Retrofitting Purity with Comonads

Neel Krishnaswami

June 25, 2018

University of Cambridge

Once Upon a Time

Once Upon a Time

- There was a PhD student

Once Upon a Time

- There was a PhD student
- who finished her dissertation…

Once Upon a Time

Once Upon a Time

- Her advisor said, “It’s time for you to go out into the wide world!”

Once Upon a Time

- Her advisor said, “It’s time for you to go out into the wide world!”
- So she did, and she designed a programming language

```
data List a = [] | a :: (List a)
```

```
data List a = [] | a :: (List a)

len : List a -> Integer
len []        = 0
len (x :: xs) = 1 + len xs
```

A Functional Language

```
data List a = [] | a :: (List a)

len : List a -> Integer
len []        = 0
len (x :: xs) = 1 + len xs

map : (a -> b) -> List a -> List b
map f []        = []
map f (x :: xs) = f x :: map f xs
```

Once Upon a Time

Once Upon a Time

- While implementing it, she added one primitive:

Once Upon a Time

- While implementing it, she added one primitive:

```
print : String -> Unit
print = Runtime.Primitive.Magic.__printf
```

- While implementing it, she added one primitive:

```
print : String -> Unit
print = Runtime.Primitive.Magic.__printf
```

- Nothing bad happened...

Once Upon a Time

- While implementing it, she added one primitive:

```
print : String -> Unit
print = Runtime.Primitive.Magic.__printf
```

- Nothing bad happened...yet!

Once Upon a Time

Once Upon a Time

- Naturally, this language was wildly successful

Once Upon a Time

- Naturally, this language was wildly successful
- Our protagonist achieved fame and fortune

Once Upon a Time

- Naturally, this language was wildly successful
- Our protagonist achieved fame and fortune
- …and feature requests and bug reports

Feature Request: List Fusion

- A user wrote the following code:

```
map f (map g reallyBigList)
```

- and complained that it allocated a really big intermediate list

Feature Request: List Fusion

Feature Request: List Fusion

- Our protagonist wrote a compiler pass to turn this:

```
map f (map g reallyBigList)
```

Feature Request: List Fusion

- Our protagonist wrote a compiler pass to turn this:

```
map f (map g reallyBigList)
```

- into this:

```
map (f o g) reallyBigList
```

Feature Request: List Fusion

- Our protagonist wrote a compiler pass to turn this:

```
map f (map g reallyBigList)
```

- into this:

```
map (f o g) reallyBigList
```

- Much RAM was saved!

Feature Request: List Fusion

- Our protagonist wrote a compiler pass to turn this:

```
map f (map g reallyBigList)
```

- into this:

```
map (f o g) reallyBigList
```

- Much RAM was saved!
- Benchmarks improved!

Bug Reports

Bug Reports

- This code

```
f : Int -> Int
f n = print "a"; n + 1

g : Int -> Int
g n = print "b"; n + 1

printList (map f (map g [1, 2, 3]))
```

Bug Reports

- This code

```
f : Int -> Int
f n = print "a"; n + 1

g : Int -> Int
g n = print "b"; n + 1

printList (map f (map g [1, 2, 3]))
```

- In the old version, it printed:

```
bbbaaa[3, 4, 5]
```

Bug Reports

- This code

```
f : Int -> Int
f n = print "a"; n + 1

g : Int -> Int
g n = print "b"; n + 1

printList (map f (map g [1, 2, 3]))
```

- In the old version, it printed:

```
bbbaaa[3, 4, 5]
```

- In the “optimized” version, it printed:

```
bababa[3, 4, 5]
```

Narrative Tension!

Narrative Tension!

- Our protagonist was worried:

Narrative Tension!

- Our protagonist was worried:
- She wanted purity for optimization purposes

Narrative Tension!

- Our protagonist was worried:
- She wanted purity for optimization purposes
- But her language was already impure

Narrative Tension!

- Our protagonist was worried:
- She wanted purity for optimization purposes
- But her language was already impure
- Was she out of luck?

Syntax

$$
\begin{array}{llll}
\text{Types} & A & ::= & \mathsf{File} \mid \mathsf{char} \mid A \to B \\
\text{Terms} & e & ::= & x \mid c \mid e.\mathsf{print}(e') \mid \lambda x.e \mid e\,e' \\
\\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A \\
\\
\text{Judgements} & & & \Gamma \vdash e : A
\end{array}
$$

Syntax

$$\begin{array}{llcl} \text{Types} & A & ::= & \mathsf{File} \mid \mathsf{char} \mid A \to B \mid \mathsf{Pure}\,A \\ \text{Terms} & e & ::= & x \mid c \mid e.\mathsf{print}(e') \mid \lambda x.e \mid e\,e' \\ & & \mid & \mathsf{pure}(e) \mid \mathsf{let\ pure}(x) = e \mathsf{\ in\ } e' \\ \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, x :: A \\ & & & \\ \text{Judgements} & & & \Gamma \vdash e : A \end{array}$$

Typing Rules

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash e : \mathrm{File} \qquad \Gamma \vdash e' : \mathrm{char}}{\Gamma \vdash e.\mathrm{print}(e') : 1}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B} \qquad \frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e\, e' : B}$$

Typing Rules

$$\frac{x : A \in \Gamma \lor x :: A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash e : \mathrm{File} \qquad \Gamma \vdash e' : \mathrm{char}}{\Gamma \vdash e.\mathrm{print}(e') : 1}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B} \qquad \frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e\, e' : B}$$

Typing Rules

$$\frac{x : A \in \Gamma \vee x :: A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash e : \mathrm{File} \qquad \Gamma \vdash e' : \mathrm{char}}{\Gamma \vdash e.\mathrm{print}(e') : 1}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B} \qquad \frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e\,e' : B}$$

$$\frac{\Gamma^{\mathrm{pure}} \vdash e : A}{\Gamma \vdash \mathrm{pure}(e) : \mathrm{Pure}(A)}$$

$$\begin{array}{lcl} (\cdot)^{\mathrm{pure}} & = & \cdot \\ (\Gamma, x : A)^{\mathrm{pure}} & = & \Gamma^{\mathrm{pure}} \\ (\Gamma, x :: A)^{\mathrm{pure}} & = & \Gamma^{\mathrm{pure}}, x :: A \end{array}$$

Typing Rules

$$\frac{x : A \in \Gamma \vee x :: A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash e : \mathrm{File} \qquad \Gamma \vdash e' : \mathrm{char}}{\Gamma \vdash e.\mathrm{print}(e') : 1}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B} \qquad \frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e\, e' : B}$$

$$\frac{\Gamma^{\mathrm{pure}} \vdash e : A}{\Gamma \vdash \mathrm{pure}(e) : \mathrm{Pure}(A)} \qquad \frac{\Gamma \vdash e : \mathrm{Pure}(A) \qquad \Gamma, x :: A \vdash e' : C}{\Gamma \vdash \mathrm{let\ pure}(x) = e \mathrm{\ in\ } e' : C}$$

$$\begin{array}{lcl} (\cdot)^{\mathrm{pure}} & = & \cdot \\ (\Gamma, x : A)^{\mathrm{pure}} & = & \Gamma^{\mathrm{pure}} \\ (\Gamma, x :: A)^{\mathrm{pure}} & = & \Gamma^{\mathrm{pure}}, x :: A \end{array}$$

A Pure Map Function

```
data List a = [] | a :: (List a)
```

A Pure Map Function

```
data List a = [] | a :: (List a)

map : Pure(a -> b) -> List a -> List b
map (pure f) []          = []
map (pure f) (x :: xs) = f x :: map (pure f) xs
```

Principles of Retrofitted Purity

Principles of Retrofitted Purity

- We have ordinary and pure variables

Principles of Retrofitted Purity

- We have ordinary and pure variables
- We add a type for “pure values”

Principles of Retrofitted Purity

- We have ordinary and pure variables
- We add a type for “pure values”
- Pure values can only refer to pure variables

Principles of Retrofitted Purity

- We have ordinary and pure variables
- We add a type for “pure values”
- Pure values can only refer to pure variables
- Imperative functions like `print` are bound to ordinary variables

Principles of Retrofitted Purity

- We have ordinary and pure variables
- We add a type for “pure values”
- Pure values can only refer to pure variables
- Imperative functions like `print` are bound to ordinary variables
- But does this work?

Semantics

Semantics

- Let C be a set of *capabilities*

Semantics

- Let C be a set of *capabilities*
  - In our example, C is the set of file handles

Semantics

- Let C be a set of *capabilities*
    - In our example, C is the set of file handles
- A *capability space* (X, w) is a set X and a *weight function* $w : X \to \mathcal{P}(C)$

Semantics

- Let C be a set of *capabilities*
  - In our example, C is the set of file handles
- A *capability space* (X, w) is a set X and a *weight function* $w : X \to \mathcal{P}(C)$
  - Elements of X are values

Semantics

- Let C be a set of *capabilities*
  - In our example, C is the set of file handles
- A *capability space* (X, w) is a set X and a *weight function* $w : X \to \mathcal{P}(C)$
  - Elements of X are values
  - Given a value x, the weight $w(x)$ is the set of capabilities it owns

Semantics

- Let C be a set of *capabilities*
  - In our example, C is the set of file handles
- A *capability space* (X, w) is a set X and a *weight function* $w : X \to \mathcal{P}(C)$
  - Elements of X are values
  - Given a value x, the weight $w(x)$ is the set of capabilities it owns
- Given capability spaces (X, w_X) and (Y, w_Y), a function $f : X \to Y$ is *capability-respecting* when

$$w_Y(f(x)) \subseteq w_X(x)$$

Semantics

- Let C be a set of *capabilities*
  - In our example, C is the set of file handles
- A *capability space* (X, w) is a set X and a *weight function* $w : X \to \mathcal{P}(C)$
  - Elements of X are values
  - Given a value x, the weight $w(x)$ is the set of capabilities it owns
- Given capability spaces (X, w_X) and (Y, w_Y), a function $f : X \to Y$ is *capability-respecting* when

$$w_Y(f(x)) \subseteq w_X(x)$$

- Cap is the the category of capability spaces and capability-respecting functions.

Products in Cap

Given capability spaces (X, w_X) and (Y, w_Y):

- Define $(X, w_X) \times (Y, w_Y) = (X \times Y, w_{X \times Y})$ where

$$w_{X \times Y}(x, y) = w_X(x) \cup w_Y(y)$$

Products in Cap

Given capability spaces (X, w_X) and (Y, w_Y):

- Define $(X, w_X) \times (Y, w_Y) = (X \times Y, w_{X \times Y})$ where

$$w_{X\times Y}(x, y) = w_X(x) \cup w_Y(y)$$

- Define the projections

$$\begin{array}{lcl} \mathsf{fst} & : & X \times Y \to X \\ \mathsf{fst}(x, y) & = & x \\[1em] \mathsf{snd} & : & X \times Y \to Y \\ \mathsf{snd}(x, y) & = & y \end{array}$$

Cartesian Closure of Cap

Given capability spaces (X, w_X) and (Y, w_Y):

Cartesian Closure of Cap

Given capability spaces (X, w_X) and (Y, w_Y):

- $(X, w_X) \to (Y, w_Y) = (Z, w_{X \to Y})$ where

Cartesian Closure of Cap

Given capability spaces (X, w_X) and (Y, w_Y):

- $(X, w_X) \to (Y, w_Y) = (Z, w_{X \to Y})$ where

$$Z = \{f \in X \to Y \mid \exists c \subseteq C.\ \forall x \in X.\ w_Y(f(x)) \subseteq w_X(x) \cup c\}$$

Cartesian Closure of Cap

Given capability spaces (X, w_X) and (Y, w_Y):

- $(X, w_X) \to (Y, w_Y) = (Z, w_{X\to Y})$ where

$$Z = \{f \in X \to Y \mid \exists c \subseteq C.\ \forall x \in X.\ w_Y(f(x)) \subseteq w_X(x) \cup c\}$$

$$w_{X\to Y}(f) = \min \{c \in \mathcal{P}(C) \mid \forall x \in X.\ w_Y(f(x)) \subseteq w_X(x) \cup c\}$$

Cartesian Closure of Cap

Given capability spaces (X, w_X) and (Y, w_Y):

- $(X, w_X) \to (Y, w_Y) = (Z, w_{X \to Y})$ where

$$Z = \{f \in X \to Y \mid \exists c \subseteq C.\ \forall x \in X.\ w_Y(f(x)) \subseteq w_X(x) \cup c\}$$

$$w_{X \to Y}(f) = \min \{c \in \mathcal{P}(C) \mid \forall x \in X.\ w_Y(f(x)) \subseteq w_X(x) \cup c\}$$

- Intuition: weight of a function value comes from the weight of the captured variables of its closure

A Writer Monad

We can define a monad on Cap as follows.

- $T(X, w_X) = (Z, w_Z)$ where

$$Z \triangleq X \times (C \to \mathrm{String})$$

$$w_Z(x, o) = w_X(x) \cup \{c \in C \mid o(c) \neq \text{""}\}$$

- We can define the unit $\eta_X : X \to T(X)$ as

$$\eta_X(x) = (x, \lambda c.\text{""})$$

- We can define the multiplication $\mu_X : T(T(X)) \to T(X)$ as

$$\mu_X((x, o), o') = (x, \lambda c.o'(c) \cdot o(c))$$

A Purity Comonad

- $\Box(X, w_X) = (Z, w_Z)$ where

$$Z = \{x \in X \mid w_X(x) = \emptyset\}$$

$$w_Z(x) = w_X(x) = \emptyset$$

- We can define $\epsilon_X : \Box(X) \to X$ as

$$\epsilon_X(x) = x$$

- We can define $\delta_X : \Box(X) \to \Box(\Box X)$ as

$$\delta_X(x) = x$$

Escaping the Monad!

There is a *capability-respecting* function $\pi_X : \Box(TX) \to \Box X$:

$$\pi_X(x, o) = x$$

Escaping the Monad!

There is a *capability-respecting* function $\pi_X : \Box(TX) \to \Box X$:

$$\pi_X(x, o) = x$$

This looks trivial, but recall that

$$w_{T(X)}(x, o) = w_X(x) \cup \{c \in C \mid o(c) \neq \text{""}\}$$

Escaping the Monad!

There is a *capability-respecting* function $\pi_X : \Box(TX) \to \Box X$:

$$\pi_X(x, o) = x$$

This looks trivial, but recall that

$$w_{T(X)}(x, o) = w_X(x) \cup \{c \in C \mid o(c) \neq "" \}$$

The comonadic *denial* of capability ownership lets us escape!

Interpreting Types

We can interpret our programming language using the standard call-by-value interpretation of effectful functions:

$$\begin{array}{lcl}
[\![\mathrm{File}]\!] & = & C \\
[\![\mathrm{char}]\!] & = & \{0 \ldots 255\} \\
[\![A \to B]\!] & = & [\![A]\!] \to T[\![B]\!] \\
[\![\mathrm{Pure}(A)]\!] & = & \Box[\![A]\!]
\end{array}$$

Interpreting Types

We can interpret our programming language using the standard call-by-value interpretation of effectful functions:

$$
\begin{array}{lcl}
[\![\mathrm{File}]\!] & = & C \\
[\![\mathrm{char}]\!] & = & \{0 \ldots 255\} \\
[\![A \to B]\!] & = & [\![A]\!] \to T[\![B]\!] \\
[\![\mathrm{Pure}(A)]\!] & = & \Box[\![A]\!]
\end{array}
$$

$$
\begin{array}{lcl}
[\![\cdot]\!] & = & 1 \\
[\![\Gamma, x : A]\!] & = & [\![\Gamma]\!] \times [\![A]\!] \\
[\![\Gamma, x :: A]\!] & = & [\![\Gamma]\!] \times \Box[\![A]\!]
\end{array}
$$

Interpreting Types

We can interpret our programming language using the standard call-by-value interpretation of effectful functions:

$$
\begin{array}{lcl}
[\![\mathrm{File}]\!] & = & C \\
[\![\mathrm{char}]\!] & = & \{0 \ldots 255\} \\
[\![A \to B]\!] & = & [\![A]\!] \to T[\![B]\!] \\
[\![\mathrm{Pure}(A)]\!] & = & \Box[\![A]\!] \\
\\
[\![\cdot]\!] & = & 1 \\
[\![\Gamma, x : A]\!] & = & [\![\Gamma]\!] \times [\![A]\!] \\
[\![\Gamma, x :: A]\!] & = & [\![\Gamma]\!] \times \Box[\![A]\!] \\
\\
[\![\Gamma \vdash e : A]\!] & \in & [\![\Gamma]\!] \to T[\![A]\!]
\end{array}
$$

Semantics of Terms

$$[\![\Gamma \vdash e : A]\!] \quad \in \quad [\![\Gamma]\!] \to T[\![A]\!]$$

Semantics of Terms

$$
\begin{aligned}
[\![\Gamma \vdash e : A]\!] &\in [\![\Gamma]\!] \to T[\![A]\!] \\
[\![x]\!]\ \gamma &= \mathrm{return}\ \gamma(x)
\end{aligned}
$$

Semantics of Terms

$$
\begin{array}{lcl}
[\![\Gamma \vdash e : A]\!] & \in & [\![\Gamma]\!] \to T[\![A]\!] \\
[\![x]\!]\ \gamma & = & \mathrm{return}\ \gamma(x) \\
[\![\lambda x.e]\!]\ \gamma & = & \mathrm{return}\ (\lambda v.[\![e]\!](\gamma, v/x))
\end{array}
$$

Semantics of Terms

$$
\begin{array}{lcl}
[\![\Gamma \vdash e : A]\!] & \in & [\![\Gamma]\!] \to T[\![A]\!] \\
[\![x]\!]\ \gamma & = & \mathrm{return}\ \gamma(x) \\
[\![\lambda x.e]\!]\ \gamma & = & \mathrm{return}\ (\lambda v.[\![e]\!](\gamma, v/x)) \\
[\![e_1\ e_2]\!]\ \gamma & = & \begin{array}{ll} \mathrm{do} & f \leftarrow [\![e_1]\!]\ \gamma \\ & v \leftarrow [\![e_2]\!]\ \gamma \\ & f(v) \end{array}
\end{array}
$$

Semantics of Terms

$$
\begin{array}{lcl}
[\![\Gamma \vdash e : A]\!] & \in & [\![\Gamma]\!] \to T[\![A]\!] \\
[\![x]\!]\ \gamma & = & \mathrm{return}\ \gamma(x) \\
[\![\lambda x.e]\!]\ \gamma & = & \mathrm{return}\ (\lambda v.[\![e]\!](\gamma, v/x)) \\
[\![e_1\ e_2]\!]\ \gamma & = & \begin{array}{ll} \mathrm{do} & f \leftarrow [\![e_1]\!]\ \gamma \\ & v \leftarrow [\![e_2]\!]\ \gamma \\ & f(v) \end{array} \\
[\![\mathrm{pure}(e)]\!]\ \gamma & = & \mathrm{return}\ (\pi([\![e]\!]\ \gamma^{\mathrm{Pure}}))
\end{array}
$$

Semantics of Terms

$$
\begin{array}{lcl}
[\![\Gamma \vdash e : A]\!] & \in & [\![\Gamma]\!] \to T[\![A]\!] \\
[\![x]\!]\ \gamma & = & \mathrm{return}\ \gamma(x) \\
[\![\lambda x.e]\!]\ \gamma & = & \mathrm{return}\ (\lambda v.[\![e]\!](\gamma, v/x)) \\
[\![e_1\ e_2]\!]\ \gamma & = & \begin{array}{ll} \mathrm{do} & f \leftarrow [\![e_1]\!]\ \gamma \\ & v \leftarrow [\![e_2]\!]\ \gamma \\ & f(v) \end{array} \\
[\![\mathrm{pure}(e)]\!]\ \gamma & = & \mathrm{return}\ (\pi([\![e]\!]\ \gamma^{\mathrm{Pure}})) \\
[\![\mathrm{let\ pure}(x) = e\ \mathrm{in}\ e']\!]\ \gamma & = & \begin{array}{ll} \mathrm{do} & v \leftarrow [\![e]\!]\ \gamma \\ & [\![e']\!]\ (\gamma, v/x) \end{array}
\end{array}
$$

Semantics of Terms

$$
\begin{array}{lcl}
[\![\Gamma \vdash e : A]\!] & \in & [\![\Gamma]\!] \to T[\![A]\!] \\
[\![x]\!]\ \gamma & = & \mathrm{return}\ \gamma(x) \\
[\![\lambda x.e]\!]\ \gamma & = & \mathrm{return}\ (\lambda v.[\![e]\!](\gamma, v/x)) \\
[\![e_1\ e_2]\!]\ \gamma & = & \begin{array}{ll} \mathrm{do} & f \leftarrow [\![e_1]\!]\ \gamma \\ & v \leftarrow [\![e_2]\!]\ \gamma \\ & f(v) \end{array} \\
[\![\mathrm{pure}(e)]\!]\ \gamma & = & \mathrm{return}\ (\pi([\![e]\!]\ \gamma^{\mathrm{Pure}})) \\
[\![\mathrm{let\ pure}(x) = e\ \mathrm{in}\ e']\!]\ \gamma & = & \begin{array}{ll} \mathrm{do} & v \leftarrow [\![e]\!]\ \gamma \\ & [\![e']\!]\ (\gamma, v/x) \end{array} \\
[\![e_1.\mathrm{print}(e_2)]\!]\ \gamma & = & \begin{array}{l} \mathrm{let}\ (f, o_1) = [\![e_1]\!]\ \gamma\ \mathrm{in} \\ \mathrm{let}\ (c, o_2) = [\![e_2]\!]\ \gamma\ \mathrm{in} \\ \mathrm{let}\ o_3 = \lambda n.o_2(n) \cdot o_1(n)\ \mathrm{in} \\ (*, [o_3 | f : o_3(f) \cdot c]) \end{array}
\end{array}
$$

Our heroine added comonadic purity to her programming language:

Conclusion

Our heroine added comonadic purity to her programming language:

- She had a sound semantics and a clean type theory

Conclusion

Our heroine added comonadic purity to her programming language:

- She had a sound semantics and a clean type theory
- Fusion worked for pure functions

Conclusion

Our heroine added comonadic purity to her programming language:

- She had a sound semantics and a clean type theory
- Fusion worked for pure functions
- Backwards compatibility was retained for effectful code

Conclusion

Our heroine added comonadic purity to her programming language:

- She had a sound semantics and a clean type theory
- Fusion worked for pure functions
- Backwards compatibility was retained for effectful code
- Her systems programmer friends were happy she had a capability-safe language

Conclusion

Our heroine added comonadic purity to her programming language:

- She had a sound semantics and a clean type theory
- Fusion worked for pure functions
- Backwards compatibility was retained for effectful code
- Her systems programmer friends were happy she had a capability-safe language
- And she grew up to be a dinosaur pirate witch PL designer.