of composing monads. In fact, these constructions were discovered largely by experimentation, 'guessing' a definition of *join* in some particular form, for example $join = join_M \ . \ map_M \ prod$, then attempting to prove the monad laws, for example, to determine what properties *prod* should satisfy. Types played an essential part in this process, helping to suggest ways to define *join* and ensuring that the laws we used are well-typed.

The following diagram summarizes these results and the relationship between the different constructions for compositions of monads:

$$
\begin{tikzcd}
 & \boxed{\begin{array}{c} prod :: N\ (M\ (N\ a)) \to M\ (N\ a) \\ \text{satisfying P(1)–P(4)} \end{array}} \arrow[dr, "M"] & \\
\boxed{\begin{array}{c} swap :: N\ (M\ a) \to M\ (N\ a) \\ \text{satisfying S(1)–S(4)} \end{array}} \arrow[ur, "N"] \arrow[rr, "{M,\ N}"] \arrow[dr, "M"'] & & \boxed{\begin{array}{c} join :: M\ (N\ (M\ (N\ a))) \to M\ (N\ a) \\ \text{satisfying (4)–(7)} \end{array}} \\
 & \boxed{\begin{array}{c} dorp :: M\ (N\ (M\ a)) \to M\ (N\ a) \\ \text{satisfying D(1)–D(4)} \end{array}} \arrow[ur, "N"'] &
\end{tikzcd}
$$

In each case, the arrows between different constructions represent implications, with the labels indicating which of the constructors M and N is required to be a monad.

4 Converse results

We now have a number of different ways of composing two monads, but we have not made any attempt to see how general each approach might be. In particular, it is natural to ask what kinds of monads can be obtained using the constructions described above. Thinking of the diagram at the end of the previous section, we know by definition that all compositions of M and N must have a *join* function as specified by the rightmost box. The goal of this section is to establish what conditions are necessary to move back, in the opposite direction of the arrows, to each of the other three constructions.

More formally, suppose that M and N are monads and that there is a composition of M and N with operators *map*, *unit* and *join* satisfying the laws (1)–(7) and such that:

$$
\begin{array}{rcl}
map & = & map_M \ . \ map_N \\
unit & = & unit_M \ . \ unit_N
\end{array}
$$

The problem now is to determine which composite monads defined in this way can be obtained using each of the *prod*, *dorp* and *swap* constructions.

4.1 The *prod* construction

It is actually fairly easy to give a definition for a *prod* function (of the required type) in terms of the various monad operators available:

$$prod \quad = \quad join \ . \ unit_M$$

Showing that this definition of *prod* satisfies the laws P(1)–P(4) is also straightforward, as detailed by the proofs in Figure 7. In fact, the only difficulty arises when we try to

$$\begin{array}{lll}
\multicolumn{2}{l}{map\ f\ .\ prod} & \\
= & map\ f\ .\ join\ .\ unit_M & prod \\
= & join\ .\ map\ (map\ f)\ .\ unit_M & (4) \\
= & join\ .\ map_M\ (map_N\ (map\ f))\ .\ unit_M & map \\
= & join\ .\ unit_M\ .\ map_N\ (map\ f) & (3M) \\
= & prod\ .\ map_N\ (map\ f) & prod \\
\\
\multicolumn{2}{l}{prod\ .\ unit_N} & \\
= & join\ .\ unit_M\ .\ unit_N & prod \\
= & join\ .\ unit & unit \\
= & id & (5) \\
\\
\multicolumn{2}{l}{prod\ .\ map_N\ unit} & \\
= & join\ .\ unit_M\ .\ map_N\ unit & prod \\
= & join\ .\ map_M\ (map_N\ unit)\ .\ unit_M & (3M) \\
= & join\ .\ map\ unit\ .\ unit_M & map \\
= & unit_M & (6) \\
\\
\multicolumn{2}{l}{prod\ .\ map_N\ join} & \\
= & join\ .\ unit_M\ .\ map_N\ join & prod \\
= & join\ .\ map_M\ (map_N\ join)\ .\ unit_M & (3M) \\
= & join\ .\ map\ join\ .\ unit_M & map \\
= & join\ .\ join\ .\ unit_M & (7) \\
= & join\ .\ prod & prod
\end{array}$$

Figure 7: Proof of P(1)–P(4) from the existence of a composition

show that the *join* function we obtain from the *prod* construction is the same as the *join* function that we started with. One way to do this is as follows:

$$\begin{array}{rcll}
join_M\ .\ map_M\ prod & = & join_M\ .\ map_M\ (join\ .\ unit_M) & prod \\
& = & join_M\ .\ map_M\ join\ .\ map_M\ unit_M & (2M) \\
& = & join\ .\ join_M\ .\ map_M\ unit_M & J(1) \\
& = & join & (6M)
\end{array}$$

Note that this depends on an assumption about the way that *join* and $join_M$ 'commute' with one another:

$$join_M \;.\; map_M \; join \quad = \quad join \;.\; join_M \qquad \text{J(1)}$$

This condition may seem a little arbitrary, but it turns out that every composite monad obtained by the *prod* construction has this property:

$$\begin{array}{rcll}
join_M \;.\; map_M \; join & = & join_M \;.\; map_M \; (join_M \;.\; map_M \; prod) & join \\
 & = & join_M \;.\; map_M \; join_M \;.\; map_M \; (map_M \; prod) & \text{(2M)} \\
 & = & join_M \;.\; join_M \;.\; map_M \; (map_M \; prod) & \text{(7M)} \\
 & = & join_M \;.\; map_M \; prod \;.\; join_M & \text{(4M)} \\
 & = & join \;.\; join_M & join
\end{array}$$

It follows that the set of composite monads that can be obtained using the *prod* construction are precisely those satisfying J(1).

4.2 The *dorp* construction

As in the previous case, it is easy to find a suitably typed definition of the *dorp* function using the operations of the composite monad and its components:

$$dorp \quad = \quad join \;.\; map \; (map_M \; unit_N)$$

The proofs in Figure 8 show that this definition satisfies the laws D(1)–D(4) as we would hope. Once again, the most difficult task is to show that the *join* function obtained from this *dorp* function using the earlier construction is equal to the *join* operator in the composite monad. One way to prove this is as follows:

$$\begin{array}{lll}
\multicolumn{2}{l}{map_M \; join_N \;.\; dorp} & \\
 = & map_M \; join_N \;.\; join \;.\; map \; (map_M \; unit_N) & dorp \\
 = & join \;.\; map \; (map_M \; join_N) \;.\; map \; (map_M \; unit_N) & \text{J(2)} \\
 = & join \;.\; map \; (map_M \; (join_N \;.\; unit_N)) & \text{(2), (2M)} \\
 = & join \;.\; map \; (map_M \; id) & \text{(5N)} \\
 = & join & \text{(1M), (1)}
\end{array}$$

This also depends on an assumed law, this time linking the behaviour of *join* and $join_N$:

$$join \;.\; map \; (map_M \; join_N) \quad = \quad map_M \; join_N \;.\; join \qquad \text{J(2)}$$

In fact, law J(2) holds for all composite monads obtained using the *dorp* construction, as demonstrated by the following:

$$\begin{array}{lll}
\multicolumn{2}{l}{join \;.\; map \; (map_M \; join_N)} & \\
 = & map_M \; join_N \;.\; dorp \;.\; map \; (map_M \; join_N) & join \\
 = & map_M \; join_N \;.\; map \; join_N \;.\; dorp & \text{D(1)} \\
 = & map_M \; (join_N \;.\; map_N \; join_N) \;.\; dorp & map, \text{(2M)} \\
 = & map_M \; (join_N \;.\; join_N) \;.\; dorp & \text{(7N)} \\
 = & map_M \; join_N \;.\; map_M \; join_N \;.\; dorp & \text{(2M)} \\
 = & map_M \; join_N \;.\; join & join
\end{array}$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{dorp \;.\; map\ (map_M\ f)} & \\
\quad = & join \;.\; map\ (map_M\ unit_N) \;.\; map\ (map_M\ f) & dorp \\
\quad = & join \;.\; map\ (map_M\ (unit_N \;.\; f)) & \text{(2), (2M)} \\
\quad = & join \;.\; map\ (map_M\ (map_N\ f \;.\; unit_N)) & \text{(3N)} \\
\quad = & join \;.\; map\ (map\ f) \;.\; map\ (map_M\ unit_N) & \text{(2), (2M)}, map \\
\quad = & map\ f \;.\; join \;.\; map\ (map_M\ unit_N) & \text{(4)} \\
\quad = & map\ f \;.\; dorp & dorp \\
 & & \\
\multicolumn{2}{l}{dorp \;.\; unit} & \\
\quad = & join \;.\; map\ (map_M\ unit_N) \;.\; unit & dorp \\
\quad = & join \;.\; unit \;.\; map_M\ unit_N & \text{(3)} \\
\quad = & map_M\ unit_N & \text{(5)} \\
 & & \\
\multicolumn{2}{l}{dorp \;.\; map\ unit_M} & \\
\quad = & join \;.\; map\ (map_M\ unit_N) \;.\; map\ unit_M & dorp \\
\quad = & join \;.\; map\ (map_M\ unit_N \;.\; unit_M) & \text{(2)} \\
\quad = & join \;.\; map\ (unit_M \;.\; unit_N) & \text{(3M)} \\
\quad = & join \;.\; map\ unit & unit \\
\quad = & id & \text{(6)} \\
 & & \\
\multicolumn{2}{l}{dorp \;.\; join} & \\
\quad = & join \;.\; map\ (map_M\ unit_N) \;.\; join & dorp \\
\quad = & join \;.\; join \;.\; map\ (map\ (map_M\ unit_N)) & \text{(4)} \\
\quad = & join \;.\; map\ join \;.\; map\ (map\ (map_M\ unit_N)) & \text{(7)} \\
\quad = & join \;.\; map\ (join \;.\; map\ (map_M\ unit_N)) & \text{(2)} \\
\quad = & join \;.\; map\ dorp & dorp
\end{array}
$$

Figure 8: Proof of D(1)–D(4) from the existence of a composition

4.3 The *swap* construction

Our goal in this section is to determine the class of composite monads that can be constructed using the *swap* construction presented in Section 3.4. From the results given there, any composite obtained using *swap* can also be obtained from a *prod* or a *dorp* construction, so it follows (from the results in the last two sections) that any monad obtained from *swap* must satisfy at least J(1) and J(2). In fact, we will now show that these two properties are not only necessary, but also sufficient.

Following the pattern in the previous cases, we start with a definition for the *swap* function in terms of the *join* of the composite monad:

$$swap \;=\; join\ .\ unit_M\ .\ map_N\ (map_M\ unit_N)$$

or, equivalently:

$$swap \;=\; join\ .\ map\ (map_M\ unit_N)\ .\ unit_M$$

since:

$$\begin{array}{lll} & join\ .\ unit_M\ .\ map_N\ (map_M\ unit_N) & \\ = & join\ .\ map_M\ (map_N\ (map_M\ unit_N))\ .\ unit_M & \text{(3M)} \\ = & join\ .\ map\ (map_M\ unit_N)\ .\ unit_M. & map \end{array}$$

Note that these definitions of *swap* can also be expressed in terms of the *prod* and *dorp* functions used in the previous two sections:

$$\begin{array}{rcll} swap & = & join\ .\ unit_M\ .\ map_N\ (map_M\ unit_N) & swap \\ & = & prod\ .\ map_N\ (map_M\ unit_N) & prod \\ swap & = & join\ .\ map\ (map_M\ unit_N)\ .\ unit_M & swap \\ & = & dorp\ .\ unit_M & dorp \end{array}$$

Assuming J(2), we can show that the definition of *prod* in terms of *swap* given in Section 3.4 coincides with the *prod* function specified in terms of *join* in Section 4.1:

$$\begin{array}{lll} & map_M\ join_N\ .\ swap & \\ = & map_M\ join_N\ .\ join\ .\ unit_M\ .\ map_N\ (map_M\ unit_N) & swap \\ = & join\ .\ map\ (map_M\ join_N)\ .\ unit_M\ .\ map_N\ (map_M\ unit_N) & \text{J(2)} \\ = & join\ .\ unit_M\ .\ map_N\ (map_M\ join_N)\ .\ map_N\ (map_M\ unit_N) & map\text{, (3M)} \\ = & join\ .\ unit_M\ .\ map_N\ (map_M\ (join_N\ .\ unit_N)) & \text{(2N), (2M)} \\ = & join\ .\ unit_M & \text{(5N), (1M), (1N)} \end{array}$$

In a similar way, assuming J(1), we can show that the definition of *dorp* from *swap* in Section 3.4 gives the same function as the definition of *dorp* in Section 4.2:

$$\begin{array}{lll} & join_M\ .\ map_M\ swap & \\ = & join_M\ .\ map_M\ (join\ .\ map\ (map_M\ unit_N)\ .\ unit_M) & swap \\ = & join_M\ .\ map_M\ join\ .\ map_M\ (map\ (map_M\ unit_N)\ .\ unit_M) & \text{(2M)} \\ = & join\ .\ join_M\ .\ map_M\ (map\ (map_M\ unit_N)\ .\ unit_M) & \text{J(1)} \\ = & join\ .\ map\ (map_M\ unit_N)\ .\ join_M\ .\ map_M\ unit_M & map\text{, (4M), (2M)} \\ = & join\ .\ map\ (map_M\ unit_N) & \text{(6M)} \end{array}$$

With these results, it is straightforward to show that, given a composite monad satisfying J(1) and J(2), the *swap* function defined above satisfies the laws S(1)–S(4) required for the *swap* construction; see Figure 9 for the proofs. For convenience, we have used

$$
\begin{array}{lll}
\multicolumn{2}{l}{swap\ .\ map_N\ (map_M\ f)} & \\
\quad = & dorp\ .\ unit_M\ .\ map_N\ (map_M\ f) & swap \\
\quad = & dorp\ .\ map\ (map_M\ f)\ .\ unit_M & \text{(3M)},\ map \\
\quad = & map\ f\ .\ dorp\ .\ unit_M & \text{D(1)} \\
\quad = & map_M\ (map_N\ f)\ .\ swap & map,\ swap \\
\\
\multicolumn{2}{l}{swap\ .\ unit_N} & \\
\quad = & dorp\ .\ unit_M\ .\ unit_N & swap \\
\quad = & dorp\ .\ unit & unit \\
\quad = & map_M\ unit_N & \text{D(2)} \\
\\
\multicolumn{2}{l}{swap\ .\ map_N\ unit_M} & \\
\quad = & dorp\ .\ unit_M\ .\ map_N\ unit_M & swap \\
\quad = & dorp\ .\ map\ unit_M\ .\ unit_M & \text{(3M)},\ map \\
\quad = & unit_M & \text{D(3)} \\
\\
\multicolumn{2}{l}{dorp\ .\ prod} & \\
\quad = & dorp\ .\ join\ .\ unit_M & prod \\
\quad = & join\ .\ map\ dorp\ .\ unit_M & \text{D(4)} \\
\quad = & join\ .\ unit_M\ .\ map_N\ dorp & map,\ \text{(3M)} \\
\quad = & prod\ .\ map_N\ dorp & prod
\end{array}
$$

Figure 9: Proof of S(1)–S(4) from the existence of a composition

the properties of *dorp* described by the laws D(1)–D(4) to establish these properties of *swap*. Similar derivations are also possible using the laws for *prod*, or directly using the definition of *swap* in terms of *join* although the proofs are a little longer, particularly in the second case.

Given the comments above, this completes the proof that the set of monads which can be constructed from a *swap* function are precisely those satisfying both J(1) and J(2).

4.4 Summary

Corresponding to the diagram in Section 3.5, we can summarize the results about the converses for the *prod*, *dorp* and *swap* constructions as follows:

$$
\begin{tikzcd}
 & \boxed{\begin{array}{c} \mathit{prod} :: N\ (M\ (N\ a)) \to M\ (N\ a) \\ \text{satisfying P(1)–P(4)} \end{array}} \arrow[dl, "\text{J(2)}"'] & \\
\boxed{\begin{array}{c} \mathit{swap} :: N\ (M\ a) \to M\ (N\ a) \\ \text{satisfying S(1)–S(4)} \end{array}} & & \boxed{\begin{array}{c} \mathit{join} :: M\ (N\ (M\ (N\ a))) \to M\ (N\ a) \\ \text{satisfying (4)–(7)} \end{array}} \arrow[ll, "\text{J(1), J(2)}"'] \arrow[ul, "\text{J(1)}"'] \arrow[dl, "\text{J(2)}"] \\
 & \boxed{\begin{array}{c} \mathit{dorp} :: M\ (N\ (M\ a)) \to M\ (N\ a) \\ \text{satisfying D(1)–D(4)} \end{array}} \arrow[ul, "\text{J(1)}"] &
\end{tikzcd}
$$

This time, we have labeled the arrows between the different constructions with the additional properties required to establish the converse.

The properties J(1) and J(2) that we have used in these results are actually fairly similar, each given by a law of the form:

$$\mathit{join}\ .\ \mathit{map}\ f = f\ .\ \mathit{join}$$

For J(1), the *join* and *map* functions here should be read as the corresponding operators for the monad M with f the *join* function in the composite monad. For J(2), the *join* and *map* operators are taken from the composite monad with $f = \mathit{map}_M\ \mathit{join}_N$.

Incidentally, functions satisfying a law of the form shown above are actually quite widely studied in functional programming. For example, in the special case of the list monad where *join* is just the concatenation of a list of lists (the *concat* function in Haskell), functions satisfying the law above are often referred to as *list homomorphisms*.

5 Programming monad composition

Our goal in this and remaining sections is to show how the different constructions for monad composition presented above can be used in a practical programming language. For convenience, we will use the notation of *constructor classes* [5] implemented as part of the Gofer system, although the same ideas can also be applied to a much wider range of languages.

In this section, we present a general framework for working with different forms of monad composition. Later, we will describe a number of concrete examples of monad composition, and use some of these in a simple application.

5.1 Representing functors

The Haskell programming language [3] makes use of a system of *type classes* to provide a flexible treatment of ad-hoc polymorphism. A type class is a set of types, often referred to as the *instances* of the class, together with a family of operations that are defined for each instance. Constructor classes are a natural extension, allowing classes of type constructors. Since types are just nullary constructors (i.e. taking no arguments), this includes Haskell style type classes as a special case. For example, the following declaration introduces a class *Functor* and specifies that, for each instance f, there is a *map* function that maps functions of type $(a \to b)$ to functions of type $(f\ \ a \to f\ \ b)$:

$$\begin{array}{l} \textbf{class}\ \mathit{Functor}\ f\ \textbf{where} \\ \quad \mathit{map}\ \ ::\ \ (a \to b) \to (f\ \ a \to f\ \ b) \end{array}$$

This corresponds fairly closely to the definition of a functor in Section 2, except that it does not include the functor laws (1) and (2). Equality of functions is not computable so there is no way for a compiler to ensure that these laws are satisfied. However, it is useful to be able to include these as part of the program to check that they are, at the very least, type correct. One way to do this is to define an operator representing the desired equality:

$$\begin{array}{lcl} (===) & :: & a \to a \to \mathit{Law}\ \ a \\ x === y & = & \mathit{error}\ \text{"}\mathit{uncomputable\ \ equality}\text{"} \end{array}$$

The type signature here is particularly important, specifying that the two values being compared must be of the same type. The exact definition of the *Law a* type isn't important[1], but the argument type is used to record the type of the values that are asserted as being equal. Note however that any attempt to use this operator to compare two values will produce a runtime error.

Using this technique, we can represent the functor laws by the function definitions:

$$\begin{array}{lcl} \mathit{law}1 & :: & \mathit{Functor}\ f \Rightarrow () \to \mathit{Law}\ (f\ \ a \to f\ \ a) \\ \mathit{law}1\ () & = & \mathit{map}\ \mathit{id} === \mathit{id} \\ \\ \mathit{law}2 & :: & \mathit{Functor}\ f \Rightarrow (b \to c) \to (a \to b) \to \mathit{Law}\ (f\ \ a \to f\ \ c) \\ \mathit{law}2\ f\ \ g & = & \mathit{map}\ f\ \ .\ \mathit{map}\ g === \mathit{map}\ (f\ \ .\ g) \end{array}$$

Notice the use of function arguments to model free variables in the definition of *law2*. The types of these variables, as well as the type of the expressions being compared, can be read directly from the type signature. Note that, although we have written this explicitly as part of the program, the same type could also have been obtained automatically by the Gofer type inference mechanism.

In the first law we have used the unit value () to emphasize the fact that the law has no free variables[2].

[1] The definition we used was: **data** *Law* a = *Unspecified*.

[2] The addition of a dummy argument also helps to avoids the monomorphism restriction in Haskell/Gofer so that the type of *law1* can be calculated by the type inference mechanism. Without the dummy argument, an explicit type signature would be mandatory, not optional as it is here.

5.2 Representing premonads

With the framework established above, the representation of premonads is straightforward. The class of premonads can be described by the class:

$$\begin{array}{l}\textbf{class } \mathit{Functor}\ m \Rightarrow \mathit{Premonad}\ m \textbf{ where}\\ \quad \mathit{unit} \;\; :: \;\; a \rightarrow m\ a\end{array}$$

Note the first line in this declaration that captures the requirement that every premonad is also a functor; another way of saying this is that *Premonad* is a subclass of *Functor*. The premonad law, referred to as (3) in the first part of this report, can now be represented by:

$$\begin{array}{lcl}\mathit{law}3 & :: & \mathit{Premonad}\ m \Rightarrow (a \rightarrow b) \rightarrow \mathit{Law}\ (a \rightarrow m\ b)\\ \mathit{law}3\ f & = & \mathit{map}\ f\ .\ \mathit{unit} === \mathit{unit}\ .\ f\end{array}$$

5.3 Representing monads

The representation of monads also follows directly from our earlier definitions, captured by the class *Monad*, which is a subclass of *Premonad*:

$$\begin{array}{l}\textbf{class } \mathit{Premonad}\ m \Rightarrow \mathit{Monad}\ m \textbf{ where}\\ \quad \mathit{join} \;\; :: \;\; m\ (m\ a) \rightarrow m\ a\end{array}$$

The monad laws (4)–(7) are represented by the following:

$$\begin{array}{lcl}\mathit{law}4 & :: & \mathit{Monad}\ m \Rightarrow (a \rightarrow b) \rightarrow \mathit{Law}\ (m\ (m\ a) \rightarrow m\ b)\\ \mathit{law}4\ f & = & \mathit{join}\ .\ \mathit{map}\ (\mathit{map}\ f) === \mathit{map}\ f\ .\ \mathit{join}\\[1ex] \mathit{law}5 & :: & \mathit{Monad}\ m \Rightarrow () \rightarrow \mathit{Law}\ (m\ a \rightarrow m\ a)\\ \mathit{law}5\ () & = & \mathit{join}\ .\ \mathit{unit} === \mathit{id}\\[1ex] \mathit{law}6 & :: & \mathit{Monad}\ m \Rightarrow () \rightarrow \mathit{Law}\ (m\ a \rightarrow m\ a)\\ \mathit{law}6\ () & = & \mathit{join}\ .\ \mathit{map}\ \mathit{unit} === \mathit{id}\\[1ex] \mathit{law}7 & :: & \mathit{Monad}\ m \Rightarrow () \rightarrow \mathit{Law}\ (m\ (m\ (m\ a)) \rightarrow m\ a)\\ \mathit{law}7\ () & = & \mathit{join}\ .\ \mathit{map}\ \mathit{join} === \mathit{join}\ .\ \mathit{join}\end{array}$$

5.4 A general framework for composition constructions

It is easy to describe the composition of functors and premonads using the definitions:

$$\begin{array}{lcl}\mathit{mapC} & :: & (\mathit{Functor}\ f,\ \mathit{Functor}\ g) \Rightarrow (a \rightarrow b) \rightarrow (f\ (g\ a) \rightarrow f\ (g\ b))\\ \mathit{mapC} & = & \mathit{map}\ .\ \mathit{map}\\[1ex] \mathit{unitC} & :: & (\mathit{Premonad}\ m,\ \mathit{Premonad}\ n) \Rightarrow a \rightarrow m\ (n\ a)\\ \mathit{unitC} & = & \mathit{unit}\ .\ \mathit{unit}\end{array}$$

However, neither of these functions has a type of the right form to be able to define an instance of the *Functor* or *Premonad* classes. Furthermore, when we consider the different constructions for monad composition, there is nothing in a type expression of the form $f\ (g\ x)$ to indicate which construction is intended. To avoid this problem, we will define a different constructor c for each of the composition constructions with the intention that $c\ f\ g\ x$ is isomorphic to the composition $f\ (g\ x)$, identifying the construction used.

To simplify the task of converting between values of type $c\ f\ g\ x$ and those of type $f\ (g\ x)$, we introduce a constructor class to describe the required isomorphisms[3]:

$$\begin{array}{l}\textbf{class}\ \mathit{Composer}\ c\ \textbf{where}\\ \quad \begin{array}{lcl}\mathit{open} & :: & c\ f\ g\ x \to f\ (g\ x)\\ \mathit{close} & :: & f\ (g\ x) \to c\ f\ g\ x\end{array}\end{array}$$

Using these functions, we can package up the *mapC* operator defined above to give an instance of the *Functor* class:

$$\begin{array}{l}\textbf{instance}\ (\mathit{Composer}\ c,\ \mathit{Functor}\ f,\ \mathit{Functor}\ g) \Rightarrow \mathit{Functor}\ (c\ f\ g)\ \textbf{where}\\ \quad \mathit{map}\ f \ = \ \mathit{close}\ .\ \mathit{mapC}\ f\ .\ \mathit{open}\end{array}$$

Note that this definition can be used with any suitable c; we do not need to repeat the definition of the *map* function for each different construction.

The definition of the composition of premonads can also be dealt with in a similar manner.

$$\begin{array}{l}\textbf{instance}\ (\mathit{Composer}\ c,\ \mathit{Premonad}\ m,\ \mathit{Premonad}\ n) \Rightarrow \mathit{Premonad}\ (c\ m\ n)\ \textbf{where}\\ \quad \mathit{unit} \ = \ \mathit{close}\ .\ \mathit{unitC}\end{array}$$

In each of these instance declarations, we used compositions with *open* and *close* to modify the type of a value so that it could be used to define an instance of a particular class. The following function will be used a number of times in subsequent sections to *wrap* up the definition of a *join* function in an instance of the *Monad* class:

$$\begin{array}{lcl}\mathit{wrap} & :: & (\mathit{Composer}\ c,\ \mathit{Functor}\ m,\ \mathit{Functor}\ n) \Rightarrow\\ & & \quad (m\ (n\ (m\ (n\ a))) \to m\ (n\ a)) \to\\ & & \quad\quad (c\ m\ n\ (c\ m\ n\ a) \to c\ m\ n\ a)\\ \mathit{wrap}\ j & = & \mathit{close}\ .\ j\ .\ \mathit{mapC}\ \mathit{open}\ .\ \mathit{open}\end{array}$$

The type of this function may seem a little daunting. However, all it really does is convert a function with a type suitable for the *join* function of a composition to an equivalent form using a instance c of the *Composer* class.

[3] As another, technical aside, the implementations that we give in later sections are not strictly isomorphisms – the representations we use for each $c\ f\ g\ x$ are actually isomorphic to the lifted representation of $f\ (g\ x)$, not to $f\ (g\ x)$ itself. This could be fixed using strict constructors or irrefutable pattern matching; for the purposes of this work, we will assume that *open* and *close* are genuine isomorphisms and we will not consider the details any further here.

We will also need a way to embed computations in one component into the composite monad. This can be accomplished using the following functions:

$$\begin{array}{lcl} \mathit{right} & :: & (\mathit{Composer}\ c,\ \mathit{Premonad}\ f) \Rightarrow g\ a \to c\ f\ g\ a \\ \mathit{right} & = & \mathit{close}\ .\ \mathit{unit} \end{array}$$

$$\begin{array}{lcl} \mathit{left} & :: & (\mathit{Composer}\ c,\ \mathit{Functor}\ f,\ \mathit{Premonad}\ g) \Rightarrow f\ a \to c\ f\ g\ a \\ \mathit{left} & = & \mathit{close}\ .\ \mathit{map}\ \mathit{unit} \end{array}$$

See Section 7 for a concrete example of this.

5.5 Programming the *prod* construction

Our first construction, and our first application of the *Composer* class introduced above, is based on the *prod* construction first described in Section 3.2. We will use the following composer to identify this particular construction.

$$\textbf{data}\ \mathit{PComp}\ f\ g\ x \quad = \quad \mathit{PC}\ (f\ (g\ x))$$

$$\begin{array}{l} \textbf{instance}\ \mathit{Composer}\ \mathit{PComp}\ \textbf{where} \\ \quad \begin{array}{lcl} \mathit{open}\ (\mathit{PC}\ x) & = & x \\ \mathit{close} & = & \mathit{PC} \end{array} \end{array}$$

The construction itself requires a *prod* function, as described by the following class declaration:

$$\begin{array}{l} \textbf{class}\ (\mathit{Monad}\ m,\ \mathit{Premonad}\ n) \Rightarrow \mathit{PComposable}\ m\ n\ \textbf{where} \\ \quad \mathit{prod}\ \ ::\ \ n\ (m\ (n\ a)) \to m\ (n\ a) \end{array}$$

Note that *PComposable* uses two parameters and that the superclass constraints capture the requirement that m is a monad, while only a premonad structure is needed for n. The definition of the composite *join* function follows directly from our earlier results:

$$\begin{array}{lcl} \mathit{joinP} & :: & (\mathit{PComposable}\ m\ n) \Rightarrow m\ (n\ (m\ (n\ a))) \to m\ (n\ a) \\ \mathit{joinP} & = & \mathit{join}\ .\ \mathit{map}\ \mathit{prod} \end{array}$$

For good measure, we will also include the laws P(1)–P(4) that are required for the *prod* construction. For brevity, we omit the corresponding type signatures, all of which can in any case, be inferred automatically:

$$\begin{array}{lcl} p1\ f & = & \mathit{prod}\ .\ \mathit{map}\ (\mathit{mapC}\ f) === \mathit{mapC}\ f\ .\ \mathit{prod} \\ p2\ () & = & \mathit{prod}\ .\ \mathit{unit} === \mathit{id} \\ p3\ () & = & \mathit{prod}\ .\ \mathit{map}\ \mathit{unitC} === \mathit{unit} \\ p4\ () & = & \mathit{prod}\ .\ \mathit{map}\ \mathit{joinP} === \mathit{joinP}\ .\ \mathit{prod} \end{array}$$

Finally, we can package up the *joinP* function defined above to define a new instance of the *Monad* class (the corresponding superclass instances for *Functor* and *Premonad* are already covered by the definitions in the previous section):

$$\begin{array}{l} \textbf{instance}\ \mathit{PComposable}\ m\ n \Rightarrow \mathit{Monad}\ (\mathit{PComp}\ m\ n)\ \textbf{where} \\ \quad \mathit{join}\ \ =\ \ \mathit{wrap}\ \mathit{joinP} \end{array}$$

5.6 Programming the *dorp* construction

With the previous section still fresh in our minds, the treatment of the *dorp* construction in this section is unlikely to cause any big surprises. We begin with the definition of a new composer to identify compositions obtained from this construction:

$$\textbf{data}\ \mathit{DComp}\ f\ g\ x \ = \ \mathit{DC}\ (f\ (g\ x))$$

$$\begin{array}{lcl} \multicolumn{3}{l}{\textbf{instance}\ \mathit{Composer}\ \mathit{DComp}\ \textbf{where}} \\ \quad \mathit{open}\ (\mathit{DC}\ x) & = & x \\ \quad \mathit{close} & = & \mathit{DC} \end{array}$$

The construction requires a function *dorp*, specified by:

$$\begin{array}{lcl} \multicolumn{3}{l}{\textbf{class}\ (\mathit{Premonad}\ m,\ \mathit{Monad}\ n) \Rightarrow \mathit{DComposable}\ m\ n\ \textbf{where}} \\ \quad \mathit{dorp} & :: & m\ (n\ (m\ a)) \to m\ (n\ a) \end{array}$$

and yields a monad structure with a *join* function given by:

$$\begin{array}{lcl} \mathit{joinD} & :: & (\mathit{DComposable}\ m\ n) \Rightarrow m\ (n\ (m\ (n\ a))) \to m\ (n\ a) \\ \mathit{joinD} & = & \mathit{map}\ \mathit{join}\ .\ \mathit{dorp} \end{array}$$

Any definition of *dorp* is expected to satisfy the laws D(1)–D(4) which are represented as follows:

$$\begin{array}{lcl} \mathit{d1}\ f & = & \mathit{dorp}\ .\ \mathit{mapC}\ (\mathit{map}\ f) === \mathit{mapC}\ f\ .\ \mathit{dorp} \\ \mathit{d2}\ () & = & \mathit{dorp}\ .\ \mathit{unitC} === \mathit{map}\ \mathit{unit} \\ \mathit{d3}\ () & = & \mathit{dorp}\ .\ \mathit{mapC}\ \mathit{unit} === \mathit{id} \\ \mathit{d4}\ () & = & \mathit{dorp}\ .\ \mathit{joinD} === \mathit{joinD}\ .\ \mathit{mapC}\ \mathit{dorp} \end{array}$$

And we can package the *dorp* construction as an instance of the *Monad* class using the following declaration:

$$\begin{array}{lcl} \multicolumn{3}{l}{\textbf{instance}\ (\mathit{DComposable}\ m\ n) \Rightarrow \mathit{Monad}\ (\mathit{DComp}\ m\ n)\ \textbf{where}} \\ \quad \mathit{join} & = & \mathit{wrap}\ \mathit{joinD} \end{array}$$

5.7 Programming the *swap* construction

Following, once again, the pattern of the previous sections, we begin our implementation of the *swap* construction with the definition of a corresponding new composer:

$$\textbf{data}\ \mathit{SComp}\ f\ g\ x \ = \ \mathit{SC}\ (f\ (g\ x))$$

$$\begin{array}{lcl} \multicolumn{3}{l}{\textbf{instance}\ \mathit{Composer}\ \mathit{SComp}\ \textbf{where}} \\ \quad \mathit{open}\ (\mathit{SC}\ x) & = & x \\ \quad \mathit{close} & = & \mathit{SC} \end{array}$$

To compose two monads using this technique, we require a *swap* function given by:

$$\begin{array}{lcl} \multicolumn{3}{l}{\textbf{class}\ (\mathit{Monad}\ m,\ \mathit{Monad}\ n) \Rightarrow \mathit{SComposable}\ m\ n\ \textbf{where}} \\ \quad \mathit{swap} & :: & n\ (m\ a) \to m\ (n\ a) \end{array}$$

and satisfying the laws S(1)–S(4), represented by:

$$
\begin{array}{lcl}
s1\ f & = & swap\ .\ mapC\ f === mapC\ f\ .\ swap \\
s2\ () & = & swap\ .\ unit === map\ unit \\
s3\ () & = & swap\ .\ map\ unit === unit \\
s4\ () & = & prod\ .\ map\ dorp === dorp\ .\ prod \\
 & & \textbf{where}\ prod \;=\; map\ join\ .\ swap \\
 & & \qquad\quad dorp \;=\; join\ .\ map\ swap
\end{array}
$$

This allows us to define a monad structure on the composition with *join* function given by:

$$
\begin{array}{lcl}
joinS & :: & (SComposable\ m\ n) \Rightarrow m\ (n\ (m\ (n\ a))) \rightarrow m\ (n\ a) \\
joinS & = & map\ join\ .\ join\ .\ map\ swap
\end{array}
$$

which leads to the following instance of the *Monad* class:

$$
\begin{array}{l}
\textbf{instance}\ (SComposable\ m\ n) \Rightarrow Monad\ (SComp\ m\ n)\ \textbf{where} \\
\quad join \;=\; wrap\ joinS
\end{array}
$$

Finally, we can capture some aspects of the relationship between the different constructions using the following instance declarations.

$$
\begin{array}{l}
\textbf{instance}\ (SComposable\ m\ n) \Rightarrow PComposable\ m\ n\ \textbf{where} \\
\quad prod \;=\; map\ join\ .\ swap
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ (SComposable\ m\ n) \Rightarrow DComposable\ m\ n\ \textbf{where} \\
\quad dorp \;=\; join\ .\ map\ swap
\end{array}
$$

These definitions reflect the fact, proved in Section 3.4, that values for *prod* and *dorp* can always be derived from a suitable definition of *swap*.

6 Some specific monad constructions

Now that we have established the basic framework for our approach to monad composition, we will show how our results can be used to compose some specific monads. We have already seen that we cannot define a composition that works for two arbitrary monads. The next best option is to fix one of the components in a composition to be a particular monad and allow the other component to range over a family of different monads.

To illustrate the three different constructions, the following sections show how we can define certain compositions with the *Maybe*, reader, and list monads using the *prod*, *dorp* and *swap* constructions, respectively. We will also present some further examples using the same techniques to obtain compositions with other standard monads.

6.1 The *Maybe* datatype

The *Maybe* datatype, used in [12] to model a form of exception handling, is defined by:

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

There is a natural functor and monad structure corresponding to this datatype, given by the following declarations (proofs that these functions satisfy the appropriate laws are left as an exercise for the reader):

$$\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ \mathit{Maybe}\ \textbf{where} \\
\quad \begin{array}{lcl} \mathit{map}\ f\ (\mathit{Just}\ x) & = & \mathit{Just}\ (f\ x) \\ \mathit{map}\ f\ \mathit{Nothing} & = & \mathit{Nothing} \end{array}
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Premonad}\ \mathit{Maybe}\ \textbf{where} \\
\quad \mathit{unit} \ = \ \mathit{Just}
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ \mathit{Maybe}\ \textbf{where} \\
\quad \begin{array}{lcl} \mathit{join}\ (\mathit{Just}\ m) & = & m \\ \mathit{join}\ \mathit{Nothing} & = & \mathit{Nothing} \end{array}
\end{array}$$

Our goal now is to show how to construct a new monad by composing the *Maybe* constructor with an arbitrary datatype. Using the *prod* construction, a composition of the form $\mathit{Maybe}\ .\ n$ would require a function:

$$\mathit{prod} \ :: \ n\ (\mathit{Maybe}\ (n\ a)) \rightarrow \mathit{Maybe}\ (n\ a).$$

However, using only the monad operators for n, there does not appear to be any reasonable way to define a suitable function with this type. (This could probably be proved formally using the same kind of techniques as in the proof in the appendix.) On the other hand, for a composition of the form $m\ .\ \mathit{Maybe}$, we require a function:

$$\mathit{prod} \ :: \ \mathit{Maybe}\ (m\ (\mathit{Maybe}\ a)) \rightarrow m\ (\mathit{Maybe}\ a).$$

In this case, since we know something about the structure of objects constructed using *Maybe*, it is relatively easy to find a suitable definition for *prod*:

$$\begin{array}{l}
\textbf{instance}\ \mathit{PComposable}\ m\ \mathit{Maybe}\ \textbf{where} \\
\quad \begin{array}{lcl} \mathit{prod}\ (\mathit{Just}\ m) & = & m \\ \mathit{prod}\ \mathit{Nothing} & = & \mathit{unit}\ \mathit{Nothing} \end{array}
\end{array}$$

Proofs that this definition satisfies the necessary laws P(1)–P(4) are given in Figure 10. In each case, except for law P(2), we split the proof into two case – one for values of the form *Nothing* and a second for values *Just* m.

$$
\begin{array}{llll}
\textit{prod}\ (\textit{map}\ (\textit{mapC}\ f)\ \textit{Nothing}) & = & \textit{prod}\ \textit{Nothing} & \textit{map} \\
 & = & \textit{unit}\ \textit{Nothing} & \textit{prod} \\
 & = & \textit{unit}\ (\textit{map}\ f\ \textit{Nothing}) & \textit{map} \\
 & = & \textit{map}\ (\textit{map}\ f)\ (\textit{unit}\ \textit{Nothing}) & \textit{law3} \\
 & = & \textit{mapC}\ f\ (\textit{unit}\ \textit{Nothing}) & \textit{mapC} \\
 & = & \textit{mapC}\ f\ (\textit{prod}\ \textit{Nothing}) & \textit{prod} \\
\textit{prod}\ (\textit{map}\ (\textit{mapC}\ f)\ (\textit{Just}\ m)) & = & \textit{prod}\ (\textit{Just}\ (\textit{mapC}\ f\ m)) & \textit{map} \\
 & = & \textit{mapC}\ f\ m & \textit{prod} \\
 & = & \textit{mapC}\ f\ (\textit{prod}\ (\textit{Just}\ m)) & \textit{prod}
\end{array}
$$

$$
\begin{array}{llll}
\textit{prod}\ (\textit{unit}\ m) & = & \textit{prod}\ (\textit{Just}\ m) & \textit{unit} \\
 & = & m & \textit{prod}
\end{array}
$$

$$
\begin{array}{llll}
\textit{prod}\ (\textit{map}\ \textit{unitC}\ \textit{Nothing}) & = & \textit{prod}\ \textit{Nothing} & \textit{map} \\
 & = & \textit{unit}\ \textit{Nothing} & \textit{prod} \\
\textit{prod}\ (\textit{map}\ \textit{unitC}\ (\textit{Just}\ x)) & = & \textit{prod}\ (\textit{Just}\ (\textit{unitC}\ x)) & \textit{map} \\
 & = & \textit{unitC}\ x & \textit{prod} \\
 & = & \textit{unit}\ (\textit{unit}\ x) & \textit{unitC} \\
 & = & \textit{unit}\ (\textit{Just}\ x) & \textit{unit}
\end{array}
$$

$$
\begin{array}{llll}
\textit{prod}\ (\textit{map}\ \textit{joinP}\ \textit{Nothing}) & = & \textit{prod}\ \textit{Nothing} & \textit{map} \\
 & = & \textit{join}\ (\textit{unit}\ (\textit{prod}\ \textit{Nothing})) & \textit{law5} \\
 & = & \textit{join}\ (\textit{map}\ \textit{prod}\ (\textit{unit}\ \textit{Nothing})) & \textit{law3} \\
 & = & \textit{joinP}\ (\textit{unit}\ \textit{Nothing}) & \textit{joinP} \\
 & = & \textit{joinP}\ (\textit{prod}\ \textit{Nothing}) & \textit{prod} \\
\textit{prod}\ (\textit{map}\ \textit{joinP}\ (\textit{Just}\ m)) & = & \textit{prod}\ (\textit{Just}\ (\textit{joinP}\ m)) & \textit{map} \\
 & = & \textit{joinP}\ m & \textit{prod} \\
 & = & \textit{joinP}\ (\textit{prod}\ (\textit{Just}\ m)) & \textit{prod}
\end{array}
$$

Figure 10: Proof of P(1)–P(4) for the *Maybe* monad

6.2 Monad Comprehension Syntax

For most of the monad constructions introduced in the following sections, it is convenient to work with the notation of *monad comprehensions*. Several functional programming languages, including Haskell and Gofer, provide a special syntax for *list comprehensions* which allow some list-based computations to be expressed very clearly and concisely. Lists form a monad (see Section 6.4 for details) and, noticing this, Wadler [14] showed how the comprehension notation could be generalized to an arbitrary monad. A comprehension is written using the notation $[\, exp \mid gs \,]$ where exp is an expression and gs is a list of *generators* i.e. expressions of the form $x \leftarrow e$. The meaning of a comprehension can be defined by translating it into a form using the standard monad operators:

$$
\begin{array}{lcll}
[\, exp \mid x \leftarrow e \,] & = & map\ (\backslash x \rightarrow exp)\ e & \quad mapComp \\
[\, exp \,] & = & unit\ exp & \quad unitComp \\
[\, exp \mid gs,\ hs \,] & = & join\ [\, [\, exp \mid hs \,] \mid gs \,] & \quad joinComp
\end{array}
$$

The first equation can be considered as a way of defining map using the comprehension notation, with $map\ f\ [\, exp \mid gs \,] = [\, f\ exp \mid gs \,]$. The second equation gives the meaning of a comprehension with an empty sequence of qualifiers. In the last equation, the variables gs and hs range over (possibly empty) sequences of generators. Using the monad laws, we can show that the way in which these rules are used to find a translation of a monad comprehension does not have any effect on the meaning of the result. See [14] for further details.

For convenience, we will also use the following laws about monad comprehensions, each of which can be derived from the definitions above and the monad laws.

$$
\begin{array}{lcll}
[\, x \mid x \leftarrow xs \,] & = & xs & \quad compId \\
[\, f\ x \mid x \leftarrow map\ g\ e \,] & = & [\, f\ (g\ x) \mid x \leftarrow e \,] & \quad compMap \\
[\, f\ x \mid x \leftarrow unit\ e \,] & = & [\, f\ e \,] & \quad compUnit \\
[\, exp \mid x \leftarrow join\ e \,] & = & [\, exp \mid z \leftarrow e,\ x \leftarrow z \,] & \quad compJoin
\end{array}
$$

For example, the first of these is just another way of writing the functor law (1), while the second follows directly (2) and can be used to justify equalities such as:

$$
\begin{array}{lcl}
[\, h\ x\ y \mid x \leftarrow map\ f\ u,\ y \leftarrow map\ g\ v \,] & = & [\, h\ (f\ x)\ (g\ y) \mid x \leftarrow u,\ y \leftarrow v \,] \\
[\, exp \mid m \leftarrow map\ f\ n,\ x \leftarrow m \,] & = & [\, exp \mid m \leftarrow n,\ x \leftarrow f\ m \,]
\end{array}
$$

In a similar way, $compUnit$ can also be used in the justification of laws like:

$$[\, h\ x\ y \mid x \leftarrow unit\ u,\ y \leftarrow unit\ v \,] \;=\; [\, h\ u\ v \,].$$

Finally, we mention that the $compId$ and $compJoin$ can be used together to give a definition of the $join$ operator using the comprehension notation:

$$join\ e \;=\; [\, x \mid x \leftarrow join\ e \,] \;=\; [\, z \mid z \leftarrow e,\, x \leftarrow z \,].$$

6.3 Reader monads

A reader monad is described by a constructor of the form $(r \rightarrow)$ mapping each type a to the function type $r \rightarrow a$. We refer to computations in this monad as readers because they can read the value passed in the parameter of type r, but they cannot change that value. The functor, premonad and monad structures for readers are given by:

$$
\begin{array}{l}
\textbf{instance } \mathit{Functor}\ (r \rightarrow) \textbf{ where} \\
\quad \mathit{map}\ f\ g \;\; = \;\; f \,.\, g \\[1ex]
\textbf{instance } \mathit{Premonad}\ (r \rightarrow) \textbf{ where} \\
\quad \mathit{unit}\ x\ y \;\; = \;\; x \\[1ex]
\textbf{instance } \mathit{Monad}\ (r \rightarrow) \textbf{ where} \\
\quad \mathit{join}\ f\ x \;\; = \;\; f\ x\ x
\end{array}
$$

The following declaration can be used to compose an arbitrary monad n with a reader monad $(r \rightarrow)$ using the *dorp* construction:

$$
\begin{array}{l}
\textbf{instance } \mathit{DComposable}\ (r \rightarrow)\ n \textbf{ where} \\
\quad \mathit{dorp}\ m\ r \;\; = \;\; [\, g\ r \mid g \leftarrow m\ r \,]
\end{array}
$$

Proofs that this definition satisfies the laws D(1)–D(4) are included in Figure 11.

6.4 The *List* monad

Lists are one of the most widely used monads in functional programming. The structure of the list monad is captured by the type constructor *List*[4], together with the following instance declarations:

$$
\begin{array}{l}
\textbf{instance } \mathit{Functor}\ \mathit{List} \textbf{ where} \\
\quad \begin{array}{lcl} \mathit{map}\ f\ [\,] & = & [\,] \\ \mathit{map}\ f\ (x : \mathit{xs}) & = & f\ x\ :\ \mathit{map}\ f\ \mathit{xs} \end{array} \\[1ex]
\textbf{instance } \mathit{Premonad}\ \mathit{List} \textbf{ where} \\
\quad \mathit{unit}\ x \;\; = \;\; [\, x \,] \\[1ex]
\textbf{instance } \mathit{Monad}\ \mathit{List} \textbf{ where} \\
\quad \mathit{join} \;\; = \;\; \mathit{foldr}\ (+\!\!+)\ [\,]
\end{array}
$$

The *foldr* function and the list append operator $(+\!\!+)$ used here are both taken from the standard prelude.

This time, we will use the *swap* construction to obtain a composition of a monad m with the *List* monad (we will see shortly that this construction only yields a composite

[4] In fact, the concrete syntax of Gofer currently requires that we write this constructor as [ ], but we use the notation *List* here for clarity. The type *List* a is the same as the type $[a]$ in standard Haskell notation.

$$
\begin{array}{lcll}
\mathit{dorp}\ (\mathit{mapC}\ (\mathit{map}\ f)\ m)\ r & & & \\
 & = & [\,g\ r \mid g \leftarrow \mathit{map}\ (\mathit{map}\ (\mathit{map}\ f))\ m\ r\,] & \mathit{dorp},\ \mathit{mapC} \\
 & = & [\,g\ r \mid g \leftarrow \mathit{map}\ (\mathit{map}\ f)\ (m\ r)\,] & \mathit{map} \\
 & = & [\,\mathit{map}\ f\ g\ r \mid g \leftarrow m\ r\,] & \mathit{compMap} \\
 & = & [\,f\ (g\ r) \mid g \leftarrow m\ r\,] & \mathit{map} \\
 & = & \mathit{map}\ f\ [\,g\ r \mid g \leftarrow m\ r\,] & \mathit{compMap} \\
 & = & \mathit{map}\ f\ (\mathit{dorp}\ m\ r) & \mathit{dorp} \\
 & = & \mathit{map}\ (\mathit{map}\ f)\ (\mathit{dorp}\ m)\ r & \mathit{map} \\
 & = & \mathit{mapC}\ f\ (\mathit{dorp}\ m)\ r & \mathit{mapC} \\
 & & & \\
\mathit{dorp}\ (\mathit{unitC}\ f)\ r & = & [\,g\ r \mid g \leftarrow \mathit{unit}\ (\mathit{unit}\ f)\ r\,] & \mathit{dorp},\ \mathit{unitC} \\
 & = & [\,g\ r \mid g \leftarrow \mathit{unit}\ f\,] & \mathit{unit} \\
 & = & [\,f\ r\,] & \mathit{compUnit} \\
 & = & \mathit{unit}\ (f\ r) & \mathit{unitComp} \\
 & = & \mathit{map}\ \mathit{unit}\ f\ r & \mathit{map} \\
 & & & \\
\mathit{dorp}\ (\mathit{mapC}\ \mathit{unit}\ f)\ r & = & [\,g\ r \mid g \leftarrow \mathit{map}\ (\mathit{map}\ \mathit{unit})\ f\ r\,] & \mathit{dorp}, \mathit{mapC} \\
 & = & [\,g\ r \mid g \leftarrow \mathit{map}\ \mathit{unit}\ (f\ r)\,] & \mathit{map} \\
 & = & [\,\mathit{unit}\ g\ r \mid g \leftarrow f\ r\,] & \mathit{compMap} \\
 & = & [\,g \mid g \leftarrow f\ r\,] & \mathit{unit} \\
 & = & f\ r & \mathit{compId} \\
 & & & \\
\mathit{dorp}\ (\mathit{joinD}\ f)\ r & = & [\,g\ r \mid g \leftarrow \mathit{map}\ \mathit{join}\ (\mathit{dorp}\ f)\ r\,] & \mathit{dorp},\ \mathit{joinD} \\
 & = & [\,g\ r \mid g \leftarrow \mathit{join}\ (\mathit{dorp}\ f\ r)\,] & \mathit{map} \\
 & = & [\,g\ r \mid h \leftarrow \mathit{dorp}\ f\ r,\ g \leftarrow h\,] & \mathit{compJoin} \\
 & = & [\,g\ r \mid h \leftarrow [\,k\ r \mid k \leftarrow f\ r\,],\ g \leftarrow h\,] & \mathrm{dorp} \\
 & = & [\,g\ r \mid h \leftarrow f\ r,\ g \leftarrow h\ r\,] & \mathit{mapComp} \\
 & = & \mathit{join}\ [\,[\,g\ r \mid g \leftarrow h\ r\,] \mid h \leftarrow f\ r\,] & \mathit{joinComp} \\
 & = & \mathit{join}\ [\,\mathit{dorp}\ h\ r \mid h \leftarrow f\ r\,] & \mathit{dorp} \\
 & = & \mathit{join}\ [\,h\ r \mid h \leftarrow \mathit{map}\ \mathit{dorp}\ (f\ r)\,] & \mathit{compMap} \\
 & = & \mathit{join}\ [\,h\ r \mid h \leftarrow \mathit{map}\ (\mathit{map}\ \mathit{dorp})\ f\ r\,] & \mathit{map} \\
 & = & \mathit{join}\ [\,h\ r \mid h \leftarrow \mathit{mapC}\ \mathit{dorp}\ f\ r\,] & \mathit{mapC} \\
 & = & \mathit{join}\ (\mathit{dorp}\ (\mathit{mapC}\ \mathit{dorp}\ f)\ r) & \mathit{dorp} \\
 & = & \mathit{map}\ \mathit{join}\ (\mathit{dorp}\ (\mathit{mapC}\ \mathit{dorp}\ f))\ r & \mathit{map} \\
 & = & \mathit{joinD}\ (\mathit{mapC}\ \mathit{dorp}\ f)\ r & \mathit{joinD}
\end{array}
$$

Figure 11: Proof of D(1)–D(4) for reader monads

monad if m has a certain commutativity property):

$$\begin{array}{l} \textbf{instance } \mathit{SComposable}\ m\ \mathit{List}\ \textbf{where} \\ \quad \begin{array}{lcl} \mathit{swap}\ [\,] & = & \mathit{unit}\ [\,] \\ \mathit{swap}\ (x : \mathit{xs}) & = & [\, y : \mathit{ys} \mid y \leftarrow x,\ \mathit{ys} \leftarrow \mathit{swap}\ \mathit{xs}\,] \end{array} \end{array}$$

The proofs for S(1)–S(3) in Figure 12 hold for any monad m. Structural induction is required for the proofs of S(1) and S(3); we have not shown the case for $\perp$ values which follow directly from the use of pattern matching in the definition of the monad operators.

The proof of law S(4) is rather more involved. To begin with, we need to introduce the auxiliary functions:

$$\begin{array}{lcl} \mathit{prod} & = & \mathit{map}\ \mathit{join}\ .\ \mathit{swap} \\ \mathit{dorp} & = & \mathit{join}\ .\ \mathit{map}\ \mathit{swap} \end{array}$$

A quick calculation (or, if you prefer, proof by induction) based on the definition of *swap* allows us to use the following definition for the *prod* function:

$$\begin{array}{lcl} \mathit{prod}\ [\,] & = & \mathit{unit}\ [\,] \\ \mathit{prod}\ (x : \mathit{xs}) & = & [\, u +\!\!+ v \mid u \leftarrow x,\ v \leftarrow \mathit{prod}\ \mathit{xs}\,] \end{array}$$

The proof of S(4), by structural induction, is presented in Figure 13.

There are two steps in the proof which require further explanation. The first is a lemma describing the way that *swap* distributes over $(+\!\!+)$:

$$\mathit{swap}\ (\mathit{xs} +\!\!+ \mathit{ys}) \quad = \quad [\, x +\!\!+ y \mid x \leftarrow \mathit{swap}\ \mathit{xs},\ y \leftarrow \mathit{swap}\ \mathit{ys}\,]$$

This law holds for any monad m (i.e. the monad in which the comprehension is interpreted) and can be proved using the simple structural induction on xs in Figure 14.

Notice that we made use of (yet another) variant of *compMap* :

$$\begin{array}{l} [\, f\ x\ y \mid x \leftarrow u,\ t \leftarrow [\, h\ y\ z \mid y \leftarrow v,\ z \leftarrow w\,]\,] \\ \quad = \quad [\, f\ x\ (h\ y\ z) \mid x \leftarrow u,\ y \leftarrow v,\ z \leftarrow w\,] \end{array}$$

The other important step is the use of a commutativity property that enables us to swap two of the generators in a comprehension. We require that the following law is satisfied :

$$\begin{array}{l} [\, r +\!\!+ s \mid v \leftarrow x,\ r \leftarrow \mathit{swap}\ v,\ u \leftarrow \mathit{prod}\ \mathit{xs},\ s \leftarrow \mathit{swap}\ u\,] \\ \quad = \quad [\, r +\!\!+ s \mid v \leftarrow x,\ u \leftarrow \mathit{prod}\ \mathit{xs},\ r \leftarrow \mathit{swap}\ v,\ s \leftarrow \mathit{swap}\ u\,] \end{array}$$

One way to ensure that this law holds is to insist that m is a commutative monad – i.e. that it satisfies the law:

$$[\,(x, y) \mid x \leftarrow u,\ y \leftarrow v\,] \quad = \quad [\,(x, y) \mid y \leftarrow v,\ x \leftarrow u\,].$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathit{swap}\ (\mathit{mapC}\ f\ [\,])} & \\
\quad = & \mathit{swap}\ (\mathit{map}\ (\mathit{map}\ f)\ [\,]) & \mathit{mapC} \\
\quad = & \mathit{swap}\ [\,] & \mathit{mapC},\ \mathit{map} \\
\quad = & \mathit{unit}\ [\,] & \mathit{swap} \\
\quad = & \mathit{unit}\ (\mathit{map}\ f\ [\,]) & \mathit{map} \\
\quad = & \mathit{map}\ (\mathit{map}\ f)\ (\mathit{unit}\ [\,]) & \mathit{law3} \\
\quad = & \mathit{mapC}\ f\ (\mathit{swap}\ [\,]) & \mathit{mapC},\ \mathit{swap}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathit{swap}\ (\mathit{mapC}\ f\ (x : \mathit{xs}))} & \\
\quad = & \mathit{swap}\ (\mathit{map}\ (\mathit{map}\ f)\ (x : \mathit{xs})) & \mathit{mapC} \\
\quad = & \mathit{swap}\ (\mathit{map}\ f\ x\ :\ \mathit{map}\ (\mathit{map}\ f)\ \mathit{xs}) & \mathit{map} \\
\quad = & [\, y : \mathit{ys} \mid y \leftarrow \mathit{map}\ f\ x,\ \mathit{ys} \leftarrow \mathit{swap}\ (\mathit{map}\ (\mathit{map}\ f)\ \mathit{xs})\,] & \mathit{swap} \\
\quad = & [\, y : \mathit{ys} \mid y \leftarrow \mathit{map}\ f\ x,\ \mathit{ys} \leftarrow \mathit{map}\ (\mathit{map}\ f)\ (\mathit{swap}\ \mathit{xs})\,] & \text{induction} \\
\quad = & [\, f\ y\ :\ \mathit{map}\ f\ \mathit{ys} \mid y \leftarrow x,\ \mathit{ys} \leftarrow \mathit{swap}\ \mathit{xs}\,] & \mathit{compMap} \\
\quad = & [\, \mathit{map}\ f\ (y : \mathit{ys}) \mid y \leftarrow x,\ \mathit{ys} \leftarrow \mathit{swap}\ \mathit{xs}\,] & \mathit{compMap} \\
\quad = & \mathit{map}\ (\mathit{map}\ f)\ [\, y : \mathit{ys} \mid y \leftarrow x, \mathit{ys} \leftarrow \mathit{swap}\ \mathit{xs}\,] & \mathit{map} \\
\quad = & \mathit{mapC}\ f\ (\mathit{swap}\ (x : \mathit{xs})) & \mathit{mapC},\ \mathit{swap}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathit{swap}\ (\mathit{unit}\ x)} & \\
\quad = & \mathit{swap}\ [x] & \mathit{unit} \\
\quad = & [\, y : \mathit{ys} \mid y \leftarrow x,\ \mathit{ys} \leftarrow \mathit{swap}\ [\,]\,] & \mathit{swap} \\
\quad = & [\, y : \mathit{ys} \mid y \leftarrow x,\ \mathit{ys} \leftarrow \mathit{unit}\ [\,]\,] & \mathit{swap} \\
\quad = & [\, [y] \mid y \leftarrow x\,] & \mathit{compUnit} \\
\quad = & [\, \mathit{unit}\ y \mid y \leftarrow x\,] & \mathit{unitComp} \\
\quad = & \mathit{map}\ \mathit{unit}\ x & \mathit{mapComp}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathit{swap}\ (\mathit{map}\ \mathit{unit}\ [\,])} & \\
\quad = & \mathit{swap}\ [\,] & \mathit{map} \\
\quad = & \mathit{unit}\ [\,] & \mathit{swap}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathit{swap}\ (\mathit{map}\ \mathit{unit}\ (x : \mathit{xs}))} & \\
\quad = & \mathit{swap}\ (\mathit{unit}\ x\ :\ \mathit{map}\ \mathit{unit}\ \mathit{xs}) & \mathit{map} \\
\quad = & [\, y : \mathit{ys} \mid y \leftarrow \mathit{unit}\ x,\ \mathit{ys} \leftarrow \mathit{swap}\ (\mathit{map}\ \mathit{unit}\ \mathit{xs})\,] & \mathit{swap} \\
\quad = & [\, y : \mathit{ys} \mid y \leftarrow \mathit{unit}\ x,\ \mathit{ys} \leftarrow \mathit{unit}\ \mathit{xs}\,] & \text{induction} \\
\quad = & [\, x : \mathit{xs}\,] & \mathit{compUnit} \\
\quad = & \mathit{unit}\ (x : \mathit{xs}) & \mathit{unitComp}
\end{array}
$$

Figure 12: Proof of S(1)–S(3) for the list monad

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathit{prod}\ (\mathit{map}\ \mathit{dorp}\ [\,])} & \\
\quad = & \mathit{prod}\ [\,] & \mathit{map} \\
\quad = & \mathit{unit}\ [\,] & \mathit{prod} \\
\quad = & \mathit{swap}\ [\,] & \mathit{swap} \\
\quad = & \mathit{join}\ (\mathit{unit}\ (\mathit{swap}\ [\,])) & \mathit{law5} \\
\quad = & \mathit{join}\ (\mathit{map}\ \mathit{swap}\ (\mathit{unit}\ [\,])) & \mathit{law3} \\
\quad = & \mathit{dorp}\ (\mathit{unit}\ [\,]) & \mathit{dorp} \\
\quad = & \mathit{dorp}\ (\mathit{prod}\ [\,]) & \mathit{prod}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathit{prod}\ (\mathit{map}\ \mathit{dorp}\ (x : \mathit{xs}))} & \\
\quad = & \mathit{prod}\ (\mathit{dorp}\ x\ :\ \mathit{map}\ \mathit{dorp}\ \mathit{xs}) & \mathit{map} \\
\quad = & [\, r \mathbin{+\!\!+} s \mid r \leftarrow \mathit{dorp}\ x,\ s \leftarrow \mathit{prod}\ (\mathit{map}\ \mathit{dorp}\ \mathit{xs})\,] & \mathit{prod} \\
\quad = & [\, r \mathbin{+\!\!+} s \mid r \leftarrow \mathit{dorp}\ x,\ s \leftarrow \mathit{dorp}\ (\mathit{prod}\ \mathit{xs})\,] & \text{induction} \\
\quad = & [\, r \mathbin{+\!\!+} s \mid r \leftarrow \mathit{dorp}\ x,\ s \leftarrow \mathit{join}\ (\mathit{map}\ \mathit{swap}\ (\mathit{prod}\ \mathit{xs}))\,] & \mathit{dorp} \\
\quad = & [\, r \mathbin{+\!\!+} s \mid r \leftarrow \mathit{dorp}\ x,\ u \leftarrow \mathit{map}\ \mathit{swap}\ (\mathit{prod}\ \mathit{xs}),\ s \leftarrow u\,] & \mathit{compJoin} \\
\quad = & [\, r \mathbin{+\!\!+} s \mid r \leftarrow \mathit{dorp}\ x,\ u \leftarrow \mathit{prod}\ \mathit{xs},\ s \leftarrow \mathit{swap}\ u\,] & \mathit{compMap} \\
\quad = & [\, r \mathbin{+\!\!+} s \mid r \leftarrow \mathit{join}\ (\mathit{map}\ \mathit{swap}\ x),\ u \leftarrow \mathit{prod}\ \mathit{xs},\ s \leftarrow \mathit{swap}\ u\,] & \mathit{dorp} \\
\quad = & [\, r \mathbin{+\!\!+} s \mid v \leftarrow \mathit{map}\ \mathit{swap}\ x,\ r \leftarrow v,\ u \leftarrow \mathit{prod}\ \mathit{xs},\ s \leftarrow \mathit{swap}\ u\,] & \mathit{compJoin} \\
\quad = & [\, r \mathbin{+\!\!+} s \mid v \leftarrow x,\ r \leftarrow \mathit{swap}\ v,\ u \leftarrow \mathit{prod}\ \mathit{xs},\ s \leftarrow \mathit{swap}\ u\,] & \mathit{compMap} \\
\quad = & [\, r \mathbin{+\!\!+} s \mid v \leftarrow x,\ u \leftarrow \mathit{prod}\ \mathit{xs},\ r \leftarrow \mathit{swap}\ v,\ s \leftarrow \mathit{swap}\ u\,] & \text{commute} \\
\quad = & \mathit{join}\ [\,[\, r \mathbin{+\!\!+} s \mid r \leftarrow \mathit{swap}\ v,\ s \leftarrow \mathit{swap}\ u\,] & \mathit{joinComp} \\
 & \qquad\qquad \mid v \leftarrow x,\ u \leftarrow \mathit{prod}\ \mathit{xs}\,] & \\
\quad = & \mathit{join}\ [\, \mathit{swap}\ (v \mathbin{+\!\!+} u) \mid v \leftarrow x,\ u \leftarrow \mathit{prod}\ \mathit{xs}\,] & \mathit{lemma} \\
\quad = & \mathit{join}\ (\mathit{map}\ \mathit{swap}\ [\, v \mathbin{+\!\!+} u \mid v \leftarrow x,\ u \leftarrow \mathit{prod}\ \mathit{xs}\,]) & \mathit{compMap} \\
\quad = & \mathit{dorp}\ [\, v \mathbin{+\!\!+} u \mid v \leftarrow x,\ u \leftarrow \mathit{prod}\ \mathit{xs}\,] & \mathit{dorp} \\
\quad = & \mathit{dorp}\ (\mathit{prod}\ (x : \mathit{xs})) & \mathit{prod}
\end{array}
$$

Figure 13: Proof of S(4) for the list monad

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathit{swap}\ ([\,]\mathbin{+\!\!+}\mathit{ys})} & \\
\quad = & \mathit{swap}\ \mathit{ys} & (\mathbin{+\!\!+}) \\
\quad = & [\,\mathit{vs} \mid \mathit{vs} \leftarrow \mathit{swap}\ \mathit{ys}\,] & \mathit{compId} \\
\quad = & [\,[\,]\mathbin{+\!\!+}\mathit{vs} \mid \mathit{vs} \leftarrow \mathit{swap}\ \mathit{ys}\,] & (\mathbin{+\!\!+}) \\
\quad = & [\,\mathit{us}\mathbin{+\!\!+}\mathit{vs} \mid \mathit{us} \leftarrow \mathit{unit}\ [\,],\ \mathit{vs} \leftarrow \mathit{swap}\ \mathit{ys}\,] & \mathit{compUnit} \\
\quad = & [\,\mathit{us}\mathbin{+\!\!+}\mathit{vs} \mid \mathit{us} \leftarrow \mathit{swap}\ [\,],\ \mathit{vs} \leftarrow \mathit{swap}\ \mathit{ys}\,] & \mathit{swap} \\
\multicolumn{2}{l}{\mathit{swap}\ ((x : \mathit{xs})\mathbin{+\!\!+}\mathit{ys})} & \\
\quad = & \mathit{swap}\ (x : \mathit{xs}\mathbin{+\!\!+}\mathit{ys}) & (\mathbin{+\!\!+}) \\
\quad = & [\,z : \mathit{zs} \mid z \leftarrow x,\ \mathit{zs} \leftarrow \mathit{swap}\ (\mathit{xs}\mathbin{+\!\!+}\mathit{ys})\,] & \mathit{swap} \\
\quad = & [\,z : \mathit{zs} \mid z \leftarrow x,\ \mathit{zs} \leftarrow [\,\mathit{us}\mathbin{+\!\!+}\mathit{vs} \mid \mathit{us} \leftarrow \mathit{swap}\ \mathit{xs},\ \mathit{vs} \leftarrow \mathit{swap}\ \mathit{ys}\,]\,] & \text{induction} \\
\quad = & [\,z : \mathit{us}\mathbin{+\!\!+}\mathit{vs} \mid z \leftarrow x,\ \mathit{us} \leftarrow \mathit{swap}\ \mathit{xs},\ \mathit{vs} \leftarrow \mathit{swap}\ \mathit{ys}\,] & \mathit{compMap} \\
\quad = & [\,\mathit{us}\mathbin{+\!\!+}\mathit{vs} \mid \mathit{us} \leftarrow \mathit{swap}\ (x : \mathit{xs}),\ \mathit{vs} \leftarrow \mathit{swap}\ \mathit{ys}\,] & \mathit{swap}
\end{array}
$$

Figure 14: Proof of $\mathit{swap}\ (\mathit{xs}\mathbin{+\!\!+}\mathit{ys}) = [\,x\mathbin{+\!\!+}y \mid x \leftarrow \mathit{swap}\ \mathit{xs},\ y \leftarrow \mathit{swap}\ \mathit{ys}\,]$.

The identity monad, the set monad and reader monads all have this property. On the other hand, the list monad, *Maybe* monad and state monad are non-commutative.

In summary, if m is a commutative monad, then $\mathit{SComp}\ m\ \mathit{List}$ is a monad. Since we require only a single special case of the commutativity property, it is possible that we may be able to relax the restriction to commutative monads to some degree. However, it is not possible to remove the restrictions altogether. In particular, our constructions cannot be used to compose the *List* monad with itself.

The importance of commutative monads has also been recognized in other situations. For example, in [6], monads satisfying the commutativity axiom are used to capture explicit parallel execution of programs written in a monadic style.

6.4.1 Composing a monad with itself

In Section 2 we commented that, just by looking at the types involved, it was clear that a definition $\mathit{join} = \mathit{join}_M\ .\ \mathit{join}_N$ could not be used to form a composition of two monads M and N. The argument was that, while the function join_N produces a result with type of the form $N\ a$, the function join_M expects a value with a type of the form $M\ (M\ a)$. In the general case, these types do not match. However, as the reader may have realized, this argument fails in the special case where $M = N$.

In fact, although it has the required type, we still cannot use the function $\mathit{join}_M\ .\ \mathit{join}_M$, because it does not satisfy the monad laws. For example, using the list monad, we can demonstrate a direct counter example to law (6):

$$(\mathit{join}\ .\ \mathit{join}\ .\ \mathit{map}\ \mathit{unitC})\ [[\,]] \ = \ [\,] \ \neq \ [[\,]] \ = \ \mathit{id}\ [[\,]]$$

So, in general, the composition of a monad with itself cannot be treated as a special case; we still need to use one of the constructions described above.

6.4.2 Comparison with 'Combining monads'

Our results restrict composition with *List* to commutative monads. In contrast, King and Wadler [8] give a slightly different construction which seems more general, avoiding any restrictions on the choice of monads that can be composed with *List*. Unfortunately, this promise of a more general construction turns out to be something of a mirage; although developed in a rather different manner, their approach turns out to be equivalent to our *prod* construction and does, in fact, require some form of commutativity.

The construction given in [8], is based on the definitions:

$$
\begin{array}{lcl}
(\otimes) & :: & \mathit{Monad}\ m \Rightarrow m\ [a] \to m\ [a] \to m\ [a] \\
a \otimes b & = & [\, x +\!\!+\, y \mid x \leftarrow a,\ y \leftarrow b\,]
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{prod} & :: & \mathit{Monad}\ m \Rightarrow [m\ [a]] \to m\ [a] \\
\mathit{prod} & = & \mathit{foldr}\ (\otimes)\ (\mathit{unit}\ [\,])
\end{array}
$$

This *prod* function is used to define $\mathit{join} = \mathit{join}\ .\ \mathit{map}\ \mathit{prod}$, the proof that this defines a monad structure for m composed with the list monad resting on a number of properties of *prod*, including:

$$\mathit{prod}\ .\ \mathit{map}\ (\mathit{join}\ .\ \mathit{map}\ \mathit{prod}) \quad = \quad \mathit{join}\ .\ \mathit{map}\ \mathit{prod}\ .\ \mathit{prod}.$$

However, using the following, somewhat contrived counter example, we find that this law does not always hold:

```
? (prod . map (join . map prod)) [[[[[1],[3]]]],[[[[4]]],[[[2]]]]]
[[1, 4], [1, 2], [3, 4], [3, 2]]
? (join . map prod . prod) [[[[[1],[3]]]],[[[[4]]],[[[2]]]]]
[[1, 4], [3, 4], [1, 2], [3, 2]]
```

Of course, none of what we have said here proves that it is impossible to construct monads by composition of arbitrary monads with the list monad; all we know is that it is not possible using the constructions described in this report.

6.5 Some additional monad compositions

The following sections deal with some further examples, providing compositions with standard monads. In each case, we use the *swap* construction to obtain the composition, but detailed proofs are not included since they are very similar to those in previous sections[5].

[5]Detailed proofs of results in this section *are* included, as program comments, in a Gofer script containing an executable version of the definitions in this report. See comments on the Page 1 for details of availability.

6.5.1 Writer monads

A writer monad can be used for describing programs that produce both output and a return value. It pays to take a general approach, allowing the type of values used as output to be provided as a parameter to the monad constructor, rather than committing ourselves to a particular output type at an early stage. The following declarations define a constructor *Writer* and, assuming an output type s, a corresponding functor and monad structure for each *Writer* s constructor:

$$\textbf{data}\ \mathit{Writer}\ s\ a \quad = \quad \mathit{Result}\ s\ a$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Functor}\ (\mathit{Writer}\ s)\ \textbf{where} \\ \quad \mathit{map}\ f\ (\mathit{Result}\ s\ a) = \mathit{Result}\ s\ (f\ a)\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Monoid}\ s \Rightarrow \mathit{Premonad}\ (\mathit{Writer}\ s)\ \textbf{where} \\ \quad \mathit{unit} \quad = \quad \mathit{Result}\ \mathit{zero}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Monoid}\ s \Rightarrow \mathit{Monad}\ (\mathit{Writer}\ s)\ \textbf{where} \\ \quad \mathit{join}\ (\mathit{Result}\ s\ (\mathit{Result}\ t\ x)) \quad = \quad \mathit{Result}\ (\mathit{add}\ s\ t)\ x\end{array}$$

$$\begin{array}{lcl}\mathit{write} & :: & s \rightarrow \mathit{Writer}\ s\ () \\ \mathit{write}\ \mathit{msg} & = & \mathit{Result}\ \mathit{msg}\ ()\end{array}$$

The *write* function defined here is used to perform output; the argument *msg* is returned as the output and the value returned is just (), the unit value.

The values *zero* and *add* in the definitions above represent the null output, and the sequencing of one output after another. To establish the monad laws for *Writer* s constructors, we need to insist that *add* is associative with *zero* as both a left and right identity. In other words, we require that these values form a monoid. This can be captured by the following class declaration:

$$\begin{array}{l}\textbf{class}\ \mathit{Monoid}\ s\ \textbf{where} \\ \quad \begin{array}{lcl}\mathit{zero} & :: & s \\ \mathit{add} & :: & s \rightarrow s \rightarrow s\end{array}\end{array}$$

Two obvious choices for output types, each of which forms a monad, are lists and functions, as described by the following instance declarations.

$$\begin{array}{l}\textbf{instance}\ \mathit{Monoid}\ [a]\ \textbf{where} \\ \quad \begin{array}{lcl}\mathit{zero} & = & [\,] \\ \mathit{add} & = & (+\!\!+)\end{array}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Monoid}\ (a \rightarrow a)\ \textbf{where} \\ \quad \begin{array}{lcl}\mathit{zero} & = & \mathit{id} \\ \mathit{add} & = & (.)\end{array}\end{array}$$

For the example in Section 7, we will use lists of strings (i.e. values of type *String*, each corresponding to a line of output) for output values. In practice, an output type of the form $String \rightarrow String$ might be more sensible, allowing tree like structures to be printed in linear time rather than in time quadratic in the size of the tree. Both of these possibilities are permitted by the definitions above.

Incidentally, we will also mention that the monad *Writer Int*, assuming the monoid structure:

$$\begin{array}{l}\textbf{instance}\ \mathit{Monoid\ Int}\ \textbf{where}\\ \quad \begin{array}{lcl} \mathit{zero} & = & 0\\ \mathit{add} & = & (+)\end{array}\end{array}$$

gives us another example of a commutative monad (see Section 6.4). This might be used, for example, in a simple profiler, counting the number of times that particular tasks are carried out and using the command *write 1* to increment the counter at suitable points in the program.

Writer monads can be composed with arbitrary monads using the following definition of *swap*:

$$\begin{array}{l}\textbf{instance}\ \mathit{Monoid}\ s \Rightarrow \mathit{SComposable}\ m\ (\mathit{Writer}\ s)\ \textbf{where}\\ \quad \mathit{swap}\ (\mathit{Result}\ s\ m) \ = \ [\,\mathit{Result}\ s\ a \mid a \leftarrow m\,]\end{array}$$

6.5.2 The *Error* monad

As a simple variation on the *Maybe* monad described in Section 6.1, it is often useful to be able to return some form of error message when an exception occurs. This can be described by the *Error* monad:

$$\textbf{data}\ \mathit{Error}\ a \ = \ \mathit{Ok}\ a \mid \mathit{Error\ String}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Functor\ Error}\ \textbf{where}\\ \quad \begin{array}{lcl} \mathit{map}\ f\ (\mathit{Ok}\ x) & = & \mathit{Ok}\ (f\ x)\\ \mathit{map}\ f\ (\mathit{Error\ msg}) & = & \mathit{Error\ msg}\end{array}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Premonad\ Error}\ \textbf{where}\\ \quad \mathit{unit} \ = \ \mathit{Ok}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad\ Error}\ \textbf{where}\\ \quad \begin{array}{lcl} \mathit{join}\ (\mathit{Ok}\ x) & = & x\\ \mathit{join}\ (\mathit{Error\ msg}) & = & \mathit{Error\ msg}\end{array}\end{array}$$

As with *Maybe*, we can use the *prod* construction to obtain a composition of *Error* (on the left) with any monad m. Suitable definitions and proofs can be obtained by replacing each reference to *Just* with *Ok* and each *Nothing* with a value of the form *Error msg* in Section 6.1. Alternatively, we can use the *swap* construction, based on the definition:

$$\begin{array}{l}\textbf{instance}\ \mathit{SComposable}\ m\ \mathit{Error}\ \textbf{where}\\ \quad \begin{array}{lcl} \mathit{swap}\ (\mathit{Ok}\ m) & = & \mathit{map\ Ok}\ m\\ \mathit{swap}\ (\mathit{Error\ msg}) & = & \mathit{unit}\ (\mathit{Error\ msg})\end{array}\end{array}$$

6.5.3 The *Tree* monad

The final example we will consider is tree monad, defined by the constructor *Tree*, and mapping a type a to the type *Tree* a of binary trees with leaf values of type a. The monad structure is given by the following declarations:

$$\textbf{data}\ \mathit{Tree}\ a \ = \ \mathit{Leaf}\ a \ \mid \ \mathit{Tree}\ a :\!\wedge\!: \mathit{Tree}\ a$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Functor}\ \mathit{Tree}\ \textbf{where}\\ \quad \begin{array}{lcl} \mathit{map}\ f\ (\mathit{Leaf}\ x) & = & \mathit{Leaf}\ (f\ x)\\ \mathit{map}\ f\ (\mathit{lx} :\!\wedge\!: \mathit{rx}) & = & \mathit{map}\ f\ \mathit{lx}\ :\!\wedge\!:\ \mathit{map}\ f\ \mathit{rx}\end{array}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Premonad}\ \mathit{Tree}\ \textbf{where}\\ \quad \mathit{unit} \ = \ \mathit{Leaf}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ \mathit{Tree}\ \textbf{where}\\ \quad \begin{array}{lcl} \mathit{join}\ (\mathit{Leaf}\ m) & = & m\\ \mathit{join}\ (\mathit{lm} :\!\wedge\!: \mathit{rm}) & = & \mathit{join}\ \mathit{lm}\ :\!\wedge\!:\ \mathit{join}\ \mathit{rm}\end{array}\end{array}$$

Composition of the *Tree* monad with a monad m can be described using the *swap* construction with:

$$\begin{array}{l}\textbf{instance}\ \mathit{SComposable}\ m\ \mathit{Tree}\ \textbf{where}\\ \quad \begin{array}{lcl} \mathit{swap}\ (\mathit{Leaf}\ m) & = & [\,\mathit{Leaf}\ m \mid x \leftarrow m\,]\\ \mathit{swap}\ (\mathit{lm} :\!\wedge\!: \mathit{rm}) & = & [\,\mathit{lx} :\!\wedge\!: \mathit{rx} \mid \mathit{lx} \leftarrow \mathit{swap}\ \mathit{lm},\ \mathit{rx} \leftarrow \mathit{swap}\ \mathit{rm}\,]\end{array}\end{array}$$

As in the case of list monads, the proof of S(4) depends on a commutativity property for m. The proofs for S(1)–S(3) however (and hence, the proofs of the monad laws (4)–(6) for *SComp* m *Tree*) do not require any special properties of m.

7 A simple example: an evaluator

In this section, we show how two of the monad constructions introduced above can be used in a small, but practical, example – an expression evaluator. To make the example more interesting we will use an environment mapping variables to values (with a careful treatment of unbound variables) and we will provide a simple trace facility.

We will base the expression evaluator on the following types representing values, variable names, environments (i.e. mappings from variable names to values) and expressions:

$$\begin{array}{lcl}\textbf{type}\ \mathit{Value} & = & \mathit{Int}\\ \textbf{type}\ \mathit{Name} & = & \mathit{String}\\ \textbf{type}\ \mathit{Env} & = & [(\mathit{Name}, \mathit{Value})]\\ \textbf{data}\ \mathit{Expr} & = & \mathit{Const}\ \mathit{Value} \mid \mathit{Var}\ \mathit{Name} \mid \mathit{Expr} :\!+\!: \mathit{Expr} \mid \mathit{Trace}\ \mathit{String}\ \mathit{Expr}\end{array}$$

To keep this example short, we have restricted ourselves to a very simple expression language, allowing only constants, variables, addition and a simple trace mechanism.

The evaluator itself will need to access variables bound in an environment, may produce output if the trace facility is used, and requires some form of error handling to deal with unbound variables. This could be captured by defining a monad structure for the type constructor:

$$\textbf{type}\ M\ a \quad = \quad Env \rightarrow Writer\ [String]\ (Error\ a),$$

working out suitable versions of the monad operators for this particular type. If we're going to be really fussy, we should also verify the monad laws for whatever definitions we choose for these operators. Given that the definition of M is quite complex, this is likely to require a long, error-prone, and largely uninspiring calculation.

The alternative, using the tools introduced in this report, is to recognize that the definition of M can be expressed as a composition of monads. Assuming that all the components are themselves monads, our results mean that we can use this approach without requiring any further proofs:

$$\textbf{type}\ M\ a \quad = \quad DComp\ (Env \rightarrow)\ (SComp\ (Writer\ [String])\ Error)\ a$$

In all honesty, we have to admit that this looks rather ugly. However, in a language designed from scratch to support this kind of work, we might reasonably expect to be able to define M in a less clumsy manner, using an equation of the form:

$$M \quad = \quad (Env \rightarrow)\ .\ Writer\ [String]\ .\ Error$$

We will also use the following functions, defined in terms of the general *left* and *right* functions introduced in Section 5.4, to embed computations in each of the component monads into the full monad M:

$$\begin{array}{lcl} inError & :: & Error\ a \rightarrow M\ a \\ inError & = & right\ .\ right \\[1ex] inReader & :: & (Env \rightarrow a) \rightarrow M\ a \\ inReader & = & left \\[1ex] inWriter & :: & Writer\ [String]\ a \rightarrow M\ a \\ inWriter & = & right\ .\ left \end{array}$$

It is possible to package these functions in a more general way using the overloading mechanism (all have types of the form $N\ a \rightarrow M\ a$ for some constructor N), but we will not consider this any further here.

Before we can give the definition of the evaluator, we need a simple utility function to determine the value bound to a particular variable in a given environment. We will use the *Maybe* type to enable us to deal with the two cases, depending on whether the variable is bound or not in the given environment.

$$\begin{array}{lcl} lookup & :: & Name \rightarrow Env \rightarrow Error\ Value \\ lookup\ x\ ((n, v) : env) & = & \textbf{if}\ x == n\ \textbf{then}\ Ok\ v\ \textbf{else}\ lookup\ x\ env \\ lookup\ x\ [\,] & = & Error\ (\text{"}unbound\ variable\ \text{"} +\!\!+\ x) \end{array}$$

The complete evaluator is defined as follows, with a single case for each possible form of expression:

$$
\begin{array}{lcl}
eval & :: & Expr \rightarrow M\ \ Value \\
eval\ (Const\ v) & = & [\,unit\ v\,] \\
eval\ (Var\ n) & = & [\quad x \quad |\ r \leftarrow inReader\ (lookup\ n),\ x \leftarrow inError\ r\,] \\
eval\ (e :\!+\!: f) & = & [\,x + y\ |\ x \leftarrow eval\ e,\ y \leftarrow eval\ f\,] \\
eval\ (Trace\ m\ e) & = & [\quad x \quad |\ x \leftarrow eval\ e, \\
 & & \qquad\quad () \leftarrow inWriter\ (write\ [m +\!\!+ \text{"} = \text{"} +\!\!+ show\ r])\,]
\end{array}
$$

Finally, we provide a function that can be used to execute a computation in the M monad, using a given environment value and returning a string as its result:

$$
\begin{array}{lcl}
result & :: & Text\ a \Rightarrow M\ \ a \rightarrow Env \rightarrow String \\
result\ m\ env & = & unlines\ ([\text{"}Output :\text{"}] +\!\!+ s +\!\!+ [\text{"}Result :\ \text{"} +\!\!+ val]) \\
 & & \textbf{where}\ Result\ s\ x \ = \ open\ (open\ m\ env) \\
 & & \qquad\quad val \qquad\quad = \ \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad\qquad\qquad\quad Ok\ x \qquad\ \rightarrow\ show\ x \\
 & & \qquad\qquad\qquad\qquad\quad Error\ msg\ \rightarrow\ msg
\end{array}
$$

For example, using the programs described here in the Gofer interpreter with the expression $testExpr = Trace\ \text{"}sum\text{"}\ \ (Const\ \ 1\ :\!+\!:\ Const\ \ 2) :\!+\!:\ Var\ \text{"}x\text{"}$ gives the following results:

```
? result (eval testExpr) []
Output:
sum = 3
Result: unbound variable x

? result (eval testExpr) [("x",42)]
Output:
sum = 3
Result: 45
```

8 Other ways to combine monads

Having spent so much time concentrating on the composition of monads, it is important to point out that there are other methods that can be used to combine monads.

In some cases, a composition of monads is not suitable because it implies a certain level of independence between the components than may not be desired. For example, one common application of monads is to model state-based computations using *state transformers*; i.e. functions taking an initial state and returning a pair containing a final state and a return value of some type. This is described, for example in [5], using a type constructor:

$$\textbf{data}\ State\ s\ a \ = \ ST\ (s \rightarrow (s, a))$$

The type s used here gives the type of values used for the state while a represents the type of return values. The standard functor and monad structure are given by the declarations:

$$\begin{array}{l}\textbf{instance}\ \mathit{Functor}\ (\mathit{State}\ s)\ \textbf{where}\\ \quad \mathit{map}\ f\ (\mathit{ST}\ st) \ = \ \mathit{ST}\ (\backslash s \to \textbf{let}\ (s', x) = st\ s\ \textbf{in}\ (s', f\ x))\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Premonad}\ (\mathit{State}\ s)\ \textbf{where}\\ \quad \mathit{unit}\ x \ = \ \mathit{ST}\ (\backslash s \to (s, x))\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ (\mathit{State}\ s)\ \textbf{where}\\ \quad \mathit{join}\ (\mathit{ST}\ m) \ = \ \mathit{ST}\ (\backslash s \to \textbf{let}\ (s', \mathit{ST}\ m') = m\ s\ \textbf{in}\ m'\ s')\end{array}$$

Notice that, as a type constructor, $\mathit{State}\ s$ is isomorphic to the composition of a reader monad with a writer monad, $\mathit{DComp}\ (s \to)\ (\mathit{Writer}\ s)$. However, the monad structure associated with these two constructors is very different. For example, the *unit* function for the state monad returns the initial state value unchanged, while the *unit* operator for the composition returns the *zero* of s (with the additional constraint that s is a monoid). Perhaps there is another general construction for combining two monads that could be applied to $(s \to)$ and $\mathit{Writer}\ s$ to obtain a state monad?

As a more interesting example, we can combine a simple state monad with an arbitrary using the following definitions, adapted from [5]:

$$\textbf{data}\ \mathit{StateM}\ m\ s\ a \ = \ \mathit{STM}\ (s \to m\ (s, a))$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Functor}\ (\mathit{StateM}\ m\ s)\ \textbf{where}\\ \quad \mathit{map}\ f\ (\mathit{STM}\ xs) \ = \ \mathit{STM}\ (\backslash s \to [\,(s', f\ x)\ \mid\ (s', x) \leftarrow xs\ s\,])\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Premonad}\ (\mathit{StateM}\ m\ s)\ \textbf{where}\\ \quad \mathit{unit}\ x \ = \ \mathit{STM}\ (\backslash s \to \mathit{unit}\ (s, x))\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (\mathit{StateM}\ m\ s)\ \textbf{where}\\ \quad \mathit{join}\ (\mathit{STM}\ xss) \ = \ \mathit{STM}\ (\backslash s \to [\,(s'', x)\ \mid\ (\mathit{STM}\ xs, s') \leftarrow xss\ s,\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (s'', x) \leftarrow xs\ s'\,])\end{array}$$

For example, a monad of this form is often used in the construction of a combinator parser [4, 14, 15]. In this kind of application, we use the stream of tokens to be parsed as the state. Possible choices for the monad m include:

- The *List* monad, to deal with ambiguous grammars.
- The *Maybe* monad, to support backtracking parsers.
- The *Error* monad, to provide error handling.
- A composition of *Maybe* and *Error*, to allow both error handling and backtracking.

Looking only at the types involved, we can express $\mathit{StateM}\ m\ s$ as a composition of a reader, the monad m, and a writer:

$$\mathit{StateM}\ m\ s \ = \ (s \to)\ .\ m\ .\ \mathit{Writer}\ s.$$

However, using the definitions in this report, the monad structure for the composition on the right is very different from the monad structure for the constructor *StateM m s* on the left given by the declarations above. Perhaps this combination of a state monad with another arbitrary monad can be described as an instance of some more general construction for monad combination?

Another way to view many of the examples in this report is as a generalization of the concept of a monad to include a 'hole' that can be filled with another monad to obtain suitable combinations of features. For example, the constructors *PComp m Maybe*, *DComp* $(r \rightarrow)$ *m*, and *StateM m s* are all examples of this with a hole represented by the parameter m. A similar idea motivates the recent work of Steele [13] except that the holes are built into so-called *pseudomonad* operators rather than individual constructors. The pseudomonad operators are difficult to express properly in the Haskell type system and appear to require some form of existential or recursive typing. Fortunately, it is quite easy to express these operators using constructor classes:

```
class Premonad p => Pseudomonad p where
  pbind      :: Monad m => p a -> (a -> m (p b)) -> m (p b)
  pjoin      :: Monad m => p (m (p a)) -> m (p a)

  pjoin m     = m `pbind` id
  m `pbind` f = pjoin (map f m)
```

The *pbind* function is included for those familiar with [13] and provides a pseudomonad version of the monadic *bind* operator mentioned in Section 2. The *pjoin* function is not used by Steele, but is included because of its close relationship to the *join*-based formulation of monads used in this report. Strictly speaking, only one of the *pbind* and *pjoin* functions is actually required to define an instance of the *Pseudomonad* class; the default definitions provided by the last two lines of the class declaration show how each can be defined in terms of the other.

The types of both *pjoin* and *pbind* are very similar to the types of the corresponding monad operators, except that they also make use of an additional parameter representing an arbitrary monad m. Replacing this parameter with the identity monad gives the familiar monad operators. On the other hand, this extra parameter enables us to describe composition in a simple, elegant manner. Starting with the definition of a new composer:

```
data Comp f g x = CC (f (g x))

instance Composer Comp where
  open (CC x) = x
  close       = CC
```

the composition of an arbitrary pseudomonad p with an arbitrary monad m, yielding a composite monad *Comp m p*, can be described by the following instance declaration:

```
instance (Pseudomonad p, Monad m) => Monad (Comp m p) where
  join = wrap (join . map pjoin)
```

In this way, we can build up a chain of pseudomonads, $p_1, \ldots, p_n$, with the final hole plugged by a monad m:

$$Comp\ (\ldots\ (Comp\ (Comp\ m\ p_1)\ p_2)\ \ldots)\ p_n,$$

or, using an infix dot instead of *Comp*, just:

$$(\cdots((m \cdot p_1) \cdot p_2)\ \cdots\ \cdot p_n).$$

This certainly seems like a promising approach, and we hope to investigate its relationship with the constructions used here more fully in future work.

9 Conclusions

We have presented three different constructions that can be used to compose monads and shown how these can be encoded and used in practical programming problems to provide a combination of the features offered by the component monads. The proofs required to establish the monad laws for each composition require only simple equational reasoning (sometimes with structural induction), although they can be a little long. On the other hand, we have already developed a small library describing compositions with certain standard monads which can be extended to include other monads as necessary. These results can already be used to construct new monads without any further proof obligations.

One surprising aspect of this work is the need to restrict our attention to commutative monads in compositions with the *List* monad. This additional property is necessary only to establish the monad law (7), sometimes referred to as the associative law for monads. In practice, there are several examples where the basic framework suggested by the types of the monad operators is useful in practical programming examples, even though the corresponding monad laws are not all satisfied. Examples of this include the strictness monad in [14], state transformers in [10] and 'composable contexts' in [7]. In a similar way, we expect the composition of arbitrary monads with *List* may still be useful in practical programming applications, even though the associativity law does not always hold. Perhaps functional programmers will be prepared to sacrifice the algebraic properties of a full monad, gaining wider application of the techniques of the monadic style of programming as a reward.

Acknowledgments

The work described in this report grew out of a conversation between the two authors on the `comp.lang.functional` newsgroup in December 1992; we are both grateful for access to this forum. Mark Jones's work is supported in part by a grant from ARPA, contract number N00014-91-J-4043. Luc Duponcheel, much of whose work has been carried out as a recreation and hobby, would like to thank Maritza and Jos for their

support, and Alcatel Bell for the opportunity to spend a little time on this kind of research oriented work.

References

[1] L. Duponcheel. A short note on monad and monad morphism composition. Alcatel Bell Telephone, Antwerp, B-2018, Belgium. Manuscript, September 1993.

[2] C. Hall, K. Hammond, W. Partain, S.L. Peyton Jones and P. Wadler. The Glasgow Haskell compiler: a retrospective. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, Ayr, Scotland, July 1992. Springer Verlag Workshops in computing series.

[3] P. Hudak, S.L. Peyton Jones and P. Wadler (eds.). Report on the programming language Haskell, version 1.2. *ACM SIGPLAN notices*, 27, 5, May 1992.

[4] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, volume 2, part 3, July 1992.

[5] M.P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the 6th ACM conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, ACM Press, June 1993.

[6] M. Jones and P. Hudak. Implicit and Explicit Parallel Programming in Haskell. Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-982, August 1993.

[7] R.B. Kieburtz, B. Agapiev and J. Hook. Three Monads for Continuations. In *Proceedings of the ACM SIGPLAN Workshop on Continuations*, Department of Computer Science, Stanford University, Report No. STAN-CS-92-1462, June 1992.

[8] D.J. King and P. Wadler. Combining monads. In *Proceedings of the Fifth Annual Glasgow Workshop on Functional Programming*, Ayr, Scotland, Springer Verlag Workshops in Computer Science, 1992.

[9] E. Moggi. Computational lambda-calculus and monads. *IEEE Symposium on Logic in Computer Science*, Asilomar, California, 1989.

[10] M. Odersky, D. Rabin and P. Hudak. Call by name, assignment, and the lambda calculus. In *20th Annual Symposium on Principles of Programming Languages*, Charleston, South Carolina, 1993.

[11] S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *20th Annual Symposium on Principles of Programming Languages*, Charleston, South Carolina, 1993.

[12] M. Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1), 1990.

[13] G.L. Steele Jr. Building interpreters by composing monads. To appear in proceedings of *21st annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, Portland, Oregon, January 1994

[14] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2, 4 (December 1992). An earlier version appears in *Conference on LISP and Functional Programming*, Nice, France, June 1990.

[15] P. Wadler. The essence of functional programming. In *19th Annual Symposium on Principles of Programming Languages*, Santa Fe, New Mexico, January 1992.

Appendix: On the non-existence of a natural join

In Section 3, we commented that, in a certain sense, it is impossible to construct a *join* function for the composition of two monads using only the operators of the component monads. We will now sketch a proof of this claim.

A full justification of the approach that we have used goes a little beyond the scope of this paper. In particular, we need to recognize the fact that, from a categorical perspective, the type of the *map* function involves two different kinds of function – arrows between objects and arrows between arrows. These distinctions are lost in functional languages like Haskell or Gofer. Nevertheless, we believe that this result, as well as the techniques used in its proof, are likely to be of interest to some readers, and we have therefore decided to outline some of the details in this appendix.

Suppose that we are working in a category with two monads given by endofunctors M and N. In the most general case, the well-typed terms that can be constructed setting using only the functors and the natural transformations *unit* and *join* for the two monads are precisely those terms which can be obtained using the following set of typing rules:

$$a \xrightarrow{id} a$$

$$\frac{b \xrightarrow{f} c \quad a \xrightarrow{f} b}{a \xrightarrow{f\ \cdot\ g} c}$$

$$\frac{a \xrightarrow{f} b}{M\ a \xrightarrow{map_M\ f} M\ b} \qquad \frac{a \xrightarrow{f} b}{N\ a \xrightarrow{map_N\ f} N\ b}$$

$$a \xrightarrow{unit_M} M\ a \qquad a \xrightarrow{unit_N} N\ a$$

$$M\ (M\ a) \xrightarrow{join_M} M\ a \quad N\ (N\ a) \xrightarrow{join_N} N\ a$$

(Of course, this assumes only the very basic properties of a category. Richer categories may also have operations for forming products, sums, exponentials etc. In addition, we have no guarantee of full abstraction; there may well be arrows in the underlying category that cannot be obtained by these rules. After all, our main aim is to use exactly this kind of arrow (i.e. a *prod*, *dorp* or *swap* function) to construct the composition!)

Using this characterization, we will show that there is no way to construct a term with type $M\ (N\ (M\ (N\ a))) \rightarrow M\ (N\ a)$, and hence there is no natural *join* function for the composition of M with N. We can regard the types in the rules above as purely formal expressions and, for convenience, will write types like $M\,(N\,(M\,(N\,X)))$ as strings $MNMNX$. We will also use the notation $rd\ X$ for the string obtained by removing all adjacent duplicates from X. For example, $rd\ MMNMNNX = MNMNX$. The result that we want follows from the following lemma, proved by a simple structural induction.

LEMMA **1** *If* $\vdash X \rightarrow Y$ *then* $rd\ X$ *is a suffix of* $rd\ Y$*.*

Since $MNMNX$ is not a suffix of MNX, it follows that there is no way to define a *join* function with the required type. Similar arguments can be used to show that there is also no way to define natural *prod*, *dorp* or *swap* functions in this framework.