Constructing Configurable Applications by Combining Monads

Chris Angus
Department of Computing Science,
University of Newcastle upon Tyne,
NE1 7RU,
United Kingdom.
Electronic mail: `c.m.angus@ncl.ac.uk`
URL:`http://www.cs.ncl.ac.uk/people/c.m.angus/`

Abstract

Monads provide a way of structuring functional programs, with many computational behaviours being easily modeled in terms of primitive monads. Most real programs however require a combination of different actions and hence combinations of primitive monads. In practice it is usually possible to construct a monad to support combinations of features. However the way this is achieved is usually ad-hoc.

We show how composite monads may be formed by:

- combining existing monads to form Cartesian products and
- composing monad transformations.

We then demonstrate how these techniques may be used to build configurable applications, in a style reminiscent of Object Oriented programming, whose behaviour depends on the particular monad used. The monad can be selected at run-time and hence the behaviour of an application may be configured at run-time. Using the notation of constructor classes, we demonstrate the technique with an example of a configurable interpreter.

1 Introduction

The idea of a monad has recently become a practical tool for functional programmers. This is due to the fact that a monadic programming style can mimic impure features and increases the ease with which programs may be modified [6]. Sometimes these modifications are of the variety that the end user will wish to perform. For example, in an interpreter the user may wish to:

- turn reference counting On/Off,
- turn tracing On/Off.

What is desirable is the ability to plug in various monads at runtime, i.e.

- Turning tracing On = Run interpreter under a tracing monad *Tracer*,
- Turning reduction counting On = Run interpreter under a reduction-counting monad *Red*,
- Turn both On = Form a monad from *Tracer* and *Red* and run the interpreter under this.

This paper describes a way of performing this combining of features for arbitrary monads. We assume familiarity with the motivation for monads and their use in structuring functional programs, and also familiarity with constructor classes; Wadler [7, 6] provides an introduction to the former and Jones to the latter [1]. All examples are given in Haskell 1.3[1] [5].

2 Combining Monads I

As an introduction to monads and how we may combine them we note that a monad is an abstract datatype M with the two operations $bind :: M\ a \to (a \to M\ b) \to M\ b$ and $return :: a \to M\ a$ which must satisfy the three monad laws:

(1)	Left unit:	$bind\,(return\ a)\ k = k\ a$
(2)	Right unit:	$bind\,m\ return = m$
(3)	Associative:	$bind\,m\ (\lambda a \to bind\,(k\ a))\ h) = bind\,(bind\,m\ (\lambda a \to k\ a))\ h$

Two common monads are the state monad St and the exception monad Ex as presented in [3]:

$$\mathbf{type}\ St\ a = State \to (State, a)$$

This says that a state transformer on a type a is a function from a state to a pair consisting of a value of type a and a new state. For example, we may use a state in an interpreter to hold the current environment. The exception monad has the form:

$$\mathbf{data}\ Ex\ a = Return\ a | Raise\ Exception$$

This is a sum type with constructors *Return* and *Raise*. The monad may be used to structure an interpreter so that either it returns a value, or an error message.

King and Wadler [3] give two distinct approaches to combining Ex with St; either

$$\mathbf{type}\ StEx\ a = State \to (State, Ex\ a)$$

or

$$\mathbf{type}\ ExSt\ a = State \to Ex(State, a)$$

and these have entirely different meanings. In the first model, an expression that raises an exception also returns a state. This monad may be thought of as modeling the language Standard ML, where the state survives when an exception is raised. In the second model, if an expression raises an exception the state is not returned. The trouble with this sort of combining is that it is not modular and is usually performed in a very ad-hoc manner creating a monolithic combined monad. We note that it is *possible* to combine monads in this fashion and consider the problem of modularity later.

3 Use of combined monads

In this section we use the above approach to construct an interpreter which we make configurable in terms of tracing and reduction-count information, or combinations of these (including none).

[1] We use $(\lambda a \to a \circ a)$ style definitions meaning `(\a -> a.a)`. This is purely for readability.

First we define a monad to represent that the amount of monitoring information initially desired is *none*. As expected, this is the identity monad which takes the form:

$$
\begin{array}{lcl}
\textbf{type}\ Id\,a & = & a \\
return^{Id}\ a & = & a \\
bind^{Id}\ k\ cont & = & cont\ k
\end{array}
$$

We also define an interpreter in terms of a monad $InterpM$ as in [7]:

$$
\begin{array}{lcl}
\textbf{type}\ InterpM\ a & = & Environment \to (Environment, a) \\
return^{InterpM}\ x & = & \lambda env \to (env, x) \\
bind^{InterpM}\ k\ cont & = & \lambda env \to \textbf{let}\ (env', r) = k\ env\ \textbf{in}\ cont\ r\ env' \\
get^{InterpM} & = & \lambda env \to (env, env) \\
put^{InterpM}\ x & = & \lambda env \to (x, ())
\end{array}
$$

In the code examples that follow `(>>=)` is taken to be an infix version of the *bind* operation for an arbitrary monad. This is so that $InterpM$ may be considered a member of the constructor class $Monad$[2].

$$
\begin{array}{lcll}
\textbf{type}\ Name & = & String & \\
\textbf{data}\ Term & = & Var\ Name \mid Con\ Int \mid Add\ Term\ Term & \\
 & \mid & Lam\ Name\ Term \mid App\ Term\ Term & \\
\textbf{data}\ Value & = & Wrong \mid Num\ Int & \\
 & \mid & Fun(Value \to (Binding, Term)) & \\
\textbf{type}\ Binding & = & (Name, Value) & \\
\textbf{type}\ Environment & = & [Binding] & \\
 & & & \\
interp & :: & Term \to InterpM\ Value & \\
interp\ (Var\ x) & = & getInterpM & >>= \quad \lambda env \to \\
 & & lookUp\ env\ x & \\
interp\ (Con\ i) & = & return\ (Num\ i) & \\
interp\ (Add\ u\ v) & = & interp\ u & >>= \quad \lambda u \to \\
 & & interp\ v & >>= \quad \lambda v \to \\
 & & add\ u\ v & \\
interp\ (Lam\ x\ v) & = & return\ (Fun(\lambda a \to ((x, a), v))) & \\
interp\ (App\ t\ u) & = & interp\ t & >>= \quad \lambda f \to \\
 & & interp\ u & >>= \quad \lambda a \to \\
 & & app\ f\ a & \\
 & & & \\
lookUp\ []\qquad\quad y & = & return\ Wrong & \\
lookUp\ ((x, y) : xs)\ e & = & \textbf{if}\ (x == e)\ \textbf{then}\ return\ y\ \textbf{else}\ lookUp\ xs\ e & \\
 & & & \\
app\ (Fun\ k)\ a & = & \textbf{let}\ (e, v)\ =\ k\ a & \\
 & & \textbf{in}\ getInterpM & >>= \quad \lambda env \to \\
 & & \quad putInterpM\ (e : env) & >> \\
 & & \quad interp\ v & >>= \quad \lambda r \to \\
 & & \quad putInterpM\ env & >> \\
 & & \quad return\ r & \\
app\ f\ a & = & return\ Wrong & \\
 & & & \\
add\ (Num\ i)\ (Num\ j) & = & return\ (Num\ (i + j)) & \\
add\ \_\qquad\quad \_ & = & return\ Wrong &
\end{array}
$$

We now have a definition of a standard call-by-value interpreter.

[2]In Haskell 1.3 *InterpM* can only be a member of class *Monad* if *InterpM* is a datatype defined with *data* rather than a type synonym defined with *type*. This does not affect our reasoning, other than make the presentation more complex via the wrapping and unwrapping of data constructors. As in [4] we use *type* declarations as if they work as *data* declarations unless this simplifies things *too* far.

3.1 Generalising the Interpreter

We may generalise the above to include monitoring information by redefining the monad $InterpM$ as:

$$\textbf{type}\ InterpM\ a = Environment \to Mon(Environment, a)$$

where Mon models the variety of monitoring required, e.g. if $Mon{=}Id$ then we perform no monitoring. Since we have altered $InterpM$ we must redefine its functions to plumb in the internal monad Mon, although this is trivial.

$$\begin{array}{lcl}
result^{InterpM}\ x & = & \lambda env \to result^{Mon}(env, x) \\
bind^{InterpM}\ k\ cont & = & \lambda env \to bind^{Mon}\ (k\ env)\ (\lambda(env', r) \to cont\ r\ env') \\
get^{InterpM} & = & \lambda env \to result^{Mon}(env, env) \\
put^{InterpM}\ x & = & \lambda env \to result^{Mon}(x, ())
\end{array}$$

3.2 Monitoring functions

We now require functions for tracing and reduction counting so we specify that any monad used as a monitor must have functions to perform reduction counting and tracing whose types are:

$$\begin{array}{lcl}
tick^{Mon} & :: & Mon() \\
trace^{Mon} & :: & String \to Mon()
\end{array}$$

We define *tick* and *trace* functions for *InterpM* in terms of $tick^{Mon}$,$trace^{Mon}$ to to lift $tick^{Mon}$ and $trace^{Mon}$ into *InterpM*:

$$\begin{array}{lcl}
tick & :: & InterpM() \\
tick & = & \lambda env \to bind^{Mon}\ tick^{Mon}(\lambda r \to result^{Mon}(env, r)) \\
trace & :: & String \to InterpM() \\
trace\ string & = & \lambda env \to bind^{Mon}(trace^{Mon}\ string)(\lambda\, res \to result^{Mon}(env, res))
\end{array}$$

and alter the interpreter to include reduction counting:

$$\begin{array}{lcll}
app\ (Fun\ k)\ a & = & \textbf{let}\ (e, v)\ =\ k\ a & \\
 & & \textbf{in}\ getInterpM & >\!>\!= \quad \lambda env \to \\
 & & \quad putInterpM\ (e : env) & >\!> \\
 & & \quad interp\ v & >\!>\!= \quad \lambda r \to \\
 & & \quad putInterpM\ env & >\!> \\
 & & \quad tick & >\!> \\
 & & \quad return\ r & \\
 & & & \\
add\ (Num\ i)\ (Num\ j) & = & tick & >\!> \\
 & & return\ (Num\ (i + j)) &
\end{array}$$

We have now added reduction counting without having to change the definitions of $bind^M$ and $result^M$. However, we must incorporate *tick* and *trace* functions into the identity monad *Id* in order to use *Id* as a monitor (and since this monad models zero monitoring, these are just identities):

$$\begin{array}{lcl}
tick^{Id} & = & result^{Id}() \\
trace^{Id} string & = & result^{Id}()
\end{array}$$

Alternatively we can define a monitor monad which performs reduction counting:

$$
\begin{array}{lcl}
\textbf{type}\ Red\, a & = & Int \to (Int, a) \\
result^{Red}\ a & = & \lambda int \to (int, a) \\
bind^{Red}\ k\ cont & = & \lambda int \to \textbf{let}\ (int', r) = k\ int\ \textbf{in}\ cont\ r\ int' \\
trace^{Red} string & = & result^{Red}() \\
tick^{Red} & = & \lambda int \to (int + 1, ())
\end{array}
$$

which is no more than a standard state transformer on type *Int*.

Now, if we let *Mon=Red* in the definition of *InterpM*, the interpreter counts reductions. If, however, we let *Mon=Id*, as before, it does not count reductions. So in effect we have an On/Off toggle switch for reduction-counting by means of supplying either monad *Red* or monad *Id*. This replacement is preformed by textually replacing all occurences of *Mon* in the program code by either *Id* or *Red* and re-compiling. Obviously this is not an acceptable way to select behaviour. In the next section we show how this replacement, and hence behaviour selection, may be performed at run-time.

4 Tidying things up

We generalise the interpreter so that monad passing is performed explicitly via the use of constructor classes [1], i.e. we alter the type of *InterpM* from

$$\textbf{type}\ InterpM\ a = Environment \to Mon(Environment, a)$$

to

$$\textbf{type}\ InterpM\ mon\ a = Environment \to mon(Environment, a)$$

and the type of the interpreter becomes

$$interp :: Monad\, m \Rightarrow Term \to M\ m\ Value$$

Instead of defining the monad *InterpM* in terms of a specific monad *Mon* we parameterise it in terms of any monad *mon*, i.e. we convert *InterpM* from a monad to a monad transformer [4] and define its operations in terms of its argument monad *mon* .

$$
\begin{array}{l}
\textbf{instance}\ Monad\, mon \Rightarrow Monad\ (InterpM\ mon)\ \textbf{where} \\
\quad \begin{array}{lcll}
return\ x & = & \lambda env \to & return(env, x) \\
k\ >>= cont & = & \lambda env \to & k\ env \qquad >>=\ (\lambda(env', r) \to \\
 & & & cont\ r\ env')
\end{array} \\
\begin{array}{lcl}
getInterpM & = & \lambda env \to result\ (env, env) \\
putInterpM\ x & = & \lambda env \to result\ (x, ())
\end{array}
\end{array}
$$

Additionally, we can define a constructor class *Monitor* which represents the class of monads used as monitors such that the identity *trace* and *tick* functions are derived automatically. In object oriented terms this is a virtual base class with default methods:

$$
\begin{array}{l}
\textbf{class}\ Monad\, mon \Rightarrow Monitor\ mon\ \textbf{where} \\
\quad \begin{array}{lcl}
trace & :: & String \to InterpM\ mon\ () \\
tick & :: & InterpM\ mon\ () \\
trace\ s & = & return\ () \\
tick & = & return\ ()
\end{array}
\end{array}
$$

and lift these operations into (*InterpM mon*) by making (*InterpM mon*) an instance of *Monitor* for all monitors *mon*:

$$
\begin{array}{l}
\textbf{instance}\ Monitor\ mon \Rightarrow Monitor\ (InterpM\ mon)\ \textbf{where} \\
\quad \begin{array}{lcll}
tick & = & \lambda env \to tick & >>= (\lambda r \to\ return(env, r)) \\
trace\ s & = & \lambda env \to (trace\ s) & >>= (\lambda r \to\ return(env, r))
\end{array}
\end{array}
$$

The type of the interpreter then becomes

$$interp :: Monitor\ mon \Rightarrow Term \rightarrow InterpM\ mon\ Value$$

representing the fact that the monad used for monitoring must be a member of the class *Monitor*.

If we make the identity monad *Id*, and reduction counter *Red*, instances of *Monad*

$$\begin{array}{l}
\mathbf{instance}\ Monad\ Id\ \mathbf{where} \\
\quad \begin{array}{lcl} return\ x & = & x \\ k\ >\!\!>\!\!=\ cont & = & cont\ k \end{array} \\
\mathbf{instance}\ Monad\ Red\ \mathbf{where} \\
\quad \begin{array}{lcl} return\ x & = & \lambda state \rightarrow (state, x) \\ k\ >\!\!>\!\!=\ cont & = & \lambda int \rightarrow \mathbf{let}\ (i, a)\ =\ k\ int\ \mathbf{in}\ cont\ a\ i \end{array}
\end{array}$$

and *Monitor*:

$$\begin{array}{l}
\mathbf{instance}\ Monitor\ Id \\
\\
\mathbf{instance}\ Monitor\ Red\ \mathbf{where} \\
\quad tick\ =\ \lambda int \rightarrow (int + 1, ())
\end{array}$$

the interpreter can be executed as a simple interpreter $interp :: Term \rightarrow InterpM\ Id\ Value$ using monad *Id*, or as a reduction-counting interpreter $interp :: Term \rightarrow InterpM\ Red\ Value$ using monad *Red* and no recompilation is necessary.

4.1 Tracing

We now extend the interpreter to perform tracing, as follows:

$$\begin{array}{lcl}
app\ (Fun\ k)\ a & = & \mathbf{let}\ (e, v)\ =\ k\ a \\
& & \mathbf{in}\ \begin{array}{lll} getInterpM & >\!\!>\!\!= & \lambda env \rightarrow \\ \quad putInterpM\ (e : env) & >\!\!> & \\ \quad trace\ (apTell_1\ a) & >\!\!> & \\ \quad interp\ v & >\!\!>\!\!= & \lambda r \rightarrow \\ \quad trace\ (apTell_2\ a) & >\!\!> & \\ \quad putInterpM\ env & >\!\!> & \\ \quad tick & >\!\!> & \\ \quad return\ r & & \end{array} \\
\\
add\ (Num\ i)\ (Num\ j) & = & \begin{array}{lll} tick & >\!\!> & \\ return\ (i + j) & >\!\!>\!\!= & \lambda k \rightarrow \\ trace\ (addTell\ i\ j\ k) & >\!\!> & \\ return\ (Num\ k) & & \end{array}
\end{array}$$

$$\begin{array}{lcl}
apTell_1\ a & = & concat\ [\text{"}Function\ application\ arg(\text{"}, showval\ a, \text{"})\backslash n\text{"}] \\
apTell_2\ a & = & concat\ [\text{"}Function\ application\ result(\text{"}, showval\ a, \text{"})\backslash n\text{"}] \\
addTell\ x\ y\ z & = & concat\ [\text{"}Add\ called\ with\ (\text{"}, show\ x, \text{"}, \text{"}, show\ y, \text{"}) \rightarrow \text{"}, show\ z, \text{"}\backslash n\text{"}]
\end{array}$$

and define a monad *Tracer*[3]

$$
\begin{array}{l}
\textbf{data}\, \mathit{Tracer}\ a \ = \ \mathit{Tracer}\ (\mathit{IO}\ a) \\
\mathit{invTrace}\ (\mathit{Tracer}\ a) \ = \ a \\
\\
\textbf{instance}\ \mathit{Monad}\ \mathit{Tracer}\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{return}\ x & = & \mathit{Tracer}\ (\mathit{return}\ x) \\
k \ >>= \ \mathit{cont} & = & \mathit{Tracer}\ ((\mathit{invTrace}\ k) \ >>= \ (\mathit{invTrace} \circ \mathit{cont}))
\end{array} \\
\\
\textbf{instance}\ \mathit{Monitor}\ \mathit{Tracer}\ \textbf{where} \\
\quad \mathit{trace}\ \mathit{string} \ = \ \mathit{Tracer}\ (\mathit{putStr}\ \mathit{string})
\end{array}
$$

Now, at runtime, if we pass *Id* to the interpreter we still get the original interpreter; if we pass *Red* to the interpreter we get a reduction-counting interpreter; and if we pass *Trace* to the interpreter it traces the evaluation of the abstract syntax tree.

5 Combining Monads II

Suppose however we require an interpreter which both traces execution and counts reductions. How may we achieve this, given that monads exist which model each action? One possible answer is to form the Cartesian product of the two monads and run the interpreter under this. Given two arbitrary monads R and T we may combine them to form a combined monad C as their Cartesian product:

$$\textbf{type}\, C\ a = (R\ a, T\ a)$$

with *return* and *bind* operations:

$$
\begin{array}{lcl}
\mathit{return}^C & :: & a \rightarrow C\ a \\
\mathit{return}^C x & = & (\mathit{return}^R\ x, \mathit{return}^T\ x) \\
\mathit{bind}^C & :: & C\ a \rightarrow (a \rightarrow C\ b) \rightarrow C\ b \\
\mathit{bind}^C x f & = & (\mathit{bind}^R\ (\mathit{fst}\ x)\ (\mathit{fst} \circ f), \mathit{bind}^T\ (\mathit{snd}\ x)\ (\mathit{snd} \circ f))
\end{array}
$$

These satisfy the three monad laws

(1)	Left unit:	$\mathit{bind}\ (\mathit{return}\ a)\ k = k\ a$
(2)	Right unit:	$\mathit{bind}\ m\ \mathit{return} = m$
(3)	Associative:	$\mathit{bind}\ m\ (\lambda a \rightarrow \mathit{bind}\ (k\ a))\ h) = \mathit{bind}\ (\mathit{bind}\ m\ (\lambda a \rightarrow k\ a))\ h$

The structure C, together with bind^C and return^C, forms a monad. A formal proof can be found at `http://www.cs.ncl.ac.uk/people/c.m.angus/home.formal/mproof.ps`.

The Cartesian product may be integrated into the interpreter as follows:

$$
\begin{array}{l}
\textbf{type}\, \mathit{Cartesian}\ m\ n\ a \ = \ (m\ a, n\ a) \\
\\
\textbf{instance}\ (\mathit{Monad}\ m, \mathit{Monad}\ n) \Rightarrow \mathit{Monad}\ (\mathit{Cartesian}\ m\ n)\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{return}\ a & = & (\mathit{return}\ a, \mathit{return}\ a) \\
k \ >>= \ \mathit{cont} & = & (\mathit{fst}\ k \ >>= \ (\mathit{fst} \circ \mathit{cont}), \mathit{snd}\ k \ >>= \ (\mathit{snd} \circ \mathit{cont}))
\end{array} \\
\\
\textbf{instance}\ (\mathit{Monad}\ m, \mathit{Monad}\ n) \Rightarrow \mathit{Monad}\ (\mathit{Cartesian}\ m\ n)\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{tick} & = & (\mathit{tick}, \mathit{tick}) \\
\mathit{trace}\ \mathit{string} & = & (\mathit{trace}\ \mathit{string}, \mathit{trace}\ \mathit{string})
\end{array}
\end{array}
$$

If we execute the interpreter under (*Cartesian Tracer Ref*) or (*Cartesian Ref Tracer*) we have a tracing, reduction-counting interpreter and the job is complete.

[3] We use a *data* declaration here as a type synonym would simplify matters too far.

6 Combining Monads III

The advantages of the technique of the last section are that it requires no internal knowledge of the monads in question, and is a quick and simple way of combining two monads when there is no interaction between them. For example, if we take the Cartesian product of two tracers we have a situation similar to running one tracer, then the other. The idea of separation here is more important than ordering. By forming the Cartesian product we are ensuring that one monad *cannot* affect the other, so if both output to the same stream then to be separated one *must* precede the other. However, it is conceivable that these could output to different windows in a GUI system. We now show how monad transformers may be composed so that, for example, two tracers may be combined to execute in lockstep fashion.

6.1 Monad Transformers

Monad transformers are arrows between monads, and their power lies in the fact that they can be composed[4]. For example consider a state monad transformer

$$\textbf{type}\ m\ a = State \to m(State, a)$$

and apply it to *Id*, the identity monad, the result is the standard state monad[5]:

$$\begin{aligned} StT\ Id\ a &= State \to Id(State, a) \\ &= State \to (State, a) \end{aligned}$$

Alternatively, if we compose two state monad transformers, i.e. compose the two arrows between monads and apply this to *Id*, this results in a monad with two state components

$$\begin{aligned} StT_1\ (StT_2\ Id)\ a &= State_1 \to (StT_2\ Id)(State_1, a) \\ &= State_1 \to State_2 \to Id(State_2, (State_1, a)) \\ &= State_1 \to State_2 \to (State_2, (State_1, a)) \end{aligned}$$

If we define the tracer and reduction counter as monad transformers we may combine them in the above manner. Before doing this, however, we define classes for monads with operations, in the style of [4]. For instance, the tracer produces output so we can define a class of monads with the operation write:

$$\begin{aligned} &\textbf{class}\ Monad\ m \Rightarrow OMonad\ m\ \textbf{where} \\ &\quad write\ ::\ String \to m\ () \end{aligned}$$

A similar definition can be made for state monads[6]:

$$\begin{aligned} &\textbf{class}\ Monad\ m \Rightarrow StateMonad\ m\ \textbf{where} \\ &\quad update\ ::\ (Int \to Int) \to\ m\ Int \end{aligned}$$

6.2 Tracing

We note that the tracer needs to output messages, so when defining a tracing monad transformer we must pass it a member of class *Monad* which can perform output, i.e. a member of

[4]It is not surprising that these arrows between monads prove useful if we consider a monad's category theoretic origins.

[5]The interpreter *InterpM* is a state monad transformer i.e. the monad (*InterpM Red*) is constructed by passing the monad *Red* to the transformer *InterpM*.

[6]This is simplified to consider only state monads with integer state.

class *OMonad*. We define a tracing monad transformer *TracerT* as follows (note the similarity with the definition of *Tracer*):

$$\begin{array}{l}
\mathbf{data}\, TracerT\ m\ a\ =\ TracerT\ (m\ a)\\
invTracerT\ (TracerT\ a)\ =\ a
\end{array}$$

$$\begin{array}{l}
\mathbf{instance}\, Monad\, m\ \Rightarrow Monad\,(TracerT\ m)\ \mathbf{where}\\
\quad \begin{array}{lcl} return\ a & = & TracerT\ (return\ a)\\ k\ >>=\ cont & = & (invTracerT\ k)\ >>=\ (invTracerT \circ cont) \end{array}
\end{array}$$

$$\begin{array}{l}
\mathbf{instance}\, OMonad\, m \Rightarrow OMonad\,(TracerT\ m)\ \mathbf{where}\\
\quad write\ string\ =\ TracerT\ (write\ string)
\end{array}$$

$$\begin{array}{l}
\mathbf{instance}\ (Monitor\ m, OMonad\, m)\ \Rightarrow Monitor\ (TracerT\ m)\ \mathbf{where}\\
\quad \begin{array}{lcl} trace\ string & = & TracerT\ (trace\ string\ >>\ write\ string)\\ tick & = & TracerT\ tick \end{array}
\end{array}$$

and make *IO* an instance of *OMonad*

$$\begin{array}{l}
\mathbf{instance}\ \ OMonad\, IO\ \mathbf{where}\\
\quad write\ =\ putStr
\end{array}$$

We may define a tracing monad *TracerT(IO)*, or define a monad which combines two tracers in lockstep fashion, *TracerT(TracerT(IO))*. Note: we make *TraceT(IO)* an *OMonad* so that we can apply *TracerT* to it and explicitly pass all the monitor functions through the transformer, regardless of whether that function has any computational effect inside the transformer. This is necessary since the function may have a computational effect in a monad at a lower lever.

6.3 Reduction counting

We now create a reduction-counting monad transformer in terms of a state monad transformer which can be used in conjunction with the tracer above. First we define a state monad transformer and make it instances of *Monad,OMonad,StateMonad* and *Monitor*. We again push the operations of *OMonad* and *Monitor* through the transformer to its argument monad. We define the state monad transformer as:

$$\mathbf{type}\, StateT\ m\ a\ =\ Int \rightarrow m(Int, a)$$

$$\begin{array}{l}
\mathbf{instance}\, Monad\, m\ \Rightarrow Monad\,(StateT\ m)\ \mathbf{where}\\
\quad \begin{array}{lcll} return\ x & = & \lambda s \rightarrow & return\ (s, x)\\ m\ >>=\ k & = & \lambda s_0 \rightarrow & m\ s_0\ >>= \lambda(s_1, a) \rightarrow\\ & & & k\ a\ s_1 \end{array}
\end{array}$$

$$\begin{array}{l}
\mathbf{instance}\, Monad\, m \Rightarrow StateMonad\,(StateT\ m)\ \mathbf{where}\\
\quad update\ f\ =\ \lambda s \rightarrow return(f\ s, s)
\end{array}$$

$$\begin{array}{l}
\mathbf{instance}\, OMonad\, m \Rightarrow OMonad\,(StateT\ m)\ \mathbf{where}\\
\quad write\ string\ =\ \lambda s \rightarrow write\ string\ >>= \lambda a \rightarrow return(s, a)
\end{array}$$

$$\begin{array}{l}
\mathbf{instance}\ (Monitor\ m)\ \Rightarrow Monitor\ (StateT\ m)\ \mathbf{where}\\
\quad \begin{array}{lcl} trace\ str & = & \lambda s \rightarrow trace\ str\ >>= \lambda a \rightarrow return(s, a)\\ tick & = & \lambda s \rightarrow tick\ >>= \lambda a \rightarrow return(s, a) \end{array}
\end{array}$$

and then define the reduction-counting transformer *ReductionT*[7] in terms of it:

```
newtype ReductionT m a = ReductionT (StateT m a)
invReductionT (ReductionT a) = a

instance Monad m ⇒ Monad (ReductionT m) where
    return x      =  ReductionT (return x)
    k >>= cont    =  ReductionT ((invReductionT k) >>= (invReductionT ∘ cont))

instance OMonad m ⇒ OMonad (ReductionT m) where
   write string = ReductionT (write string)

instance (Monitor m) ⇒ Monitor (ReductionT m) where
    tick       =  ReductionT (update (+1) >> tick)
    trace str  =  ReductionT (trace str)
```

We may now define a tracing, reduction-counting monad as *ReductionT (TracerT IO)*, or as *TracerT (ReductionT IO))* and hence define a tracing, reduction-counting interpreter by executing the interpreter under either of these monads. This produces an interpreter which is more efficient to that produced via the use of the Cartesian product.

7 Putting it all together

Now we have defined appropriate interpreters how do we obtain the answer?

If we define a constructor class *Run* this solves the problem:

```
class Monad m ⇒ Run m where
   exec :: m a → IO a
```

with the following instances

```
instance Run Id where
   exec a = return a

instance Run Tracer where
   exec (Tracer t) = t

instance Run Red where
     exec stateM  =  let  (state, result) = stateM 0
                     in   putStrLn ("Reduction count = " ++ show state) >>
                          return result

instance (Run m, Run n) ⇒ Run(Cartesian m n) where
   exec (m, n) = exec m >> exec n

instance Run IO where
   exec a = a

instance Run m ⇒ Run (ReductionT m) where
 exec (ReductionT(StateT m))  =
                                  exec (m 0)                                   >>= λ(c, r) →
                                  putStrLn ("Reduction count = " ++ show c)    >>
                                  return r
```

[7] Again a wrapper is used as to abandon it would confuse the presentation

$$
\begin{array}{l}
\textbf{instance } Run\, m \Rightarrow Run\, (TracerT m) \textbf{ where} \\
\quad exec\, (TracerT\ m) \ = \ exec\, m
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } Run\, m \Rightarrow Run\, (InterpM\ m) \textbf{ where} \\
\quad \begin{array}{lcl} exec\, x & = & exec\, (x\, [])\ >\!\!>\!\!=\ \lambda(env, result) \rightarrow \\ & & return\ result \end{array}
\end{array}
$$

We now run the whole program using

$$
\begin{array}{l}
execM\ :: \ Run\, m \Rightarrow InterpM\ m\ Value \rightarrow IO\ () \\
execM\ x\ = exec\, x\ \ >\!\!>\!\!=\ \ \lambda result \rightarrow putStr\ (showval\ result)
\end{array}
$$

with calls as follows where it is the type of the monitoring monad which defines the exact meaning of the expression:

$$
\begin{array}{l}
choice\ 1\ exp\ =\ execM(interp\ exp :: InterpM\ Id\ Value) \\
choice\ 2\ exp\ =\ execM(interp\ exp :: InterpM\ Tracer\ Value) \\
\quad \vdots \\
choice\ m\ exp\ =\ execM(interp\ exp :: InterpM(Cartesian\ Tracer\ (ReductionT\ Id))\ Value) \\
\quad \vdots \\
choice\ n\ exp\ =\ execM(interp\ exp :: InterpM(ReductionT\ (TracerT\ IO))\ Value)
\end{array}
$$

8 Relation to Previous work

- King and Wadler [3] combine monads in the style of section 2 and attempt to define a library of useful primitive monads which may be combined using certain recipes, looking at the class of monads containing trees, lists, bags and sets and considering how they may be combined with other monads.
- Jones and Duponcheel [2] define 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. They establish a set of laws that the auxiliary function must satisfy to ensure the composition is itself a monad. They demonstrate this with the example of a simple expression evaluator built by composing monads. This is similar to the work presented here although it does not use Cartesian products of monads or monad transformers and this behaviour cannot be selected at run-time.
- Liang, Hudak and Jones [4] show how monad transformers can be composed. Monad transformers work in a similar fashion to *InterpM*, i.e. they are parameterised and accept a monad as input and can be chained together to form composite monads. Their work is geared towards expressing language features as monad transformers and defining interpreters with different semantics as combinations of these. Again this is geared more towards providing different applications rather than configurable ones.

9 Conclusion

In this paper we have presented a way to combine arbitrary monads to form new monads, and how this form of combining may be used to write modular, configurable software by implementing a simple call-by-value interpreter which may be executed in many different configurations.

These configurations were built from their constituent parts. To the best of our knowledge this is the first time that arbitrary monads have been combined in this way and monad transformers used for this purpose. The advantages of the Cartesian product approach are that the programmer need not know about the internal workings of the particular monads used, and it is very simple to use in practice. The disadvantage is that there appears to be a certain amount of recomputation going on when this method is used, similar to the recomputation in a naive implementation of a Fibonacci function. We have also demonstrated how monad transformers may be used to combine features and note that this approach creates more efficient programs than using Cartesian products, but is more involved. A menu driven version of the interpreter presented in this paper extended to include error handling is available at `http://www.cs.ncl.ac.uk/people/c.m.angus/home.formal/interp.lhs`.

10 Further Work

This work was carried out as a precursor to investigating, designing and implementing declarative IDEs and debuggers. It is obvious that implementations need a great deal of configurability which is available to the user at runtime. The ease with which Wadler's use of monads [7] allow the programmer to modify a program's behaviour by replacing the monad specific to that code made us wonder if this could not be done dynamically. One of the questions which definitely needs to be answered with this approach is: how much work is recomputed via the use of the monad Cartesian product and hence should this combining approach be limited to prototyping? It would also be interesting to see if we can integrate this approach with the Glasgow Haskell concurrency extensions so that, for example, in a GUI system different monitors appear in separate windows.

References

[1] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FCPA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, Copenhagen, Denmark, June 1993. ACM Press.

[2] Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Department of Computer Science, Yale University, New Haven, Connecticut, December 1993.

[3] David King and Philip Wadler. Combining monads. In *Proceedings of the Fifth Annual Glasgow Workshop on Functional Programming, Ayr, Scotland*, Workshops in Computer Science. Springer Verlag, July 92.

[4] Sheing Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Janurary 1995.

[5] John Peterson and Kevin Hammond (eds). Report on the programming language Haskell, version 1.3. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, New Haven, Connecticut, May 1996.

[6] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the Twentieth Annual ACM Symposium on Principle s of Programming Languages*, Jan 93.

[7] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 1–14, Janurary 1992.