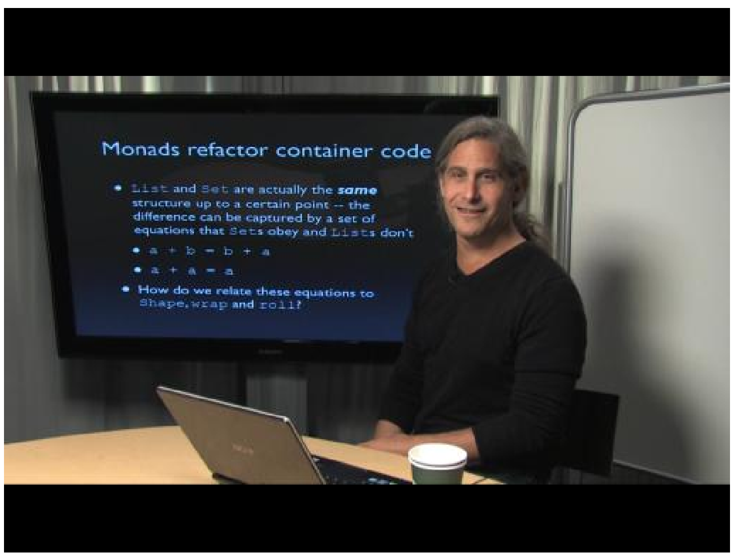
The First Monad Tutorial

Philip Wadler
University of Edinburgh

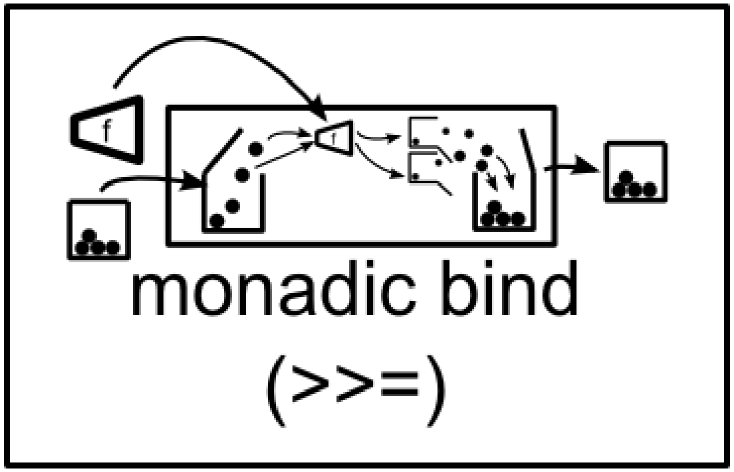
YOW! Melbourne, Brisbane, Sydney
December 2013

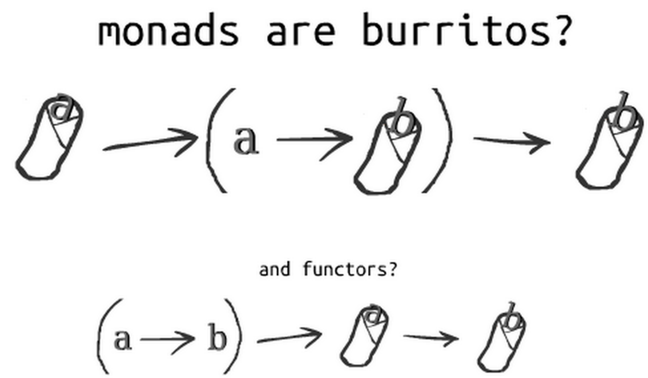

SCALA MONADS
DECLUTTER YOUR CODE WITH MONADIC DESIGN
SC
Dan Rosen, Marakana, Inc.
San Francicso, CA - February 09, 2012

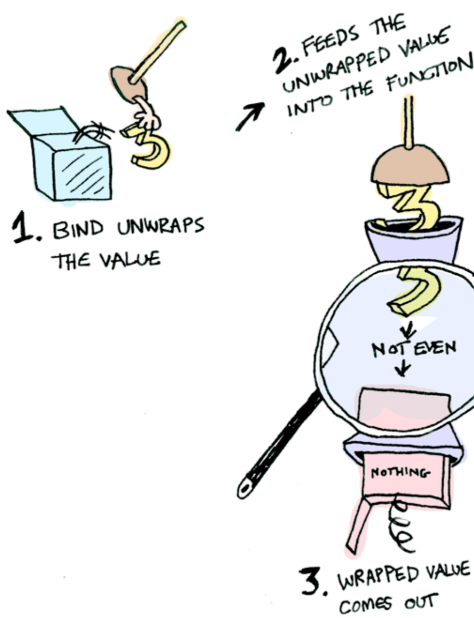

Monads in Perl
本間　雅洋
hiratara <hira.tara@gmail.com>

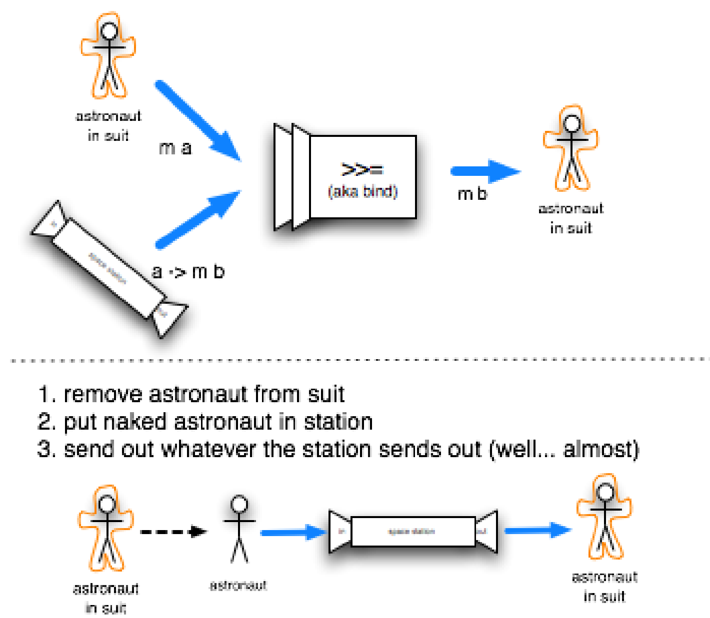

Monads refactor container code
List and Set are actually the same structure up to a certain point -- the difference can be captured by a set of equations that Sets obey and Lists don't
a + b = b + a
a + a = a
How do we relate these equations to Shape,wrap and roll?


Yet another presentation on
MONADS
Marcin Gryszko
@mgryszko


f
f
monadic bind
(>>=)


monads are burritos?
a
and functors?
a
b


1. BIND UNWRAPS THE VALUE
2. FEEDS THE UNWRAPPED VALUE INTO THE FUNCTION
NOT EVEN
NOTHING
3. WRAPPED VALUE COMES OUT


astronaut in suit
m a
>>= (aka bind)
m b
astronaut in suit
a -> m b
1. remove astronaut from suit
2. put naked astronaut in station
3. send out whatever the station sends out (well... almost)
astronaut in suit
astronaut
astronaut in suit

Haskell

Clojure

Microsoft Visual F#

C# Microsoft .net

Scala

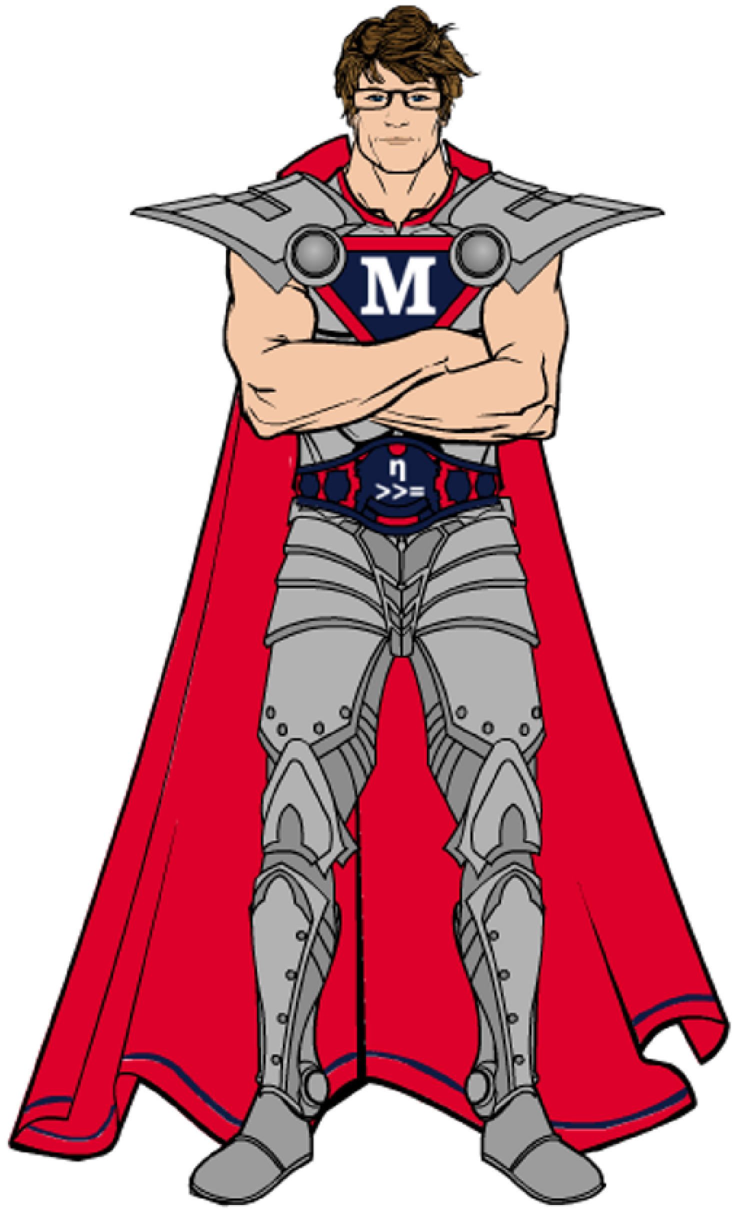
python

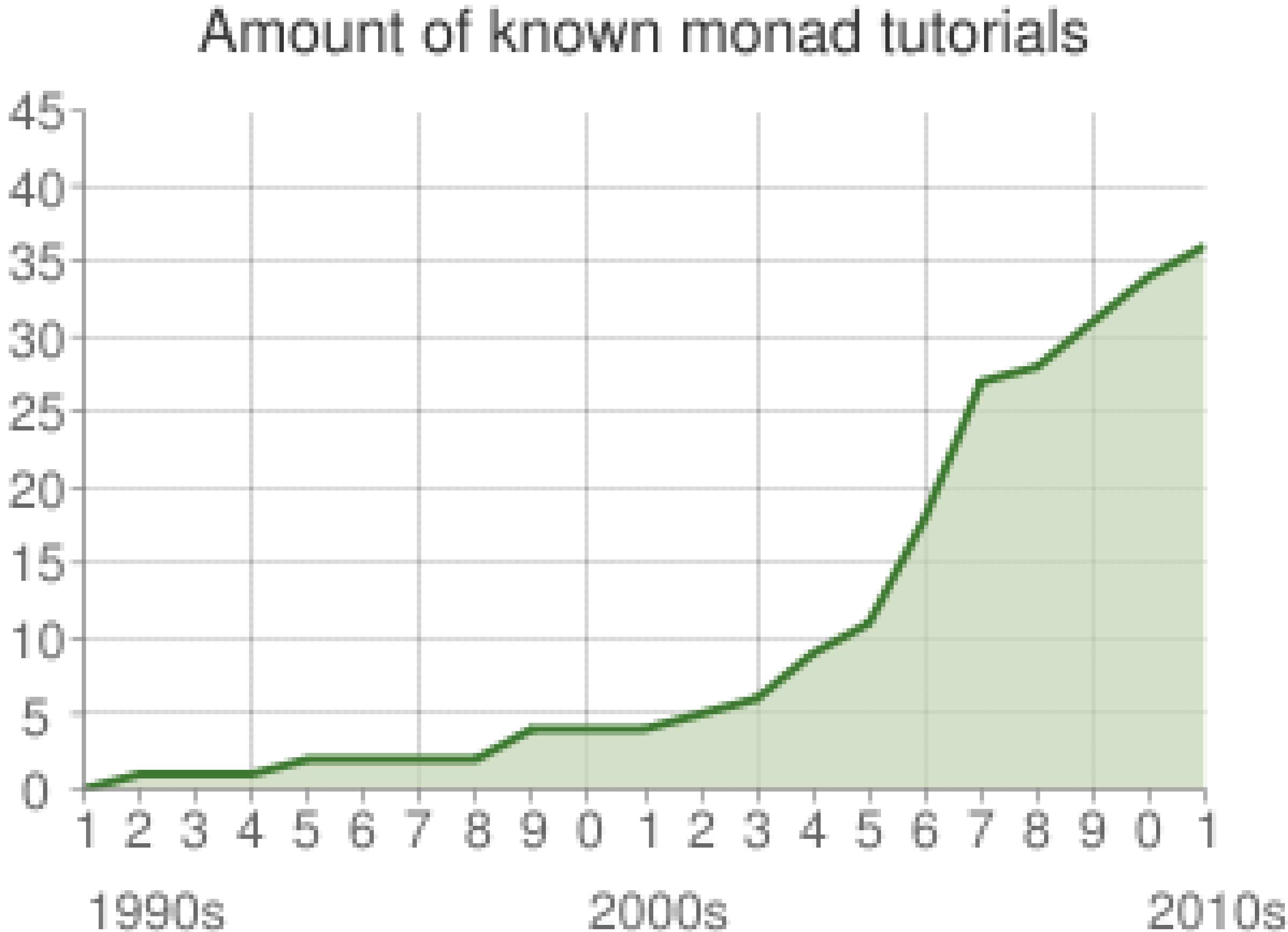

trapd in Monad tutorl
plz help


MONADS
MONADS EVERYWHERE
makeameme.org


trapd in IO monad
plz help


BRACE YOURSELVES
THE MONAD TUTORIALS ARE
COMING


M
η
>>=


Amount of known monad tutorials
45
40
35
30
25
20
15
10
5
0
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
1990s
2000s
2010s

Church and State 1: Evaluating monads

Philip Wadler

~~University of Glasgow~~
Bell Labs, Lucent Technologies

Pure vs. impure

Modification	Impure language (Standard ML, Scheme)	Pure language (Miranda*, Haskell†)
Error messages	use exceptions	rewrite
Execution counts	use state	rewrite
Output	use output	rewrite
Backwards output	rewrite	modify output

*Miranda is a trademark of Research Software Limited.

†Haskell is not a trademark.

Variations on an evaluator

monad *n.* **1.** *Philosophy* **a.** any fundamental entity, esp. if autonomous.

– *Collins English Dictionary*

Variation zero: The basic evaluator

$$
\begin{array}{lcl}
\textbf{data}\ \ \mathit{Term} & = & \mathit{Con}\,\mathit{Int} \mid \mathit{Div}\ \mathit{Term}\ \mathit{Term} \\
\\
\mathit{eval} & :: & \mathit{Term} \to \mathit{Int} \\
\mathit{eval}\,(\mathit{Con}\,a) & = & a \\
\mathit{eval}\,(\mathit{Div}\ t\ u) & = & \mathit{eval}\,t \div \mathit{eval}\,u
\end{array}
$$

Using the evaluator

Test data:

$$
\begin{array}{lcl}
\mathit{answer}, \mathit{error} & :: & \mathit{Term} \\
\mathit{answer} & = & (\mathit{Div}\,(\mathit{Div}\,(\mathit{Con}\,1972)\,(\mathit{Con}\,2))\,(\mathit{Con}\,23)) \\
\mathit{error} & = & (\mathit{Div}\,(\mathit{Con}\,1)\,(\mathit{Con}\,0))
\end{array}
$$

A sample run:

? *eval answer*
42

? *eval error*
< *BOOM!* >

Variation one: Exceptions

$$
\begin{array}{lll}
\textbf{data}\ \ M\,a & = & \mathit{Raise}\ \mathit{Exception} \mid \mathit{Return}\ a \\
\textbf{type}\ \ \mathit{Exception} & = & \mathit{String} \\
\\
\mathit{eval} & :: & \mathit{Term} \rightarrow M\ \mathit{Int} \\
\mathit{eval}\,(\mathit{Con}\ a) & = & \mathit{Return}\ a \\
\mathit{eval}\,(\mathit{Div}\ t\ u) & = & \textbf{case}\ \ \mathit{eval}\ t\ \ \textbf{of} \\
 & & \quad \mathit{Raise}\ e \rightarrow \mathit{Raise}\ e \\
 & & \quad \mathit{Return}\ a \rightarrow \\
 & & \qquad \textbf{case}\ \ \mathit{eval}\ u\ \ \textbf{of} \\
 & & \qquad\quad \mathit{Raise}\ e \rightarrow \mathit{Raise}\ e \\
 & & \qquad\quad \mathit{Return}\ b \rightarrow \\
 & & \qquad\qquad \textbf{if}\ \ b == 0 \\
 & & \qquad\qquad\quad \textbf{then}\ \ \mathit{Raise}\ \text{“divide by zero”} \\
 & & \qquad\qquad\quad \textbf{else}\ \ \ \mathit{Return}\,(a \div b)
\end{array}
$$

Using the evaluator

A sample run:

?eval answer
Return 42

?eval error
Raise "divide by zero"

Variation two: State

$$
\begin{array}{lcl}
\textbf{type}\ \ M\,a & = & State \rightarrow (a, State) \\
\textbf{type}\ \ State & = & Int \\[1ex]
eval & :: & Term \rightarrow M\,Int \\
eval\,(Con\,a)\,x & = & (a, x) \\
eval\,(Div\,t\,u)\,x & = & \textbf{let}\ \ (a, y) = eval\,t\,x\ \ \textbf{in} \\
 & & \textbf{let}\ \ (b, z) = eval\,u\,y\ \ \textbf{in} \\
 & & (a \div b, z + 1)
\end{array}
$$

Using the evaluator

Test data:

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

A sample run:

$$? \mathit{eval}\ \mathit{answer}\ 0$$
$$(42, 2)$$

Variation three: Output

$$
\begin{array}{lcl}
\textbf{type}\ \ M\,a & = & (Output, a) \\
\textbf{type}\ \ Output & = & String \\[1ex]
eval & :: & Term \to M\,Int \\
eval\,(Con\,a) & = & (line\,(Con\,a)\,a, a) \\
eval\,(Div\,t\,u) & = & \textbf{let}\ \ (x, a) = eval\,t\ \ \textbf{in} \\
 & & \textbf{let}\ \ (y, b) = eval\,u\ \ \textbf{in} \\
 & & (x +\!\!+\, y +\!\!+\, line\,(Div\,t\,u)\,(a \div b), a \div b) \\[1ex]
line & :: & Term \to Int \to Output \\
line\,t\,a & = & showterm\,t +\!\!+ \text{“} = \text{”} +\!\!+ showint\,a +\!\!+ \text{“}\hookleftarrow\text{”}
\end{array}
$$

Using the evaluator

A sample run:

$?$ *eval answer*
$(\text{``}\mathit{Con}\ 1972 = 1972\hookleftarrow$
$\mathit{Con}\ 2 = 2\hookleftarrow$
$\mathit{Div}\ (\mathit{Con}\ 1972)\ (\mathit{Con}\ 2) = 986\hookleftarrow$
$\mathit{Con}\ 23 = 23\hookleftarrow$
$\mathit{Div}\ (\mathit{Div}\ (\mathit{Con}\ 1972)\ (\mathit{Con}\ 2))\ (\mathit{Con}\ 23)) = 42\hookleftarrow$
$\text{''}, 42)$

Changing output order

Replace

$$x +\!\!+\ y +\!\!+\ line\,(Div\ t\ u)\,(a \div b)$$

with

$$line\,(Div\ t\ u)\,(a \div b) +\!\!+\ y +\!\!+\ x.$$

Modified sample run:

$?eval\ answer$
$(\text{"}Div\,(Div\,(Con\,1972)\,(Con\,2))\,(Con\,23)) = 42\hookleftarrow$
$Con\,23 = 23\hookleftarrow$
$Div\,(Con\,1972)\,(Con\,2) = 986\hookleftarrow$
$Con\,2 = 2\hookleftarrow$
$Con\,1972 = 1972\hookleftarrow$
$\text{"}, 42)$

Monads

monad *n.* **1. b.** (in the metaphysics of Leibnitz) a simple indestructible nonspatial element regarded as the unit of which reality consists.

– *Collins English Dictionary*

What is a monad?

1. For each type a of *values*, a type $M\,a$ to represent *computations*.

   In general, $a \to b$ becomes $a \to M\,b$.

   In particular, $eval :: Term \to Int$ becomes $Term \to M\,Int$.

2. A way to turn values into computations.

$$unit \;::\; a \to M\,a$$

3. A way to combine computations.

$$(\star) \;::\; M\,a \to (a \to M\,b) \to M\,b$$

Compare $\star$ with **let**

$$\frac{\Gamma \vdash m :: M\,a \qquad \Gamma, x :: a \vdash n :: M\,b}{\Gamma \vdash (m \star \lambda x.\, n) :: M\,b}$$

$$\frac{\Gamma \vdash m :: a \qquad \Gamma, x :: a \vdash n :: b}{\Gamma \vdash (\mathbf{let}\ \ x = m\ \ \mathbf{in}\ \ n) :: b}$$

Monad laws

Left unit

$$unit\, x \star \lambda y.\, n = n[x/y]$$

Right unit

$$m \star \lambda x.\, unit\, x = m$$

Associative (when x is not free in o).

$$m \star (\lambda x.\, n \star \lambda y.\, o) = (m \star \lambda x.\, n) \star \lambda y.\, o$$

These resemble the laws for a *monoid*, except for λ binding.

Monad Laws

$$
\begin{aligned}
\mathit{return}\ v \gg\!= \lambda x.\, k\ x &\quad=\quad k\ v \\
m \gg\!= \lambda x.\, \mathit{return}\ x &\quad=\quad m \\
m \gg\!= (\lambda x.\, k\ x \gg\!= (\lambda y.\, h\ y)) &\quad=\quad (m \gg\!= (\lambda x.\, k\ x)) \gg\!= (\lambda y.\, h\ y)
\end{aligned}
$$

Do notation

$$
\begin{pmatrix} \mathsf{do}\ x \leftarrow \mathit{return}\ v \\ \quad k\ x \end{pmatrix} \quad=\quad k\ v
$$

$$
\begin{pmatrix} \mathsf{do}\ x \leftarrow m \\ \quad \mathit{return}\ x \end{pmatrix} \quad=\quad m
$$

$$
\begin{pmatrix} \mathsf{do}\ y \leftarrow \mathsf{do}\ x \leftarrow m \\ \qquad\qquad\qquad k\ x \\ \quad h\ y \end{pmatrix} \quad=\quad \begin{pmatrix} \mathsf{do}\ x \leftarrow m \\ \quad y \leftarrow k\ x \\ \quad h\ y \end{pmatrix}
$$

The evaluator revisited

monad *n.* **1. c.** (in the pantheistic philosophy of Giordano Bruno) a fundamental metaphysical unit that is spatially extended and psychically aware.

– *Collins English Dictionary*

Monadic evaluator

$$
\begin{array}{lcl}
eval & :: & Term \rightarrow M\ Int \\
eval\ (Con\ a) & = & unit\ a \\
eval\ (Div\ t\ u) & = & eval\ t \star \lambda a.\ eval\ u \star \lambda b.\ unit\ (a \div b)
\end{array}
$$

Variation zero, revisited: The basic evaluator

$$
\begin{array}{lcl}
\textbf{type}\ \ M\,a & = & a \\[1ex]
\mathit{unit} & :: & a \to M\,a \\
\mathit{unit}\ a & = & a \\[1ex]
(\star) & :: & M\,a \to (a \to M\,b) \to M\,b \\
a \star k & = & k\,a
\end{array}
$$

Variation one, revisited: Exceptions

$$
\begin{array}{lcl}
\textbf{data}\ \ M\,a & = & \mathit{Raise}\ \mathit{Exception} \mid \mathit{Return}\ a \\
\textbf{type}\ \ \mathit{Exception} & = & \mathit{String} \\
\\
\mathit{unit} & :: & a \to M\,a \\
\mathit{unit}\ a & = & \mathit{Return}\ a \\
\\
(\star) & :: & M\,a \to (a \to M\,b) \to M\,b \\
m \star k & = & \textbf{case}\ \ m\ \ \textbf{of} \\
 & & \quad \mathit{Raise}\ e \to \mathit{Raise}\ e \\
 & & \quad \mathit{Return}\ a \to k\ a \\
\\
\mathit{raise} & :: & \mathit{Exception} \to M\,a \\
\mathit{raise}\ e & = & \mathit{Raise}\ e
\end{array}
$$

Modifying the evaluator

$$
\begin{array}{lcl}
\mathit{eval} & :: & \mathit{Term} \rightarrow M\ \mathit{Int} \\
\mathit{eval}\,(\mathit{Con}\ a) & = & \mathit{unit}\ a \\
\mathit{eval}\,(\mathit{Div}\ t\ u) & = & \mathit{eval}\ t \star \lambda a. \\
 & & \mathit{eval}\ u \star \lambda b. \\
 & & \textbf{if}\ \ b == 0 \\
 & & \quad \textbf{then}\ \ \mathit{raise}\ \text{“}\mathit{divide\ by\ zero}\text{”} \\
 & & \quad \textbf{else}\ \ \ \mathit{unit}\,(a \div b)
\end{array}
$$

Variation two, revisited: State

$$
\begin{array}{lcl}
\textbf{type}\ \ M\,a & = & State \rightarrow (a, State) \\
\textbf{type}\ \ State & = & Int \\[1ex]
unit & :: & a \rightarrow M\,a \\
unit\ a & = & \lambda x.\,(a, x) \\[1ex]
(\star) & :: & M\,a \rightarrow (a \rightarrow M\,b) \rightarrow M\,b \\
m \star k & = & \lambda x.\,\textbf{let}\ \ (a, y) = m\,x\ \ \textbf{in} \\
 & & \textbf{let}\ \ (b, z) = k\,a\,y\ \ \textbf{in} \\
 & & (b, z) \\[1ex]
tick & :: & M\,() \\
tick & = & \lambda x.\,((), x + 1)
\end{array}
$$

Modifying the evaluator

$$
\begin{array}{lcl}
\mathit{eval} & :: & \mathit{Term} \rightarrow M\ \mathit{Int} \\
\mathit{eval}\,(\mathit{Con}\ a) & = & \mathit{unit}\ a \\
\mathit{eval}\,(\mathit{Div}\ t\ u) & = & \mathit{eval}\ t \star \lambda a. \\
 & & \mathit{eval}\ u \star \lambda b. \\
 & & \mathit{tick}\quad \star \lambda (). \\
 & & \mathit{unit}\,(a \div b)
\end{array}
$$

Variation three, revisited: Output

$$
\begin{array}{lll}
\textbf{type}\ \ M\,a & = & (\mathit{Output}, a) \\
\textbf{type}\ \ \mathit{Output} & = & \mathit{String} \\
\\
\mathit{unit} & :: & a \to M\,a \\
\mathit{unit}\,a & = & (\text{“”}, a) \\
\\
(\star) & :: & M\,a \to (a \to M\,b) \to M\,b \\
m \star k & = & \textbf{let}\ \ (x, a) = m\ \ \textbf{in} \\
 & & \textbf{let}\ \ (y, b) = k\,a\ \ \textbf{in} \\
 & & (x +\!\!+\, y, b) \\
\\
\mathit{out} & :: & \mathit{Output} \to M\,() \\
\mathit{out}\,x & = & (x, ())
\end{array}
$$

Modifying the evaluator

$$
\begin{array}{lcll}
eval & :: & Term \rightarrow M\ Int & \\
eval\ (Con\ a) & = & out\ (line\ (Con\ a)\ a) & \star\lambda().\\
 & & unit\ a & \\
eval\ (Div\ t\ u) & = & eval\ t & \star\lambda a.\\
 & & eval\ u & \star\lambda b.\\
 & & out\ (line\ (Div\ t\ u)\ (a \div b)) & \star\lambda().\\
 & & unit\ (a \div b) &
\end{array}
$$

Changing output order

$$
\begin{array}{lcl}
(\star) & :: & M\,a \to (a \to M\,b) \to M\,b \\
m \star k & = & \textbf{let}\ \ (x, a) = m\ \ \textbf{in} \\
 & & \textbf{let}\ \ (y, b) = k\,a\ \ \textbf{in} \\
 & & (y +\!\!+\, x, b)
\end{array}
$$

Conclusions

monad *n.* **2.** a single-celled organism, especially a flagellate protozoan

– Collins English Dictionary

The Glasgow Haskell compiler

Joint work with

Cordy Hall, Kevin Hammond,
Will Partain, Simon Peyton Jones.

Glasgow Haskell compiler is written in Haskell.

Each phase uses a monad.

Has proved easy to modify in practice.

Monads in the Glasgow Haskell compiler

Type inference phase.

- Exceptions for errors,
- state for current substitution,
- state for fresh variable names,
- read-only state for current location.

Simplification phase.

- State for fresh variable names.

Code generator phase.

- Output for code generated so far,
- state for table mapping variables to addressing modes,
- state for table to cache known state of stack.

Origins

Eugenio Moggi, *Computational λ-calculus and monads*, 1989.

values (*int*) vs. computations (T *int*)
call-by-value ($int \to T\ int$)
call-by-name ($T\ int \to T\ int$)

Michael Spivey, *A functional theory of exceptions*, 1990.

John Reynolds, *The essence of Algol*, 1981.

data types (*int*) vs. phrase types (*int exp*)
call-by-value ($int \to int\ exp$)
call-by-name ($int\ exp \to int\ exp$)

But Reynolds missed *unit* and $\star$.

monadism *or* **monadology** *n.* (esp. in writings of Leibnitz) the philosophical doctrine that monads are the ultimate units of reality.

– Collins English Dictionary

Monads

$$
\begin{array}{llcl}
(1) & \mathit{return}\ v \gg\!= \lambda x.\, k\ x & = & k\ v \\
(2) & m \gg\!= \lambda x.\, \mathit{return}\ x & = & m \\
(3) & m \gg\!= (\lambda x.\, k\ x \gg\!= (\lambda y.\, h\ y)) & = & (m \gg\!= (\lambda x.\, k\ x)) \gg\!= (\lambda y.\, h\ y)
\end{array}
$$

- Eugenio Moggi, Computational Lambda Calculus and Monads, *Logic in Computer Science*, 1989.
- Philip Wadler, Comprehending Monads, *International Conference on Functional Programming*, 1990.
- Philip Wadler, The Essence of Functional Programming, *Principles of Programming Languages*, 1992.

Arrows

$$\begin{array}{lrcl}
(1) & \mathit{arr}\ \mathit{id} \ggg f & = & f \\
(2) & f \ggg \mathit{arr}\ \mathit{id} & = & f \\
(3) & (f \ggg g) \ggg h & = & f \ggg (g \ggg h) \\
(4) & \mathit{arr}\ (g \cdot f) & = & \mathit{arr}\ f \ggg \mathit{arr}\ g \\
(5) & \mathit{first}\ (\mathit{arr}\ f) & = & \mathit{arr}\ (f \times \mathit{id}) \\
(6) & \mathit{first}\ (f \ggg g) & = & \mathit{first}\ f \ggg \mathit{first}\ g \\
(7) & \mathit{first}\ f \ggg \mathit{arr}\ (\mathit{id} \times g) & = & \mathit{arr}\ (\mathit{id} \times g) \ggg \mathit{first}\ f \\
(8) & \mathit{first}\ f \ggg \mathit{arr}\ \mathit{fst} & = & \mathit{arr}\ \mathit{fst} \ggg f \\
(9) & \mathit{first}\ (\mathit{first}\ f) \ggg \mathit{arr}\ \mathit{assoc} & = & \mathit{arr}\ \mathit{assoc} \ggg \mathit{first}\ f
\end{array}$$

- John Hughes, Generalising Monads to Arrows, *Science of Computer Programming*, 2000.

Idioms (Applicative Functors)

$$
\begin{array}{lrcl}
(1) & u & = & \mathit{pure}\ \mathit{id} \otimes u \\
(2) & \mathit{pure}\ f \otimes \mathit{pure}\ p & = & \mathit{pure}\ (f\ p) \\
(3) & u \otimes (v \otimes w) & = & \mathit{pure}\ (\cdot) \otimes u \otimes v \otimes w \\
(4) & u \otimes \mathit{pure}\ x & = & \mathit{pure}\ (\lambda f.\ f\ x) \otimes u
\end{array}
$$

- Conor McBride and Ross Patterson, Applicative Programming with Effects, *Journal of Functional Programming*, 2008.