Lecture 7: Monads and IO

Søren Haagerup

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

December 5, 2016

Some slides of this presentation are borrowed from
http://foswiki.cs.uu.nl/foswiki/pub/USCS/CourseSchedule/C2-Monads-final.pdf

Introduction

The **monad abstraction** is a widely used design pattern in modern Functional Programming.

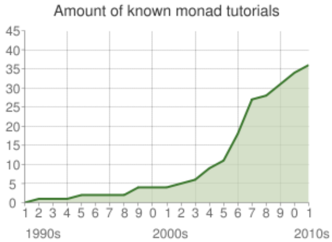
- **1960s:** Category theorists invented monads to concisely express certain aspects of universal algebra.
- **1970s:** Functional programmers invented list comprehensions to concisely express certain programs involving lists.
- **1992:** List comprehensions were generalized to arbitrary monads in 1992 by Philip Wadler - the resulting programming feature can concisely express **pure functional programs** that
  - manipulate state,
  - handle exceptions etc.

Introduction

- **1996:** Monads became a "first-class citizen" of Haskell, with
  - special syntax for monads: the **do**-notation
  - the IO-support of the language exposed as a monad

Abstraction, intuition, and the "monad tutorial fallacy"

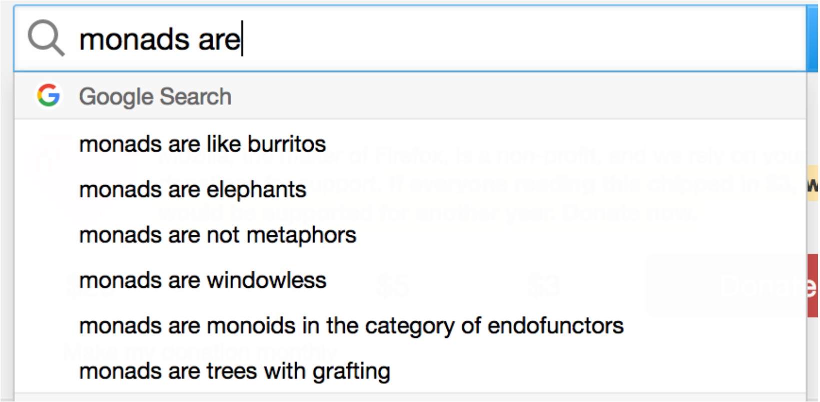
- How to learn and gain intuition about abstract concepts?
  - Begin with the concrete - then move to the abstract.



- Throughout the 2000s, the concept of monads have been treated in numerous blog posts of varying quality
- In a sense, this culminated with the **Monads are Burritos** tutorial

Monads in memes

Let’s ask Google what a monad is...



Examples of monads

A type constructor is a **monad** if it implements the **monad operations** (which I have not defined yet).

The monad operations can be used to implement a wide range of concepts:

Error handling	*Non-determinism*
Maybe a Either String a	[a]
Stateful computation	***Side-effectful computation***
Reader a Writer a State a	IO a

Special language syntax for monadic computations

The special syntax is just syntactic sugar for the application of the **monadic operations** (which I have not defined yet).

Haskell

$$\begin{aligned} \mathit{pairs} = \mathbf{do}\ & x \leftarrow xs \\ & y \leftarrow ys \\ & \mathit{return}\ (x, y) \end{aligned}$$

Scala

$$\begin{aligned} \mathit{val\ pairs} = \mathit{for}\ \{\ & x \leftarrow xs \\ & y \leftarrow ys \\ \}\ & \mathit{yield}\ (x, y) \end{aligned}$$

C# (LINQ)

$$\begin{aligned} \mathit{var\ pairs} = & \mathit{from}\ x\ \mathbf{in}\ xs \\ & \mathit{from}\ y\ \mathbf{in}\ ys \\ & \mathit{select\ new}\ \{x, y\}; \end{aligned}$$

The IO monad

The *IO* type constructor is an example of a monad. Code written with this monad is evaluated sequentially, and can make use of the IO operations of the Operating System:

```
main = do
  colors ← forM [1,2,3,4] (λa → do
    putStrLn ("Which color do you associate "
              ++ "with the number " ++ show a ++ "?")
    color ← getLine
    return color)
  putStrLn ("The colors that you associate "
            ++ "with 1, 2, 3 and 4 are: ")
  mapM putStrLn colors
```

Asynchronous computation
Control.Monad.Par

do

$fx \leftarrow spawn\ (return\ (f\ x))$ -- start evaluating (f x)
$gx \leftarrow spawn\ (return\ (g\ x))$ -- start evaluating (g x)
$a \leftarrow get\ fx$ -- wait for fx
$b \leftarrow get\ gx$ -- wait for gx
$return\ (a, b)$ -- return results

"Futures/promises" which exist in many languages, fit nicely into the monad abstraction.

f and g could be functions doing a HTTP POST request, or doing an expensive computation.

Pure vs. impure languages

- Monads can also help with making pure Haskell code shorter and easier to read and write. This is the focus in the beginning of this lecture.
- **Pure languages**
  - easy to reason about (equational reasoning)
  - may benefit from lazy evaluation,
- **Impure languages**
  - offer efficiency benefits
  - sometimes make possible a more compact mode of expression

Using **monads** is an approach to integrate impure effects into pure programs.

“In short, Haskell is the world’s finest imperative programming language”

– Simon Peyton Jones

Problems with pure functional programming

- Pure functional languages have this advantage: All flow of data is made explicit.
- And this disadvantage: Sometimes it is painfully explicit.
- The essence of an algorithm can become buried under the plumbing required to carry data from its point of creation to its point of use.
- We will now exemplify these statements, and later propose solutions based on monads.

Running example: A small interpreter

We introduce the type of terms, and an evaluation function:

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

$$\begin{aligned}
&\mathit{eval} :: \mathit{Term} \to \mathit{Int}\\
&\mathit{eval}\ (\mathit{Con}\ a) = a\\
&\mathit{eval}\ (\mathit{Div}\ t\ u) = \mathit{eval}\ t\ \text{`}\mathit{div}\text{`}\ \mathit{eval}\ u
\end{aligned}$$

Examples of terms

$$\begin{aligned}
&\mathit{answer}, \mathit{err} :: \mathit{Term}\\
&\mathit{answer} = (\mathit{Div}\ (\mathit{Div}\ (\mathit{Con}\ 1972)\ (\mathit{Con}\ 2))\ (\mathit{Con}\ 23))\\
&\mathit{err} \qquad = (\mathit{Div}\ (\mathit{Con}\ 1)\ (\mathit{Con}\ 0))
\end{aligned}$$

We can evaluate...

$$\begin{aligned}
&\mathit{eval}\ \mathit{answer} = 42\\
&\mathit{eval}\ \mathit{err} \qquad = \bot
\end{aligned}$$

Extension 1/3: Adding error handling

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

$$\begin{aligned}
&\mathit{eval} :: \mathit{Term} \to \mathit{Int} \\
&\mathit{eval}\ (\mathit{Con}\ a) = a \\
&\mathit{eval}\ (\mathit{Div}\ t\ u) = \mathit{eval}\ t\ \text{`}\mathit{div}\text{`}\ \mathit{eval}\ u
\end{aligned}$$

- We would like to add **error handling** to *eval*, such that the program will not crash when dividing by zero, but instead return an error that can be used by other parts of the program.
- In impure languages, we would throw an exception
- What can we do in Haskell?

Extension 1/3: Adding error handling

We would like errors to be a *value*, and not $\perp$:

$$\textbf{data}\ \mathit{Error}\ a = \mathit{Raise}\ \mathit{Exception} \mid \mathit{Return}\ a\ \textbf{deriving}\ \mathit{Show}$$
$$\textbf{type}\ \mathit{Exception} = \mathit{String}$$

$$
\begin{array}{l}
\mathit{eval} :: \mathit{Term} \to \mathit{Error}\ \mathit{Int} \\
\mathit{eval}\ (\mathit{Con}\ a) \quad = \ \mathit{Return}\ a \\
\mathit{eval}\ (\mathit{Div}\ t\ u) = \ \textbf{case}\ \mathit{eval}\ t\ \textbf{of} \\
\qquad\qquad\qquad\quad \mathit{Raise}\ e \to \mathit{Raise}\ e \\
\qquad\qquad\qquad\quad \mathit{Return}\ a \to \\
\qquad\qquad\qquad\qquad \textbf{case}\ \mathit{eval}\ u\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad \mathit{Raise}\ e \to \mathit{Raise}\ e \\
\qquad\qquad\qquad\qquad\quad \mathit{Return}\ b \to \\
\qquad\qquad\qquad\qquad\qquad \textbf{if}\ b \equiv 0 \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathit{Raise}\ \texttt{"divide by zero"} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \mathit{Return}\ (a\ \text{`}\mathit{div}\text{`}\ b)
\end{array}
$$

Extension 2/3: Adding state - Counting the number of operations performed

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

$$\mathit{eval} :: \mathit{Term} \to \mathit{Int}$$
$$\mathit{eval}\ (\mathit{Con}\ a) = a$$
$$\mathit{eval}\ (\mathit{Div}\ t\ u) = \mathit{eval}\ t\ \text{`}\mathit{div}\text{`}\ \mathit{eval}\ u$$

- We would like to add a **a count of operations** performed by the evaluation engine.
- In impure languages: increment a global variable
- What can we do in Haskell?

Extension 2/3: Adding state - Counting the number of operations performed

$$
\begin{aligned}
&\textbf{type}\ State\ s\ a = s \rightarrow (a, s)\\
&eval :: Term \rightarrow State\ Int\ Int\\
&eval\ (Con\ a)\ x \quad = (a, x)\\
&eval\ (Div\ t\ u)\ x = \textbf{let}\ (a, y) = eval\ t\ x\ \textbf{in}\\
&\qquad\qquad\qquad\quad\ \textbf{let}\ (b, z) = eval\ t\ y\ \textbf{in}\\
&\qquad\qquad\qquad\quad\ (a\ \text{‘}div\text{‘}\ b, z + 1)
\end{aligned}
$$

$$
\begin{aligned}
&> eval\ answer\ 0\\
&(42, 2)
\end{aligned}
$$

Extension 3/3: Adding an execution trace

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

$$\mathit{eval} :: \mathit{Term} \rightarrow \mathit{Int}$$
$$\mathit{eval}\ (\mathit{Con}\ a) = a$$
$$\mathit{eval}\ (\mathit{Div}\ t\ u) = \mathit{eval}\ t\ \text{`}\mathit{div}\text{`}\ \mathit{eval}\ u$$

- We would like to add an **execution trace** to the program - and write out which calls are made to eval, and what are their arguments?
- In impure languages: Execute a print statement inside eval
- How yould we do in Haskell?

$$> \mathit{putStrLn}\ \$\ \mathit{fst}\ \$\ \mathit{eval}\ \mathit{answer}$$
$$\mathit{eval}\ (\mathit{Con}\ 1972) = 1972$$
$$\mathit{eval}\ (\mathit{Con}\ 2) = 2$$
$$\mathit{eval}\ (\mathit{Div}\ (\mathit{Con}\ 1972)\ (\mathit{Con}\ 2)) = 986$$
$$\mathit{eval}\ (\mathit{Con}\ 23) = 23$$
$$\mathit{eval}\ (\mathit{Div}\ (\mathit{Div}\ (\mathit{Con}\ 1972)\ (\mathit{Con}\ 2))\ (\mathit{Con}\ 23)) = 42$$

Adding an execution trace

$$
\begin{array}{l}
\textbf{type}\ \mathit{Writer}\ a = (\mathit{Output}, a) \\
\textbf{type}\ \mathit{Output} = \mathit{String} \\
\mathit{eval} :: \mathit{Term} \to \mathit{Writer}\ \mathit{Int} \\
\mathit{eval}\ (\mathit{Con}\ a) \quad = (\mathit{line}\ (\mathit{Con}\ a)\ a, a) \\
\mathit{eval}\ (\mathit{Div}\ t\ u) = \textbf{let}\ (x, a) = \mathit{eval}\ t\ \textbf{in} \\
\qquad\qquad\qquad\quad \textbf{let}\ (y, b) = \mathit{eval}\ u\ \textbf{in} \\
\qquad\qquad\qquad\quad (x \mathbin{+\!\!+} y \mathbin{+\!\!+} \mathit{line}\ (\mathit{Div}\ t\ u)\ (a\ \text{`}\mathit{div}\text{`}\ b), a\ \text{`}\mathit{div}\text{`}\ b) \\
\mathit{line} :: \mathit{Term} \to \mathit{Int} \to \mathit{Output} \\
\mathit{line}\ t\ a \qquad\quad = \texttt{"eval ("} \mathbin{+\!\!+} \mathit{show}\ t \mathbin{+\!\!+} \texttt{") = "} \mathbin{+\!\!+} \mathit{show}\ a \mathbin{+\!\!+} \texttt{"\textbackslash n"}
\end{array}
$$

Adding an execution trace

$$> putStrLn \$ fst \$ eval\ answer$$

$$eval\ (Con\ 1972) \leqslant 1972$$
$$eval\ (Con\ 2) \leqslant 2$$
$$eval\ (Div\ (Con\ 1972)\ (Con\ 2)) \leqslant 986$$
$$eval\ (Con\ 23) \leqslant 23$$
$$eval\ (Div\ (Div\ (Con\ 1972)\ (Con\ 2))\ (Con\ 23)) \leqslant 42$$

$$> putStrLn \$ fst \$ eval\ err$$

$$eval\ (Con\ 1) \leqslant 1$$
$$eval\ (Con\ 0) \leqslant 0$$
$$eval\ (Div\ (Con\ 1)\ (Con\ 0)) \leqslant * * *Exception : divide\ by\ zero$$

Adding an execution trace

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

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

  could be changed to

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

- So - explicit data flow gives flexibility, but sometimes the explicit way of programming is too explicit and buries the essence of the algoritm being implemented.

What is a monad?

A monad is a triple $(M, \mathit{return}, (\gg\!=))$ where

- M is a type constructor
- $\mathit{return} :: a \to M\ a$ is a function (also called *unit*)
- $(\gg\!=) :: M\ a \to (a \to M\ b) \to M\ b$ is a function (also called *bind*)

The old code

```
data Error a = Raise Exception | Return a deriving Show
type Exception = String

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

The new code

$$
\begin{aligned}
&\textbf{data}\ \mathit{Error}\ a = \mathit{Raise}\ \mathit{Exception} \mid \mathit{Return}\ a\ \textbf{deriving}\ \mathit{Show}\\
&\textbf{type}\ \mathit{Exception} = \mathit{String}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Term} \rightarrow \mathit{Error}\ \mathit{Int}\\
&\mathit{eval}\ (\mathit{Con}\ a) \quad = \mathit{return}\ a\\
&\mathit{eval}\ (\mathit{Div}\ t\ u) = \textbf{do}\ a \leftarrow \mathit{eval}\ t\\
&\qquad\qquad\qquad\qquad\ \ b \leftarrow \mathit{eval}\ u\\
&\qquad\qquad\qquad\qquad\ \ \textbf{if}\ b \equiv 0\\
&\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \mathit{raise}\ \texttt{"Divide by zero"}\\
&\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ \mathit{return}\ (a\ \text{`}\mathit{div}\text{`}\ b)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{raise} :: \mathit{Exception} \rightarrow \mathit{Error}\ a\\
&\mathit{raise}\ e = \mathit{Raise}\ e
\end{aligned}
$$

The Error monad

$$\textbf{data}\ \mathit{Error}\ a = \mathit{Raise}\ \mathit{Exception} \mid \mathit{Return}\ a\ \textbf{deriving}\ \mathit{Show}$$
$$\textbf{type}\ \mathit{Exception} = \mathit{String}$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Monad}\ \mathit{Error})\ \textbf{where} \\
\quad \mathit{return} \quad = \mathit{Return} \\
\quad m \ggg k = \textbf{case}\ m\ \textbf{of} \\
\qquad \mathit{Return}\ a \to k\ a \\
\qquad \mathit{Raise}\ e \quad \to \mathit{Raise}\ e
\end{array}
$$

Verify the types

- $\mathit{return} :: a \to \mathit{Error}\ a$
- $(\ggg) :: \mathit{Error}\ a \to (a \to \mathit{Error}\ b) \to \mathit{Error}\ b$

Question: How would you define a function f such that $(\mathit{Return}\ x) \ggg f = \mathit{Return}\ (x + 5)$ Note that we need no handling of errors in the *input* given to f.

The **do** notation

The **do**-notation

$$
\begin{aligned}
val = \mathbf{do}\ & a \leftarrow eval\ t \\
& b \leftarrow eval\ u \\
& return\ (a + b)
\end{aligned}
$$

is just syntactic sugar for

$$
\begin{aligned}
val = {} & eval\ t \gg\!= \lambda a \rightarrow \\
& eval\ u \gg\!= \lambda b \rightarrow \\
& return\ (a + b)
\end{aligned}
$$

How to define a monad for *Maybe*?

A simpler version of what we have just seen is:

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

Try to suggest definitions of the following functions:

- $\mathit{return} :: a \to \mathit{Maybe}\ a$
- $(\gg\!=) :: \mathit{Maybe}\ a \to (a \to \mathit{Maybe}\ b) \to \mathit{Maybe}\ b$

Introducing state - The old code

$$
\begin{aligned}
&\textbf{type } \mathit{State}\ s\ a = s \rightarrow (a, s)\\
&\mathit{eval} :: \mathit{Term} \rightarrow \mathit{State}\ \mathit{Int}\ \mathit{Int}\\
&\mathit{eval}\ (\mathit{Con}\ a)\ x \quad = (a, x)\\
&\mathit{eval}\ (\mathit{Div}\ t\ u)\ x = \textbf{let } (a, y) = \mathit{eval}\ t\ x \textbf{ in}\\
&\qquad\qquad\qquad\quad\ \textbf{let } (b, z) = \mathit{eval}\ t\ y \textbf{ in}\\
&\qquad\qquad\qquad\quad\ (a \text{ `}\mathit{div}\text{` } b, z + 1)
\end{aligned}
$$

What we would like to end up with...

$$
\begin{array}{l}
\textbf{type } \mathit{State}\ s\ a = s \to (a, s) \\[4pt]
\mathit{eval} :: \mathit{Term} \to \mathit{State}\ \mathit{Int}\ \mathit{Int} \\
\mathit{eval}\ (\mathit{Con}\ a) \quad = \mathit{return}\ a \\
\mathit{eval}\ (\mathit{Div}\ t\ u) = \textbf{do}\ a \leftarrow \mathit{eval}\ t \\
\qquad\qquad\qquad\qquad\quad b \leftarrow \mathit{eval}\ u \\
\qquad\qquad\qquad\qquad\quad \mathit{tick} \\
\qquad\qquad\qquad\qquad\quad \mathit{return}\ (a\ \text{`}\mathit{div}\text{`}\ b)
\end{array}
$$

Notice that the counting is done with the function *tick*, and keeping track of the state variable is done behind the scenes! But we cannot exactly achieve this with standard Haskell.

We would need to activate the **TypeSynonymInstances** compiler pragma, to declare *State s* as a Monad instance.

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

$$
\begin{aligned}
&return\ a = \lambda x \to (a, x) \\
&m \ggg k \ = \lambda x \to \textbf{let}\ (a, y) = m\ x\ \textbf{in} \\
&\qquad\qquad\qquad\ \ \textbf{let}\ (b, z) = k\ a\ y\ \textbf{in} \\
&\qquad\qquad\qquad\ \ (b, z)
\end{aligned}
$$

$$
\begin{aligned}
&tick :: State\ s\ () \\
&tick = \lambda x \to ((), x + 1)
\end{aligned}
$$

$$
\begin{aligned}
&eval :: Term \to State\ Int\ Int \\
&eval\ (Con\ a) \quad = return\ a \\
&eval\ (Div\ t\ u) = eval\ t \ggg \lambda a \to \\
&\qquad\qquad\qquad\ \ eval\ u \ggg \lambda b \to \\
&\qquad\qquad\qquad\ \ tick \quad\ \ \ggg \backslash\_ \to \\
&\qquad\qquad\qquad\ \ return\ (a\ \text{`}div\text{`}\ b)
\end{aligned}
$$

$$
\begin{aligned}
&> eval\ answer\ 0 \\
&(42, 2)
\end{aligned}
$$

Making it an instance of *Monad*

For technical reasons, we need a **newtype** instead of an ordinary type synonym **type**

$$\textbf{newtype}\ \mathit{State}\ s\ a = \mathit{State}\ \{\mathit{runState} :: s \rightarrow (a, s)\}$$

$$\begin{array}{l}
\textbf{instance}\ (\mathit{Monad}\ (\mathit{State}\ s))\ \textbf{where} \\
\quad \mathit{return}\ a = \mathit{State}\ (\lambda x \rightarrow (a, x)) \\
\quad m \gg\!= k\ \ = \mathit{State}\ (\lambda x \rightarrow \textbf{let}\ (a, y) = \mathit{runState}\ m\ x\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let}\ (b, z) = \mathit{runState}\ (k\ a)\ y\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad (b, z))
\end{array}$$

$$\begin{array}{l}
\mathit{tick} :: M\ () \\
\mathit{tick} = \mathit{State}\ (\lambda x \rightarrow ((), x + 1))
\end{array}$$

So we end up with the nice code that we wanted

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Term} \to \mathit{State}\ \mathit{Int}\ \mathit{Int}\\
&\mathit{eval}\ (\mathit{Con}\ a) \quad = \mathit{return}\ a\\
&\mathit{eval}\ (\mathit{Div}\ t\ u) = \mathbf{do}\ a \leftarrow \mathit{eval}\ t\\
&\qquad\qquad\qquad\qquad\quad\ \ b \leftarrow \mathit{eval}\ u\\
&\qquad\qquad\qquad\qquad\quad\ \ \_ \leftarrow \mathit{tick}\\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ (a\ \text{‘}\mathit{div}\text{‘}\ b)
\end{aligned}
$$

$$
\begin{aligned}
&> \mathit{runState}\ (\mathit{eval}\ \mathit{answer})\ 0\\
&(42, 2)
\end{aligned}
$$

The **do** notation allows us to remove the $\_ \leftarrow$ (values which are thrown away)

So we end up with the nice code that we wanted

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Term} \to \mathit{State}\ \mathit{Int}\ \mathit{Int}\\
&\mathit{eval}\ (\mathit{Con}\ a) \quad = \mathit{return}\ a\\
&\mathit{eval}\ (\mathit{Div}\ t\ u) = \mathbf{do}\ a \leftarrow \mathit{eval}\ t\\
&\qquad\qquad\qquad\qquad\quad\ b \leftarrow \mathit{eval}\ u\\
&\qquad\qquad\qquad\qquad\quad\ \mathit{tick}\\
&\qquad\qquad\qquad\qquad\quad\ \mathit{return}\ (a\ \text{`}\mathit{div}\text{`}\ b)
\end{aligned}
$$

$$
\begin{aligned}
&> \mathit{runState}\ (\mathit{eval}\ \mathit{answer})\ 0\\
&(42, 2)
\end{aligned}
$$

Final variant: Adding execution trace

$$
\begin{array}{l}
\textbf{type } M\ a = (\mathit{Output}, a) \\
\textbf{type } \mathit{Output} = \mathit{String}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{eval} :: \mathit{Term} \to M\ \mathit{Int}} \\
\mathit{eval}\ (\mathit{Con}\ a) & = & (\mathit{line}\ (\mathit{Con}\ a)\ a, a) \\
\mathit{eval}\ (\mathit{Div}\ t\ u) & = & \textbf{let } (x, a) = \mathit{eval}\ t \textbf{ in} \\
 & & \textbf{let } (y, b) = \mathit{eval}\ u \textbf{ in} \\
 & & (x +\!\!+\ y +\!\!+\ \mathit{line}\ (\mathit{Div}\ t\ u)\ (a \text{ `}\mathit{div}\text{' } b), a \text{ `}\mathit{div}\text{' } b)
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{line} :: \mathit{Term} \to \mathit{Int} \to \mathit{Output}} \\
\mathit{line}\ t\ a & = & \texttt{"eval("} +\!\!+\ \mathit{show}\ t +\!\!+\ \texttt{")="} +\!\!+\ \mathit{show}\ a +\!\!+\ \texttt{"\textbackslash n"}
\end{array}
$$

The code that we want to end up with...

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Term} \to M\ \mathit{Int}\\
&\mathit{eval}\ (\mathit{Con}\ a) \quad = \mathbf{do}\ \mathit{out}\ (\mathit{line}\ (\mathit{Con}\ a)\ a)\\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ a\\
&\mathit{eval}\ (\mathit{Div}\ t\ u) = \mathbf{do}\ a \leftarrow \mathit{eval}\ t\\
&\qquad\qquad\qquad\qquad\quad\ \ b \leftarrow \mathit{eval}\ u\\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{out}\ (\mathit{line}\ (\mathit{Div}\ t\ u)\ (a\ \text{`}\mathit{div}\text{`}\ b))\\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ (a\ \text{`}\mathit{div}\text{`}\ b)
\end{aligned}
$$

The definition of the monad...

$$
\begin{aligned}
&\textbf{type}\ M\ a = (Output, a)\\
&\textbf{type}\ Output = String\\[4pt]
&return\ a = (\texttt{""}, a)\\
&m \ggg\!= k \ = \textbf{let}\ (x, a) = m\ \textbf{in}\\
&\qquad\qquad\quad \textbf{let}\ (y, b) = k\ a\ \textbf{in}\\
&\qquad\qquad\quad (x +\!\!+\ y, b)\\[4pt]
&out :: Output \to M\ ()\\
&out\ x \quad\ = (x, ())
\end{aligned}
$$

Technicalities - introducing a **newtype**

$$
\begin{array}{l}
\textbf{newtype}\ M\ a = \mathit{Writer}\ \{\mathit{runWriter} :: (\mathit{Output}, a)\} \\
\textbf{type}\ \mathit{Output} = \mathit{String} \\
\textbf{instance}\ (\mathit{Monad}\ M)\ \textbf{where} \\
\quad \mathit{return}\ a = \mathit{Writer}\ (\texttt{""}, a) \\
\quad m \gg\!= k\ = \textbf{let}\ \mathit{Writer}\ (x, a) = m\ \textbf{in} \\
\qquad\qquad\quad\ \ \textbf{let}\ \mathit{Writer}\ (y, b) = k\ a\ \textbf{in} \\
\qquad\qquad\quad\ \ \mathit{Writer}\ (x +\!\!+\, y, b) \\
\mathit{out} :: \mathit{Output} \to M\ () \\
\mathit{out}\ x \qquad = \mathit{Writer}\ (x, ())
\end{array}
$$

Questions

Recall that a monad is a triple $(M, return, (\gg=))$ where

$$
\begin{aligned}
&return :: a \to M\ a \\
&(\gg=)\ \ :: M\ a \to (a \to M\ b) \to M\ b
\end{aligned}
$$

Suggest definitions for $(\gg=)$ and *return* for the following type constructors:

- **data** $Id\ a = Id\ a$
- **data** $[a] = [\,]\ |\ a : [a]$
- **data** $Tree\ a = Tip\ a\ |\ Node\ (Tree\ a)\ (Tree\ a)$

The List monad

Encoding multiple results and nondeterminism

Get the length of all words in a list of multi-line texts:

$$map\ length\ (concat\ (map\ words\ (concat\ (map\ lines\ txts))))$$

Easier to understand with a list comprehension:

$$[length\ w \mid t \leftarrow txts, l \leftarrow lines\ t, w \leftarrow words\ l]$$

We can also define sequencing and embedding, i.e., $(\ggg\!=)$ and $return$:

$$\begin{aligned}
&(\ggg\!=) :: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b] \\
&xs \ggg\!= f = concat\ (map\ f\ xs) \\
&return :: a \rightarrow [a] \\
&return\ x = [x]
\end{aligned}$$

Four equivalent pieces of code

$$map\ length\ (concat\ (map\ words\ (concat\ (map\ lines\ txts))))$$

$$[length\ w \mid t \leftarrow txts, l \leftarrow lines\ t, w \leftarrow words\ l]$$

$$\begin{array}{l} \mathbf{do}\ t \leftarrow txts \\ \quad\ \ l \leftarrow lines\ t \\ \quad\ \ w \leftarrow words\ l \\ \quad\ \ return\ (length\ w) \end{array}$$

$$\begin{array}{ll} txts & \ggg\!= \lambda t \rightarrow \\ lines\ t & \ggg\!= \lambda l \rightarrow \\ words\ l & \ggg\!= \lambda w \rightarrow \\ return\ (length\ w) & \end{array}$$

Adding *filter* functionality

$$odds = [x \mid x \leftarrow [1\,..\,10], odd\ x]$$

$$\begin{aligned} odds' = \mathbf{do}\ & x \leftarrow [1\,..\,10] \\ & \quad \text{-- how to filter odd x?} \\ & return\ x \end{aligned}$$

$$\begin{aligned} odds' = \mathbf{do}\ & x \leftarrow [1\,..\,10] \\ & \_ \leftarrow \mathbf{if}\ (odd\ x)\ \mathbf{then}\ [()]\ \mathbf{else}\ [\,] \\ & return\ x \end{aligned}$$

$$\begin{aligned} & \mathbf{import}\ Control.Monad \\ & odds' = \mathbf{do}\ x \leftarrow [1\,..\,10] \\ & \qquad\qquad\ \ guard\ (odd\ x) \end{aligned}$$

The Monad Laws

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

- The name "monad" is borrowed from category theory.
- A monad is an algebraic structure similar to a monoid.
- Monads have been popularized in functional programming via the work of Moggi and Wadler.
- Every monad instance should satisfy the following laws:
  1. Left identity: $\mathit{return}\ x \gg\!= k \equiv k\ x$
  2. Right identity: $m \gg\!= \mathit{return} \equiv m$
  3. Associativity: $m \gg\!= (\lambda a \to k\ a \gg\!= l) \equiv (m \gg\!= k) \gg\!= l$

The IO Monad

Another type with actions that require sequencing. The *IO* monad is special in several ways:

- *IO* is a primitive type, and $(\gg\!=)$ and *return* for *IO* are primitive functions,
- I there is no (politically correct) function $runIO :: IO\ a \rightarrow a$, whereas for most other monads there is a corresponding function,
- values of IO a denote side-effecting programs that can be executed by the run-time system.
- Note that the specialty of IO has really not much to do with being a monad.

IO, Internally

$$
\begin{array}{l}
\mathit{Prelude} > :i\ \mathit{IO} \\
\textbf{newtype}\ \mathit{IO}\ a \\
\quad = \mathit{GHC.Types.IO}\ (\mathit{GHC.Prim.State}\ \#\ \mathit{GHC.Prim.RealWorld} \\
\qquad \to (\#\mathit{GHC.Prim.State}\ \#\ \mathit{GHC.Prim.RealWorld}, a\#)) \\
\qquad\quad \text{-- Defined in `GHC.Types'} \\
\textbf{instance}\ \mathit{Monad}\ \mathit{IO} \quad \text{-- Defined in `GHC.Base'} \\
\textbf{instance}\ \mathit{Functor}\ \mathit{IO} \quad \text{-- Defined in `GHC.Base'}
\end{array}
$$

Internally, GHC models *IO* as a state monad having the “real world” as state!

The role of IO in Haskell

More and more features have been integrated into *IO*, for instance:

- classic file and terminal IO
  putStr, *hPutStr*
- references
  newIORef, *readIORef*, *writeIORef*
- access to the system
  getArgs, *getEnvironment*, *getClockTime*
- exceptions
  throwIO, *catch*
- concurrency
  forkIO

IO examples

Stdout output

$$Main > putStr\ \texttt{"Hi"}$$
$$Hi$$
$$Main > \mathbf{do}\ \{putChar\ \texttt{'H'}; putChar\ \texttt{'i'}; putChar\ \texttt{'\textbackslash n'}\}$$
$$Hi$$

Stdin input

$$Main > \mathbf{do}\ \{c \leftarrow getChar; putStrLn\ (\texttt{"'"} +\!\!+ [c] +\!\!+ \texttt{"'"})\}$$
$$z'z'$$

IO examples

File IO

$$Main > \textbf{do}\ \{h \leftarrow openFile\ \texttt{"TMP"}\ WriteMode; hPutStrLn\ h\ \texttt{"Hi"}\}$$
$$Main > :q$$

```
Leaving GHCi
$ cat TMP
Hi
```

IO examples

Side-effect: variables

$$
\begin{aligned}
\mathbf{do}\ & v \leftarrow \mathit{newIORef}\ \texttt{"text"} \\
& \mathit{modifyIoRef}\ v\ (\lambda t \rightarrow t +\!\!+\ \texttt{" and more text"}) \\
& w \leftarrow \mathit{readIORef}\ v \\
& \mathit{print}\ w
\end{aligned}
$$

Results in `text and more text`

The role of *IO* in Haskell (contd.)

- A program that involves *IO* in its type can do everything. The absence of *IO* tells us a lot, but its presence does not allow us to judge what kind of *IO* is performed.
- It would be nice to have more fine-grained control on the effects a program performs.
- For some, but not all effects in *IO*, we can use or build specialized monads.

Lifting functions to monads

$$
\begin{aligned}
&\mathit{liftM} :: (\mathit{Monad}\ m) \Rightarrow (a \to b) \to m\ a \to m\ b \\
&\mathit{liftM2} :: (\mathit{Monad}\ m) \Rightarrow (a \to b \to c) \to m\ a \to m\ b \to m\ c \\
&.. \\
&\mathit{liftM}\ f\ m = \mathbf{do}\ \{x \leftarrow m; \mathit{return}\ (f\ x)\} \\
&\mathit{liftM2}\ f\ m1\ m2 = \mathbf{do}\ \{x1 \leftarrow m1; x2 \leftarrow m2; \mathit{return}\ (f\ x1\ x2)\} \\
&..
\end{aligned}
$$

Question: What is $\mathit{liftM}\ (+1)\ [1\mathinner{..}5]$? **Answer:** Same as $\mathit{map}\ (+1)\ [1\mathinner{..}5]$. The function *liftM* generalizes *map* to arbitrary monads.

Monadic map

$$
\begin{aligned}
&mapM :: (Monad\ m) \Rightarrow (a \to m\ b) \to [a] \to m\ [b] \\
&mapM\_ :: (Monad\ m) \Rightarrow (a \to m\ b) \to [a] \to m\ () \\
&mapM\ f\ [\,] = return\ [\,] \\
&mapM\ f\ (x : xs) = liftM2\ (:)\ (f\ x)\ (mapM\ f\ xs) \\
&mapM\_\ f\ [\,] = return\ () \\
&mapM\_\ f\ (x : xs) = f\ x \gg mapM\ f\ xs
\end{aligned}
$$

Sequencing monadic actions

$$\begin{aligned}
&\mathit{sequence} :: (\mathit{Monad}\ m) \Rightarrow [m\ a] \to m\ [a] \\
&\mathit{sequence_} :: (\mathit{Monad}\ m) \Rightarrow [m\ a] \to m\ () \\
&\mathit{sequence} = \mathit{foldr}\ (\mathit{liftM2}\ (:))\ (\mathit{return}\ [\,]) \\
&\mathit{sequence_} = \mathit{foldr}\ (\gg)\ (\mathit{return}\ ())
\end{aligned}$$

Monadic fold

$$
\begin{aligned}
&\mathit{foldM} :: (\mathit{Monad}\ m) \Rightarrow (a \to b \to m\ a) \to a \to [b] \to m\ a \\
&\mathit{foldM}\ \mathit{op}\ e\ [\,] = \mathit{return}\ e \\
&\mathit{foldM}\ \mathit{op}\ e\ (x : \mathit{xs}) = \mathbf{do}\ r \leftarrow \mathit{op}\ e\ x \\
&\mathit{foldM}\ f\ r\ \mathit{xs}
\end{aligned}
$$

More monadic operations

Browse *Control.Monad* :

$$filterM :: (Monad\ m) \Rightarrow (a \to m\ Bool) \to [a] \to m\ [a]$$
$$replicateM :: (Monad\ m) \Rightarrow Int \to m\ a \to m\ [a]$$
$$replicateM :: (Monad\ m) \Rightarrow Int \to m\ a \to m\ ()$$
$$join :: (Monad\ m) \Rightarrow m\ (m\ a) \to m\ a$$
$$when :: (Monad\ m) \Rightarrow Bool \to m\ () \to m\ ()$$
$$unless :: (Monad\ m) \Rightarrow Bool \to m\ () \to m\ ()$$
$$forever :: (Monad\ m) \Rightarrow m\ a \to m\ ()$$

. . . and more!

Exercises Wednesday

- Will be relevant for the exam project.
- *minimax* and *minimaxAlphabeta* will be discussed.
- Different tree algorithms on "Game trees"
- The exercises next week will be on Real World Haskell
- The labs on friday will be about monad problems, but you can also work on your project.