J. Functional Programming **1** (1): 1–000, January 1993

FUNCTIONAL PEARLS

A Poor Man's Concurrency Monad

Koen Claessen
Chalmers University of Technology
email: koen@cs.chalmers.se

Abstract

Without adding any primitives to the language, we define a concurrency monad transformer in Haskell. This allows us to add a limited form of concurrency to any existing monad. The atomic actions of the new monad are lifted actions of the underlying monad. Some extra operations, such as `fork`, to initiate new processes, are provided. We discuss the implementation, and use some examples to illustrate the usefulness of this construction.

1 Introduction

The concept of a *monad* (Wadler, 1995) is nowadays heavily used in modern functional programming languages. Monads are used to model some form of computation, such as non-determinism or a stateful calculation. Not only does this solve many of the traditional problems in functional programming, such as I/O and mutable state, but it also offers a general framework that abstracts over many kinds of computation.

It is known how to use monads to model concurrency. To do this, one usually constructs an imperative monad, with operations that resemble the Unix `fork` (Jones & Hudak, 1993). For reasons of efficiency and control, *Concurrent Haskell* (Peyton Jones *et al.*, 1996) even provides primitive operations, which are defined outside the language.

This paper presents a way to model concurrency, generalising over arbitrary monads. The idea is to have *atomic* actions in some monad that can be *lifted* into a concurrent setting. We explore this idea within the language; we will not add any primitives.

2 Monads

To express the properties of monads in Haskell, we will use the following type class definition. The bind operator of the monad is denoted by $(\star)$, and the unit operator by `return`.

```
class Monad m where
  (⋆)    :: m α → (α → m β) → m β
  return :: α → m α
```

Furthermore, throughout this paper we will use the so-called *do*-notation as syntactic sugar for monadic expressions. The following example illustrates a traditional monadic expression on the left, and the same, written in do-notation, on the right.

```
expr1  ⋆ λx.               do x ← expr1
expr2  ⋆ λ_.                ; expr2
expr3  ⋆ λy.                ; y ← expr3
return expr4                ; return expr4
```

As an example, we present a monad with output, called the *writer monad*. This monad has an extra operator called `write`. It takes a string as argument, which becomes output in a side effect of the monad. The bind operator (⋆) of the monad has to take care of combining the output of two computations.

A monad having this operator is an instance of the following class.

```
class Monad m ⇒ Writer m where
  write :: String → m ()
```

A typical implementation of such a monad is a pair containing the result of the computation, together with the output produced during that computation.

```
type W α = (α, String)

instance Monad W where
  (a, s) ⋆ k = let (b, s') = k a in (b, s++s')
  return x = (x, "")

instance Writer W where
  write s = ((), s)
```

Note how the bind operator concatenates the output of the two subactions.

Most monads come equipped with a *run* function. This function executes a computation, taking the values inside one level downwards. The monad `W` has such a run function, we call it `output`, which returns the output of a computation in `W`.

```
output        :: W α → String
output (a, s) = s
```

2.1 Monad Transformers

Sometimes, a monad is parametrised over another monad. This is mostly done to add more functionality to an existing monad. In this case we speak of a *monad transformer* (Liang *et al.*, 1995). An example is the exception monad transformer; it adds a way to escape a monadic computation with an error message. In general, operations that work on one specific monad can be *lifted* into the new, extended monad.

Again, we can express this by using a type class.

$$\begin{array}{l}\texttt{class MonadTrans } \tau \texttt{ where} \\ \quad \texttt{lift} \ :: \ \texttt{Monad } m \ \Rightarrow \ m\ \alpha \rightarrow (\tau\ m)\ \alpha\end{array}$$

A type constructor τ forms a monad transformer if there is an operation `lift` that transforms any action in a monad m into an action in a monad $\tau\ m$.

In this paper we will discuss a monad transformer called `C`. It has the interesting property that any monadic action that is lifted into the new monad will be considered an *atomic* action in a concurrent setting. Also some extra operations are provided for this monad, for example `fork`, which deals with process initiation.

3 Concurrency

How are we going to model concurrency? Since we are not allowed to add primitives to the language, we are going to simulate concurrent processes by *interleaving* them. Interleaving implements concurrency by running the first part of one process, suspending it, and then allowing another process to run.

3.1 Continuations

To suspend a process, we need to grab its future and stick it away for later use. *Continuations* are an excellent way of doing this. We can change a function into *continuation passing style* by adding an extra parameter, the continuation. Instead of producing the result directly, the function will now apply the continuation to the result. We can view the continuation as the *future* of the computation, as it specifies what to do with the result of the function.

Given a computation type `Action`, a function that uses a continuation with result type α has the following type.

$$\texttt{type C } \alpha \ = \ (\alpha \rightarrow \texttt{Action}) \rightarrow \texttt{Action}$$

The type `Action` contains the actual computation. Since, in our case, we want to parametrise this over an arbitrary monad, we want `Action` (and also `C`) to be dependent on a monad m.

$$\texttt{type C } m\ \alpha \ = \ (\alpha \rightarrow \texttt{Action } m) \rightarrow \texttt{Action } m$$

`C` is the concurrency monad transformer we use in this paper. That means that `C` m is a monad, for every monad m.

$$\begin{array}{l} \texttt{instance Monad } m \Rightarrow \texttt{Monad (C } m\texttt{) where} \\ \quad f \star k \qquad = \lambda c.\ f\ (\lambda a.\ k\ a\ c) \\ \quad \texttt{return } x = \lambda c.\ c\ x \end{array}$$

Sequencing of continuations is done by creating a new continuation for the left computation that contains the right computation. The unit operator just passes its argument to the continuation.

3.2 Actions

The type `Action` m specifies the actual actions we can do in the new monad. What does this type look like? For reasons of simplicity, flexibility, and expressiveness (Scholz, 1995), we implement it as a datatype that describes the different actions we provide in the monad.

First of all, we need atoms, which are computations in the monad m. We are inside a continuation, so we want these atomic computations to return a new action. Also, we need a constructor for creating new processes. Lastly, we provide a constructor that does not have a continuation; we will use it to end a process. We also call this the empty process.

$$\begin{array}{l} \texttt{data Action } m \\ \quad = \texttt{Atom } (m\ (\texttt{Action } m)) \\ \quad\ \mid \texttt{Fork } (\texttt{Action } m)\ (\texttt{Action } m) \\ \quad\ \mid \texttt{Stop} \end{array}$$

To express the connection between an expression of type `C` $m\ \alpha$ and an expression of type `Action` m, we define a function `action` that transforms one into the other. It finishes the computation by giving it the `Stop` continuation.

$$\begin{array}{lcl} \texttt{action} & :: & \texttt{Monad } m \Rightarrow \texttt{C } m\ \alpha \rightarrow \texttt{Action } m \\ \texttt{action } m & = & m\ (\lambda a.\ \texttt{Stop}) \end{array}$$

To make the constructors of the datatype `Action` easily accessible, we can define functions that correspond to them. They will create an action in the monad `C` m.

The first function is the function `atom`, which turns an arbitrary computation in the monad m into an atomic action in `C` m. It runs the atomic computation and monadically returns a new action, using the continuation.†

$$\begin{array}{lcl} \texttt{atom} & :: & \texttt{Monad } m \Rightarrow m\ \alpha \rightarrow \texttt{C } m\ \alpha \\ \texttt{atom } m & = & \lambda c.\ \texttt{Atom (do } a \leftarrow m\ ;\ \texttt{return } (c\ a)) \end{array}$$

† This is actually the monadic map, but because `Functor` is not a superclass of `Monad` in Haskell we cannot use `map`.

In addition, we have a function that uses the `Stop` constructor, called `stop`. It discards any continuation, thus ending a computation.

```
stop :: Monad m ⇒ C m α
stop  = λc. Stop
```

To access `Fork`, we define two operations. The first, called `par`, combines two computations into one by forking them both, and passing the continuation to both parts. The second, `fork`, resembles the more traditional imperative fork. It forks its argument after turning it into an action, and continues by passing () to the continuation.

```
par       :: Monad m ⇒ C m α → C m α → C m α
par m1 m2  = λc. Fork (m1 c) (m2 c)

fork      :: Monad m ⇒ C m α → C m ()
fork m     = λc. Fork (action m) (c ())
```

The type constructor `C` is indeed a monad transformer. Its lifting function is the function `atom`; every lifted action becomes an atomic action in the concurrent setting.

```
instance MonadTrans C where
  lift = atom
```

We have now defined ways to construct actions of type `C` m α, but we still can not *do* anything with them. How do we model concurrently running actions? How do we interpret them?

3.3 Semantics

At any moment, the status of the computation is going to be modelled by a list of (concurrently running) actions. We will use a scheduling technique called *round-robin* to interleave the processes. The concept is easy: if there is an empty list of processes, we are done. Otherwise, we take a process, run its first part, take the continuation, and put that at the back of the list. We keep doing this recursively until the list is empty.

We implement this idea in the function `round`.

```
round :: Monad m ⇒ [Action m] → m ()
round []       = return ()
round (a : as) = case a of
  Atom am    → do a' ← am ; round (as ++ [a'])
  Fork a1 a2 → round (as ++ [a1, a2])
  Stop       → round as
```

An `Atom` monadically executes its argument, and puts the resulting process at the back of the process list. `Fork` creates two new processes, and `Stop` discards its process.

As for any monad, we need a run function for `C` m as well. It just transforms its argument into an action, creates a singleton process list, and applies the round-robin function to it.

$$\begin{aligned} &\texttt{run} && :: \texttt{Monad}\ m \Rightarrow \texttt{C}\ m\ \alpha \rightarrow m\ () \\ &\texttt{run}\ m && = \texttt{round}\ [\texttt{action}\ m] \end{aligned}$$

As we can see, the type α disappears in the result type. This means that we lose the result of the original computation. This seems very odd, but often (and in the cases of the examples in this paper) we are only interested in the *side effects* of a computation. It is possible to generalise the type of `run`, but that goes beyond the scope of this paper.

4 Examples

We will present two examples of monads that can be lifted into the new concurrent world.

4.1 Concurrent Output

Recall the writer monad example from Sect. 2. We can try lifting this monad into the concurrent world. To do this, we want to say that every instance of a writer monad can be lifted into a concurrent writer monad.‡

$$\begin{aligned} &\texttt{instance Writer}\ m \Rightarrow \texttt{Writer}\ (\texttt{C}\ m)\ \texttt{where} \\ &\quad \texttt{write}\ s\ = \texttt{lift}\ (\texttt{write}\ s) \end{aligned}$$

The function `lift` here is the `atom` of the monad transformer `C`. Every `write` action, after lifting, becomes an atomic action. This means that no computation will produce output while another `write` is writing.

Before we present an example, we first define an auxilary function `loop`. This function works in any writer monad. It takes one argument, a string, and writes it repeatedly to the output.

$$\begin{aligned} &\texttt{loop} && :: \texttt{Writer}\ m \Rightarrow \texttt{String} \rightarrow m\ () \\ &\texttt{loop}\ s && = \texttt{do write}\ s\ ;\ \texttt{loop}\ s \end{aligned}$$

We use this function to define a computation in $C\ m\ \alpha$ that creates two processes that are constantly writing. One process writes the string "`fish`", the other writes "`cat`".

‡ Actually, we want to say this for all monad transformers at once, but Haskell does not currently allow us to express this.

```
example :: Writer m ⇒ C m ()
example = do write "start!"
             ; fork (loop "fish")
             ; loop "cat"
```

The result of the expression `output (run example)` looks like the following string.

```
"start!fishcatfishcatfishcatfishcatfishcatfishca ..."
```

Because we defined `write` as an atomic action, the writing of one "`fish`" and one "`cat`" cannot interfere. If we want finer grained behaviour, we can split one write action into several write actions, e.g. the separate characters of a string. A simple way of doing this is to change the lifting of `write`.

```
instance Writer m ⇒ Writer (C m) where
  write []      = return ()
  write (c : s) = do lift (write [c]) ; write s
```

The lifting is now done character-by-character. The result of the expression `output (run example)` now looks like this.

```
"start!fciasthcfaitschafticsahtfciasthcfaitscha ..."
```

4.2 Merging of Infinite Lists

A well known problem, called the *merging of infinite lists*, is as follows. Suppose we have an infinite list of infinite lists, and want to collapse this list into one big infinite list. The property we want to hold is that every element in any of the original lists is reachable within a finite number of steps in the new list. This technique is for example used to prove that the set $\mathbf{Q}$ of rationals has a countable number of elements.

Using the writer monad with the new lifting, we can solve this problem for an infinite list of infinite strings. The idea is that, for each string, we create a process that writes the string. If we fork this infinite number of processes, and run the resulting computation, the output will be the desired infinite string.

We will take a step back in order to present a piece of useful theory. There are monads that have a so-called *monoidal* structure on them. That means that there is an operator, denoted by $(+\!\!+)$, that combines two computations of the same type into one, and that there is an identity element for this operation, called `zero`. In Haskell, we can say:

```
class Monad m ⇒ Monoidal m where
  (++)  :: m α → m α → m α
  zero  :: m α
```

The function `concat`, with type $\texttt{Monoidal}\ m \Rightarrow [m\ \alpha] \to \alpha$, uses $(+\!\!+)$ and `zero` to concatenate a (possibly infinite) list of such computations together.

The reason we are looking at this is that $\texttt{C}\ m$ admits a monoidal structure; the parallel composition `par` represents the $(+\!\!+)$, and the process `stop` represents its identity element `zero`.

$$
\begin{array}{l}
\texttt{instance Monad}\ m \Rightarrow \texttt{Monoidal (C}\ m\texttt{) where} \\
\quad (+\!\!+) \ \ = \ \texttt{par} \\
\quad \texttt{zero} \ = \ \texttt{stop}
\end{array}
$$

This means we can use `concat` to transform an infinite list of processes into a process that concurrently runs these computations. To merge an infinite list of infinite strings, we transform every string into a writing process, fork them with `concat`, and extract the output.

$$
\begin{array}{lcl}
\texttt{merge} & :: & [\texttt{String}] \to \texttt{String} \\
\texttt{merge} & = & \texttt{output} \circ \texttt{run} \circ \texttt{concat} \circ \texttt{map write}
\end{array}
$$

Of course, this function also works for finite lists, and can be adapted to act on more general lists than strings.

4.3 Concurrent State

In Haskell, the so-called `IO` monad provides *mutable state*. Within the monad we can create, access, and update pieces of storage. The type of a storage that contains an object of type α is $\texttt{Var}\ \alpha$. The functions we use to control these `Var`s, the non-proper morphisms of `IO`, have the following types.

$$
\begin{array}{lcl}
\texttt{newVar} & :: & \texttt{IO (Var}\ \alpha\texttt{)} \\
\texttt{readVar} & :: & \texttt{Var}\ \alpha \to \texttt{IO}\ \alpha \\
\texttt{writeVar} & :: & \texttt{Var}\ \alpha \to \alpha \to \texttt{IO}\ ()
\end{array}
$$

In the lifted version of this monad, the `C IO` monad, we can have several concurrent processes sharing pieces of state. In a concurrent world however, we often want more structure on shared state. Concurrent Haskell (Peyton Jones *et al.*, 1996), an extension of Haskell with primitives for creating concurrent processes, recognised this. It introduces a new form of shared state: the `MVar`.

Like a `Var`, an `MVar` can contain a value, but it may also be empty. An `MVar` becomes empty after a process has done a read operation on it. Processes reading an empty `MVar` will block, until a new value is put into the `MVar`. `MVar`s are a powerful mechanism for creating higher level concurrent data abstractions. They can for example be used for synchronization and data sharing at the same time.

It is possible to integrate `MVar`s with our concurrency monad transformer, using the mutable state primitives we already have. First, we have to think of how to represent an `MVar`. An `MVar` can be in two different states; it can either be full (containing some value), or empty.

```
type MVar α    = Var (Maybe α)
data Maybe α  = Just α | Nothing
```

We use the datatype `Maybe` to indicate that there is `Just` a value in an `MVar`, or `Nothing` at all.

Let us now define the operations that work on `MVar`s. The function that creates an `MVar` lifts the creation of a `Var`, and puts `Nothing` in it.

```
newMVar :: C IO (MVar α)
newMVar = lift ( do v ← newVar
                  ; writeVar v Nothing
                  ; return v )
```

We can use the same trick when writing to an `MVar`.[§]

```
writeMVar     :: MVar α → α → C IO ()
writeMVar v a  = lift ( writeVar v (Just a) )
```

The hardest function to define is `readMVar`, since it has to deal with blocking. To avoid interference when reading an `MVar`, we perform an atomic action that pulls the value out of the `Var` and puts `Nothing` back. We introduce an auxilary function `takeVar`, working on the unlifted `IO` monad, that does this.

```
takeVar    :: MVar α → IO (Maybe α)
takeVar v  = do a_m ← readVar v
                ; writeVar v Nothing
                ; return a_m
```

Once we have this function, the definition of a blocking `readMVar` is not hard anymore. We represent blocking by repeatedly trying to read the variable. We realise that this busy-wait implementation is very inefficient, and we indeed have used other methods as well (such as the one used in (Jones, M. *et al.*, 1997)), but we present the easiest implementation here.

```
readMVar    :: MVar α → C IO α
readMVar v  = do a_m ← lift (takeVar v)
                 ; case a_m of
                     Nothing  → readMVar v
                     Just a   → return a
```

Note that `readMVar` itself is not an atomic action, so other processes can also read the `MVar` just after `takeVar`. Fortunately, at that point, the `MVar` is already blocked by the function `takeVar`. It is impossible for `readMVar` to be atomic, since other processes deserve a chance when it is blocking on an `MVar`.

[§] We are a bit sloppy here; the real semantics of `MVar`s is slightly different (Peyton Jones *et al.*, 1996).

For some examples of the use of `MVars`, we refer the reader to the paper about Concurrent Haskell (Peyton Jones *et al.*, 1996), where `MVars` are introduced.

5 Discussion

The work presented in this paper is an excellent example of the flexiblity of monads and monad transformers. The power of dealing with different types of computations in this way is very general, and should definitely be more widely used and supported by programming languages. We really had to push the Haskell type class mechanism to its limits in order to make this work. A slightly extended class mechanism would have been helpful (Peyton Jones *et al.*, 1997).

To show that this idea is more than just a toy, we have used this same setting to add concurrency to the graphical system *TkGofer* (Vullinghs *et al.*, 1996). The system increased in expressive power, and its implementation in simplicity. It turns out to be a very useful extension to TkGofer.

We have also experimented with lifting other well-known monads into this concurrent setting. Lifted lists, for example, can be used to express the infinite merging problem more concisely. However, a problem with the type system forced us to fool it in order to make this work. Exception and environment monads (Wadler, 1995) do have the expected behaviour, though we are not able to lift all of the non-proper morphisms of these monads. This is because some of them need a computation as an *argument*, so that lifting becomes non-trivial.

However, there are a few drawbacks. We have not implemented *real* concurrency. We simply allow interleaving of *atomic* actions, whose atomicity plays a vital role in the system. If one atomic action itself does not terminate, the concurrent computation of which it is a part of does not terminate either. We cannot change this, because we cannot step outside the language to interrupt the evaluation of an expression.

The source code of the functions and classes mentioned in this paper is publically available at http://www.cs.chalmers.se/~koen/Code/pearl.hs. It also contains another, more efficient but slightly bigger implementation of `MVars`.

Acknowledgements

I would like to thank Richard Bird, Byron Cook, Andrew Moran, Thomas Nordin, Andrei Sabelfeld, Mark Shields, Ton Vullinghs, and Arjan van Yzendoorn for their useful comments on earlier drafts of this paper. Most of the work for this paper was done while visiting the Oregon Graduate Institute, and an earlier version was used as part of my Master's thesis at the University of Utrecht, under supervision of Erik Meijer.

References

Jones, M., & Hudak, P. (1993). Implicit and Explicit Parallel Programming in Haskell. Yale University. Tech. Rep. YALEU/DCS/RR-982.

Jones, M. *et al.* (1997). The Hugs System. Nottingham University and Yale University. Url: http://www.haskell.org.

Liang, Sh., Hudak, P., & Jones, M. (1995). Monad Transformers and Modular Interpreters. *Conference Record of 22nd POPL '95*. ACM.

Peyton Jones, S., Gordon, A., & Finne, S. (1996). Concurrent Haskell. *Proceedings of the 23rd POPL '96*. ACM.

Peyton Jones, S., Jones, M., & Meijer, E. (1997). Type Classes: An Exploration of the Design Space. *Proceedings of the Haskell Workshop of the ICPF '97*. ACM.

Scholz, E. (1995). A Concurrency Monad Based on Constructor Primitives. Universität Berlin.

Vullinghs, T., Schulte, W., & Schwinn, T. 1996 (June). *An Introduction to Tk-Gofer*. Tech. rept. 96-03. University of Ulm. Url: http://www.informatik.uni-ulm.de/pm/ftp/tkgofer.html.

Wadler, Ph. (1995). Monads for Functional Programming. *Advanced Functional Programming*. Lecture Notes in Computer Science. Springer Verlag.