

Asymptotic Improvement of Computations over Free Monads

Janis Voigtländer

Technische Universität Dresden

MPC'08

Monads for IO in Haskell

Program:

```
echo :: IO ()  
echo = do c ← getChar  
         when (c ≠ '*') $  
           do putChar c  
             echo
```

Monads for IO in Haskell

Program:

```
echo :: IO ()  
echo = do c ← getChar  
         when (c ≠ '*') $  
           do putChar c  
             echo
```

Behaviour:

```
stdin :  
stdout :
```

Monads for IO in Haskell

Program:

```
echo :: IO ()  
echo = do c ← getChar  
        when (c ≠ '*') $  
            do putChar c  
                echo
```

Behaviour:

```
stdin  : a  
stdout: a
```

Monads for IO in Haskell

Program:

```
echo :: IO ()  
echo = do c ← getChar  
        when (c ≠ '*') $  
            do putChar c  
                echo
```

Behaviour:

```
stdin  : a b  
stdout: a b
```

Monads for IO in Haskell

Program:

```
echo :: IO ()  
echo = do c ← getChar  
        when (c ≠ '*') $  
            do putChar c  
                echo
```

Behaviour:

```
stdin  : a b c  
stdout: a b c
```

Monads for IO in Haskell

Program:

```
echo :: IO ()
echo = do c ← getChar
      when (c ≠ '*') $
        do putChar c
          echo
```

Behaviour:

```
stdin  : a b c *
stdout : a b c
```

Monads for IO in Haskell

Program:

```
echo :: IO ()  
echo = do c ← getChar  
        when (c ≠ '*') $  
            do putChar c  
                echo
```

Behaviour:

```
stdin  : a b c *  
stdout: a b c
```


Testing IO Programs: IOSpec [Swierstra & Altenkirch, 07]

Program:

```
echo :: IOSpec Teletype ()  
echo = do c ← getChar  
        when (c ≠ '*') $  
            do putChar c  
            echo
```

Testing IO Programs: IOSpec [Swierstra & Altenkirch, 07]

Program:

```
echo :: IOSpec Teletype ()  
echo = do c ← getChar  
         when (c ≠ '*') $  
           do putChar c  
           echo
```

Testing:

```
> run (evalIOSpec echo singleThreaded) "abc*"
```

Testing IO Programs: IOSpec [Swierstra & Altenkirch, 07]

Program:

```
echo :: IOSpec Teletype ()  
echo = do c ← getChar  
         when (c ≠ '*') $  
           do putChar c  
           echo
```

Testing:

```
> run (evalIOSpec echo singleThreaded) "abc*"  
Read (Print 'a' (Read (Print 'b' (Read (Print 'c' (Read (Finish ())))))))))
```

Testing IO Programs: IOSpec [Swierstra & Altenkirch, 07]

Program:

```
echo :: IOSpec Teletype ()
echo = do c ← getChar
      when (c ≠ '*') $
        do putChar c
          echo
```

Testing:

```
prop cs = run (evalIOSpec echo singleThreaded) (cs ++ "*")
          ≡ copy cs
where copy (c : cs) = Read (Print c (copy cs))
      copy []       = Read (Finish ())
```

Testing IO Programs: IOSpec [Swierstra & Altenkirch, 07]

Program:

```
echo :: IOSpec Teletype ()
echo = do c ← getChar
      when (c ≠ '*') $
        do putChar c
          echo
```

Testing:

```
prop cs = run (evalIOSpec echo singleThreaded) (cs ++ "*")
          ≡ copy cs
where copy (c : cs) = Read (Print c (copy cs))
      copy []       = Read (Finish ())
```

> quickCheck prop

OK, passed 100 tests.

A Slight Variation of the Example

Program:

```
revEcho :: IO ()  
revEcho = do c ← getChar  
           when (c ≠ '*') $  
             do revEcho  
               putChar c
```

A Slight Variation of the Example

Program:

```
revEcho :: IO ()  
revEcho = do c ← getChar  
           when (c ≠ '*') $  
             do revEcho  
               putChar c
```

Behaviour:

```
stdin :  
stdout :
```

A Slight Variation of the Example

Program:

```
revEcho :: IO ()  
revEcho = do c ← getChar  
           when (c ≠ '*') $  
             do revEcho  
               putChar c
```

Behaviour:

```
stdin  : a  
stdout:
```


A Slight Variation of the Example

Program:

```
revEcho :: IO ()  
revEcho = do c ← getChar  
           when (c ≠ '*') $  
             do revEcho  
               putChar c
```

Behaviour:

```
stdin  : a b  
stdout :
```

A Slight Variation of the Example

Program:

```
revEcho :: IO ()  
revEcho = do c ← getChar  
           when (c ≠ '*') $  
             do revEcho  
               putChar c
```

Behaviour:

```
stdin  : a b c  
stdout :
```

A Slight Variation of the Example

Program:

```
revEcho :: IO ()  
revEcho = do c ← getChar  
           when (c ≠ '*') $  
             do revEcho  
               putChar c
```

Behaviour:

```
stdin  : a b c *  
stdout: c b a
```

A Slight Variation of the Example

Program:

```
revEcho :: IO ()  
revEcho = do c ← getChar  
           when (c ≠ '*') $  
             do revEcho  
               putChar c
```

Behaviour:

```
stdin  : a b c *  
stdout: c b a
```

A Slight Variation of the Example

Program:

```
revEcho :: IOSpec Teletype ()  
revEcho = do c ← getChar  
           when (c ≠ '*') $  
             do revEcho  
               putChar c
```

Testing:

```
> run (evalIOSpec revEcho singleThreaded) "abc*"
```

A Slight Variation of the Example

Program:

```
revEcho :: IOSpec Teletype ()  
revEcho = do c ← getChar  
           when (c ≠ '*') $  
             do revEcho  
               putChar c
```

Testing:

```
> run (evalIOSpec revEcho singleThreaded) "abc*"  
Read (Read (Read (Read (Print 'c' (Print 'b' (Print 'a' (Finish ())))))))))
```

A Slight Variation of the Example

Program:

```
revEcho :: IOSpec Teletype ()
revEcho = do c ← getChar
           when (c ≠ '*') $
             do revEcho
               putChar c
```

Testing:

```
prop cs = run (evalIOSpec revEcho singleThreaded) (cs ++ "*")
           ≡ mirror cs (Finish ())
where mirror (c : cs) acc = Read (mirror cs (Print c acc))
       mirror [] acc = Read acc
```

A Slight Variation of the Example

Program:

```
revEcho :: IOSpec Teletype ()
revEcho = do c ← getChar
           when (c ≠ '*') $
             do revEcho
               putChar c
```

Testing:

```
prop cs = run (evalIOSpec revEcho singleThreaded) (cs ++ "*")
           ≡ mirror cs (Finish ())
  where mirror (c : cs) acc = Read (mirror cs (Print c acc))
        mirror []       acc = Read acc
```

> quickCheck prop

OK, passed 100 tests.

A Slight Variation of the Example: Ouch!

Program:

```
revEcho :: IOSpec Teletype ()
revEcho = do c ← getChar
           when (c ≠ '*') $
             do revEcho
               putChar c
```

Testing:

```
prop cs = run (evalIOSpec revEcho singleThreaded) (cs ++ "*")
           ≡ mirror cs (Finish ())
  where mirror (c : cs) acc = Read (mirror cs (Print c acc))
        mirror []       acc = Read acc
```

> quickCheck prop
OK, passed 100 tests.

But each test takes quadratic time!

But Why?

Let's take a closer look at "IOSpec Teletype", henceforth "IO_{tt}".

But Why?

Let's take a closer look at "IOSpec Teletype", henceforth "IO_{tt}".

```
data IOtt α = GetChar (Char → IOtt α) | PutChar Char (IOtt α)  
           | Return α
```

But Why?

Let's take a closer look at "IOSpec Teletype", henceforth "IO_{tt}".

```
data IOtt α = GetChar (Char → IOtt α) | PutChar Char (IOtt α)
           | Return α
```

```
instance Monad IOtt where
```

```
...
```

But Why?

Let's take a closer look at "IOspec Teletype", henceforth "IO_{tt}".

```
data IOtt α = GetChar (Char → IOtt α) | PutChar Char (IOtt α)
           | Return α
```

```
instance Monad IOtt where
```

```
...
```

```
getChar :: IOtt Char
```

```
getChar = GetChar Return
```

```
putChar :: Char → IOtt ()
```

```
putChar c = PutChar c (Return ())
```

But Why?

Let's take a closer look at "IOspec Teletype", henceforth "IO_{tt}".

```
data IOtt α = GetChar (Char → IOtt α) | PutChar Char (IOtt α)
           | Return α
```

```
instance Monad IOtt where
```

```
...
```

```
getChar :: IOtt Char
```

```
getChar = GetChar Return
```

```
putChar :: Char → IOtt ()
```

```
putChar c = PutChar c (Return ())
```

```
run :: IOtt α → String → Output α
```

```
run (GetChar f) (c : cs) = Read (run (f c) cs)
```

```
run (PutChar c p) cs = Print c (run p cs)
```

```
run (Return a) cs = Finish a
```

But Why?

Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = GetChar *f*

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

But Why?

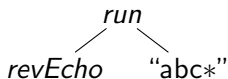
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = GetChar *f*

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps:



But Why?

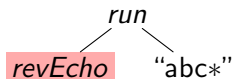
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = GetChar *f*

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps:



But Why?

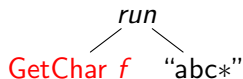
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = **GetChar** *f*

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps:



But Why?

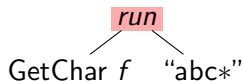
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = GetChar *f*

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps:



run (GetChar *f*) (*c* : *cs*) = Read (*run* (*f* *c*) *cs*)

But Why?

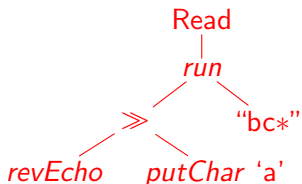
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = GetChar *f*

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps:



$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

But Why?

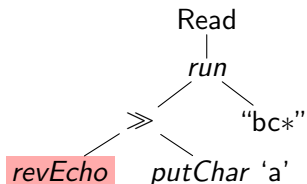
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = GetChar *f*

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps:



$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

But Why?

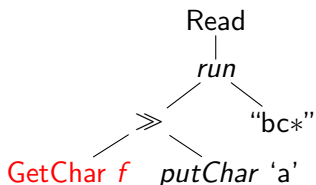
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = **GetChar** *f*

where *f* = $\lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps:



$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

But Why?

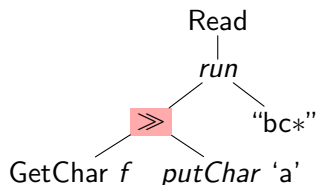
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = GetChar *f*

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps:



$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

$(\text{GetChar } f) \gg m = \text{GetChar } (\lambda c \rightarrow f \ c \gg m)$

But Why?

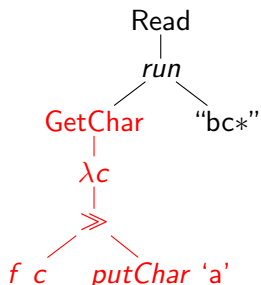
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = GetChar *f*

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps: 1



$\text{run (GetChar } f) (c : cs) = \text{Read (run (f c) cs)}$

$(\text{GetChar } f) \gg m = \text{GetChar } (\lambda c \rightarrow f\ c \gg m)$

But Why?

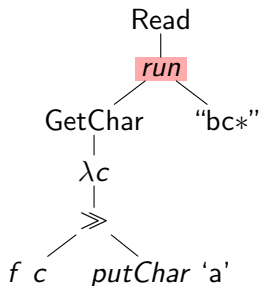
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = GetChar *f*

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps: 1



$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

$(\text{GetChar } f) \gg m = \text{GetChar } (\lambda c \rightarrow f \ c \gg m)$

But Why?

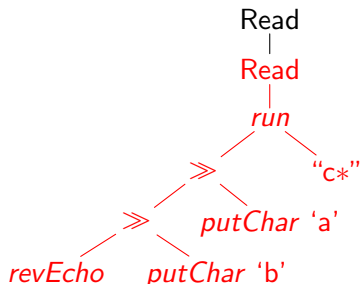
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = GetChar *f*

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps: 1



$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

$(\text{GetChar } f) \gg m = \text{GetChar } (\lambda c \rightarrow f \ c \gg m)$

But Why?

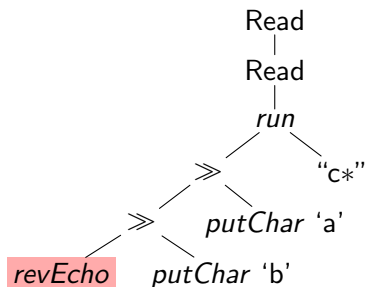
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = GetChar *f*

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps: 1



$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

$(\text{GetChar } f) \gg m = \text{GetChar } (\lambda c \rightarrow f \ c \gg m)$

But Why?

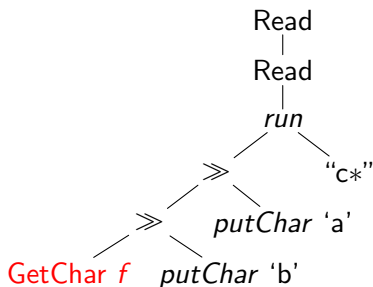
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = **GetChar** *f*

where *f* = $\lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps: 1



$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

$(\text{GetChar } f) \gg m = \text{GetChar } (\lambda c \rightarrow f \ c \gg m)$

But Why?

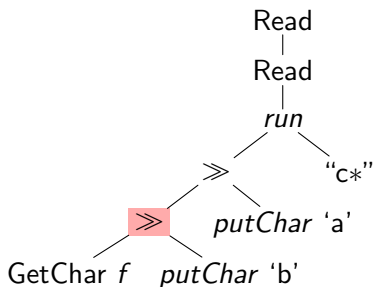
Now, *revEcho* desugared and with some inlining:

revEcho :: IO_{tt} ()

revEcho = GetChar *f*

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps: 1



$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

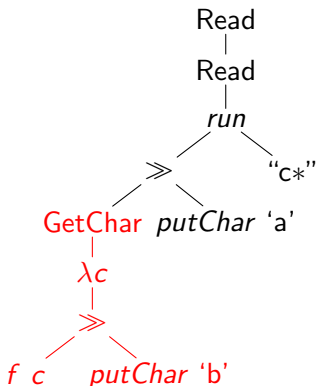
$(\text{GetChar } f) \gg m = \text{GetChar } (\lambda c \rightarrow f \ c \gg m)$

But Why?

...

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps: 1 + 1



$\text{run} (\text{GetChar } f) (c : \text{cs}) = \text{Read} (\text{run} (f \ c) \ \text{cs})$

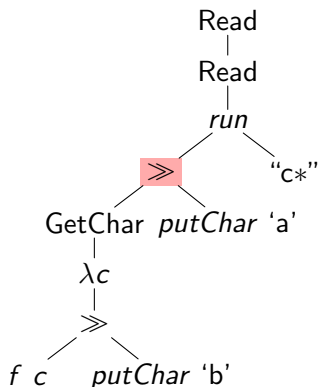
$(\text{GetChar } f) \gg m = \text{GetChar} (\lambda c \rightarrow f \ c \gg m)$

But Why?

...

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps: 1 + 1



$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f\ c) cs)$

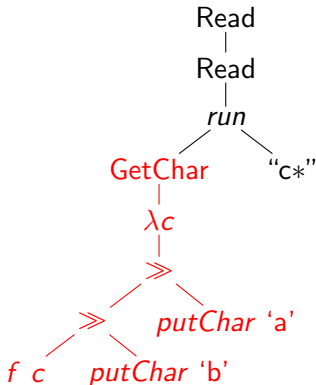
$(\text{GetChar } f) \gg m = \text{GetChar } (\lambda c \rightarrow f\ c \gg m)$

But Why?

...

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps: 1 + 2



$\text{run} (\text{GetChar } f) (c : cs) = \text{Read} (\text{run} (f \ c) \ cs)$

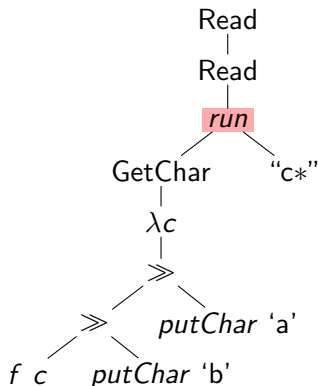
$(\text{GetChar } f) \gg m = \text{GetChar } (\lambda c \rightarrow f \ c \gg m)$

But Why?

...

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps: 1 + 2



$\text{run (GetChar } f) (c:cs) = \text{Read (run (f c) cs)}$

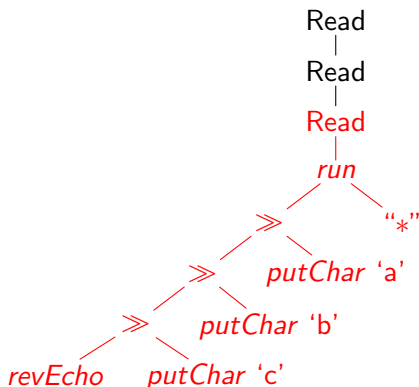
$(\text{GetChar } f) \gg m = \text{GetChar } (\lambda c \rightarrow f\ c \gg m)$

But Why?

...

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps: 1 + 2



$\text{run} (\text{GetChar } f) (c : cs) = \text{Read} (\text{run} (f \ c) \ cs)$

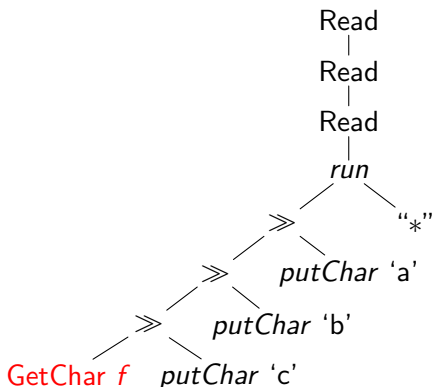
$(\text{GetChar } f) \gg m = \text{GetChar} (\lambda c \rightarrow f \ c \gg m)$

But Why?

...

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps: 1 + 2



$\text{run (GetChar } f) (c : cs) = \text{Read (run (} f \text{ } c) cs)$

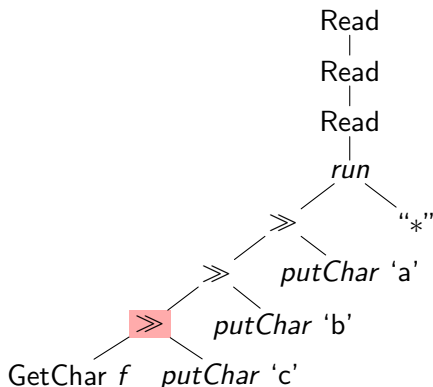
$(\text{GetChar } f) \gg m = \text{GetChar } (\lambda c \rightarrow f \text{ } c \gg m)$

But Why?

...

where $f = \lambda c \rightarrow \text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c$

An example evaluation, counting (certain) steps: 1 + 2

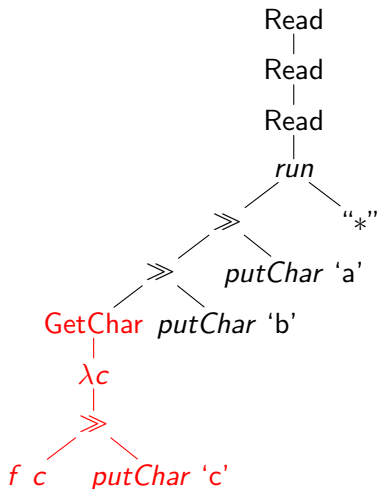


$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

$(\text{GetChar } f) \gg m = \text{GetChar } (\lambda c \rightarrow f \ c \gg m)$

But Why?

An example evaluation, counting (certain) steps: 1 + 2 + 1

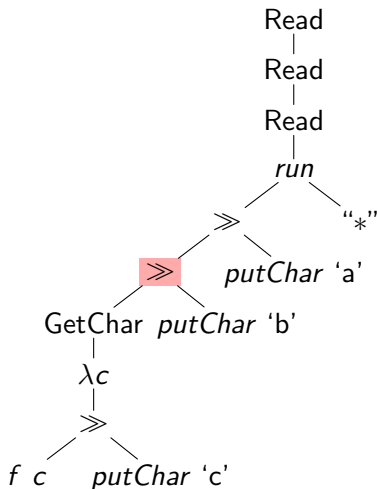


$run (GetChar f) (c : cs) = Read (run (f c) cs)$

$(GetChar f) \gg m = GetChar (\lambda c \rightarrow f c \gg m)$

But Why?

An example evaluation, counting (certain) steps: $1 + 2 + 1$

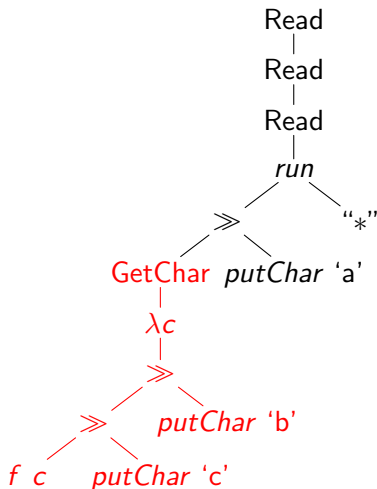


$run (GetChar f) (c : cs) = Read (run (f c) cs)$

$(GetChar f) \gg m = GetChar (\lambda c \rightarrow f c \gg m)$

But Why?

An example evaluation, counting (certain) steps: 1 + 2 + 2

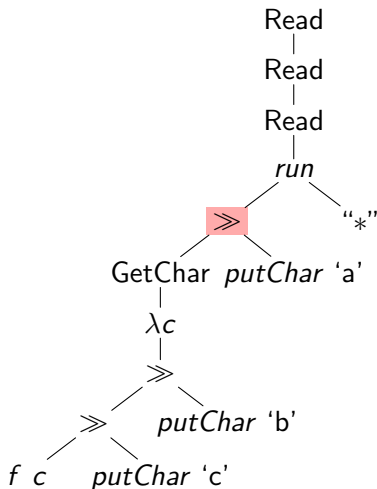


$run (GetChar f) (c : cs) = Read (run (f c) cs)$

$(GetChar f) \gg m = GetChar (\lambda c \rightarrow f c \gg m)$

But Why?

An example evaluation, counting (certain) steps: $1 + 2 + 2$

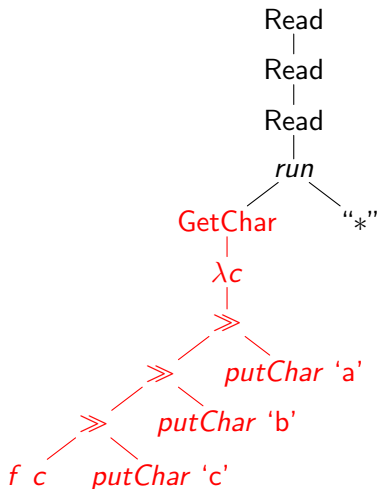


$run (GetChar f) (c : cs) = Read (run (f c) cs)$

$(GetChar f) \gg m = GetChar (\lambda c \rightarrow f c \gg m)$

But Why?

An example evaluation, counting (certain) steps: 1 + 2 + 3

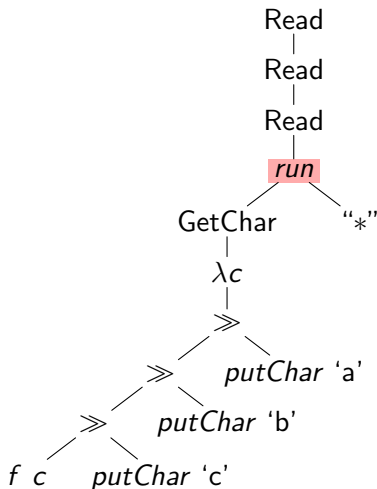


$run (GetChar f) (c : cs) = Read (run (f c) cs)$

$(GetChar f) \gg m = GetChar (\lambda c \rightarrow f c \gg m)$

But Why?

An example evaluation, counting (certain) steps: $1 + 2 + 3$

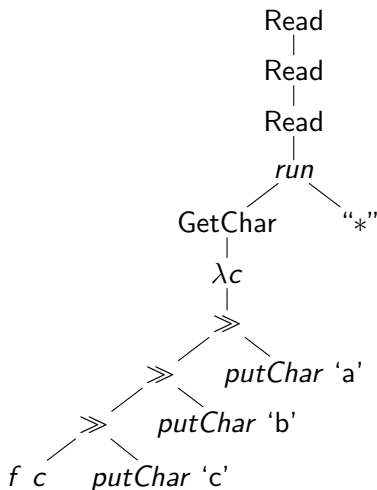


$run (GetChar f) (c : cs) = Read (run (f c) cs)$

$(GetChar f) \gg m = GetChar (\lambda c \rightarrow f c \gg m)$

But Why?

An example evaluation, counting (certain) steps: $1 + 2 + 3$



$run (GetChar f) (c : cs) = Read (run (f c) cs)$

$(GetChar f) \gg m = GetChar (\lambda c \rightarrow f c \gg m)$

What to Do?

Switch type yet again:

```
revEcho :: C ()  
revEcho = do c ← getChar  
           when (c ≠ '*') $  
             do revEcho  
               putChar c
```

What to Do?

Switch type yet again:

```
revEcho :: C ()  
revEcho = do c ← getChar  
           when (c ≠ '*') $  
             do revEcho  
               putChar c
```

type C $\alpha = \forall \beta. (\alpha \rightarrow \text{IO}_{\text{tt}} \beta) \rightarrow \text{IO}_{\text{tt}} \beta$

What to Do?

Switch type yet again:

```
revEcho :: C ()
revEcho = do c ← getChar
           when (c ≠ '*') $
             do revEcho
               putChar c
```

type C $\alpha = \forall \beta. (\alpha \rightarrow \text{IO}_{\text{tt}} \beta) \rightarrow \text{IO}_{\text{tt}} \beta$

instance Monad C **where**

...

What to Do?

Switch type yet again:

```
revEcho :: C ()  
revEcho = do c ← getChar  
           when (c ≠ '*') $  
             do revEcho  
               putChar c
```

type C $\alpha = \forall \beta. (\alpha \rightarrow \text{IO}_{\text{tt}} \beta) \rightarrow \text{IO}_{\text{tt}} \beta$

instance Monad C **where**

...

getChar :: C Char

getChar = $\lambda h \rightarrow \text{GetChar } (\lambda c \rightarrow h c)$

putChar :: Char \rightarrow C ()

putChar c = $\lambda h \rightarrow \text{PutChar } c (h ())$

And Then?

Now, *revEcho* desugared and with some inlining:

revEcho :: C ()

revEcho = $\lambda h \rightarrow \text{GetChar } f$

where $f = \lambda c \rightarrow (\text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c) \$ h$

And Then?

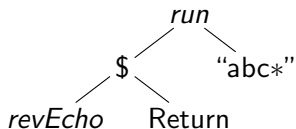
Now, *revEcho* desugared and with some inlining:

revEcho :: C ()

revEcho = $\lambda h \rightarrow \text{GetChar } f$

where $f = \lambda c \rightarrow (\text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c) \$ h$

An example evaluation:



And Then?

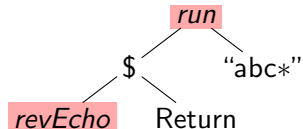
Now, *revEcho* desugared and with some inlining:

revEcho :: C ()

revEcho = $\lambda h \rightarrow$ GetChar *f*

where *f* = $\lambda c \rightarrow$ (when (*c* ≠ '*') \$ *revEcho* >> putChar *c*) \$ *h*

An example evaluation:



run (GetChar *f*) (*c* : *cs*) = Read (*run* (*f* *c*) *cs*)

And Then?

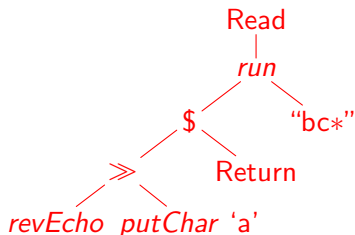
Now, *revEcho* desugared and with some inlining:

```
revEcho :: C ()
```

```
revEcho =  $\lambda h \rightarrow$  GetChar f
```

```
  where f =  $\lambda c \rightarrow$  (when ( $c \neq$  '*') $ revEcho  $\gg$  putChar c) $ h
```

An example evaluation:



```
run (GetChar f) (c : cs) = Read (run (f c) cs)
```

And Then?

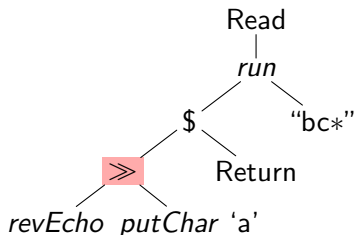
Now, *revEcho* desugared and with some inlining:

revEcho :: C ()

revEcho = $\lambda h \rightarrow \text{GetChar } f$

where $f = \lambda c \rightarrow (\text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c) \$ h$

An example evaluation:



$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

$p \gg m = \lambda h \rightarrow p \$ (\lambda_ \rightarrow m \$ h)$

And Then?

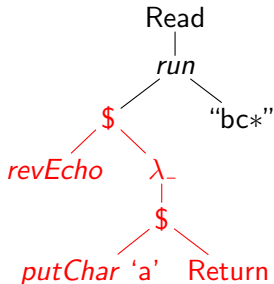
Now, *revEcho* desugared and with some inlining:

revEcho :: C ()

revEcho = $\lambda h \rightarrow \text{GetChar } f$

where $f = \lambda c \rightarrow (\text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c) \$ h$

An example evaluation:



$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

$p \gg m = \lambda h \rightarrow p \$ (\lambda_ \rightarrow m \$ h)$

And Then?

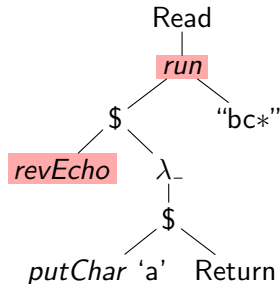
Now, *revEcho* desugared and with some inlining:

revEcho :: C ()

revEcho = $\lambda h \rightarrow$ GetChar *f*

where $f = \lambda c \rightarrow$ (when ($c \neq '*'$) \$ *revEcho* \gg putChar *c*) \$ *h*

An example evaluation:



run (GetChar *f*) (*c* : *cs*) = Read (*run* (*f* *c*) *cs*)

$p \gg m = \lambda h \rightarrow p \$ (\lambda_ \rightarrow m \$ h)$

And Then?

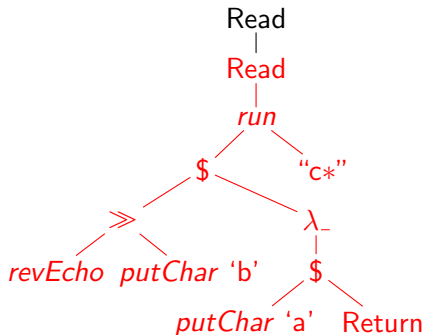
Now, *revEcho* desugared and with some inlining:

revEcho :: C ()

revEcho = $\lambda h \rightarrow \text{GetChar } f$

where $f = \lambda c \rightarrow (\text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c) \$ h$

An example evaluation:



$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

$p \gg m = \lambda h \rightarrow p \$ (\lambda_ \rightarrow m \$ h)$

And Then?

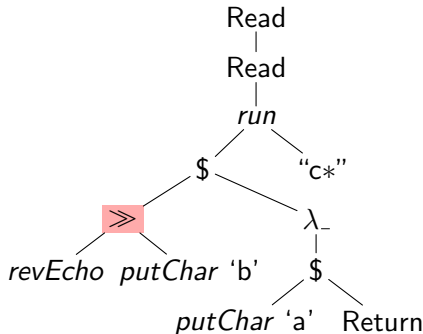
Now, *revEcho* desugared and with some inlining:

revEcho :: C ()

revEcho = $\lambda h \rightarrow \text{GetChar } f$

where $f = \lambda c \rightarrow (\text{when } (c \neq '*') \$ \text{revEcho} \gg \text{putChar } c) \$ h$

An example evaluation:

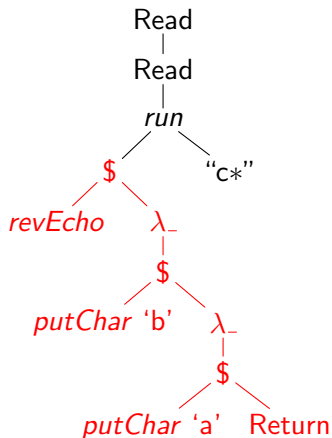


$\text{run } (\text{GetChar } f) (c : cs) = \text{Read } (\text{run } (f \ c) \ cs)$

$p \gg m = \lambda h \rightarrow p \$ (\lambda_ \rightarrow m \$ h)$

And Then?

An example evaluation:

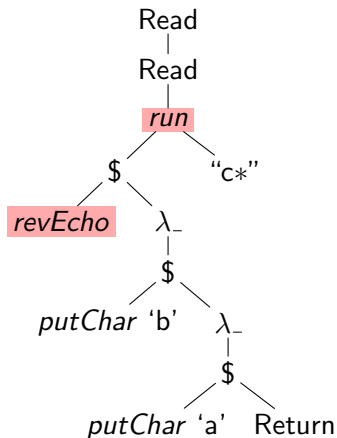


$run \text{ (GetChar } f) (c : cs) = \text{Read } (run \text{ (} f \text{ } c) \text{ } cs)$

$p \gg m = \lambda h \rightarrow p \text{ } \$ (\lambda_ \rightarrow m \text{ } \$ h)$

And Then?

An example evaluation:

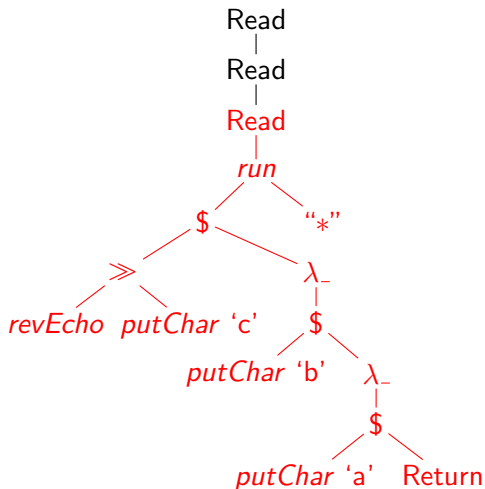


$run \text{ (GetChar } f) (c : cs) = \text{Read } (run (f \ c) \ cs)$

$p \gg m = \lambda h \rightarrow p \ \$ (\lambda_ \rightarrow m \ \$ h)$

And Then?

An example evaluation:

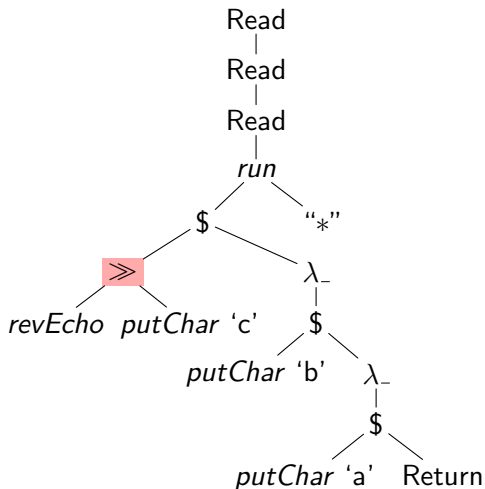


$run \text{ (GetChar } f) (c : cs) = \text{Read } (run (f \ c) \ cs)$

$p \gg m = \lambda h \rightarrow p \ \$ (\lambda_ \rightarrow m \ \$ h)$

And Then?

An example evaluation:

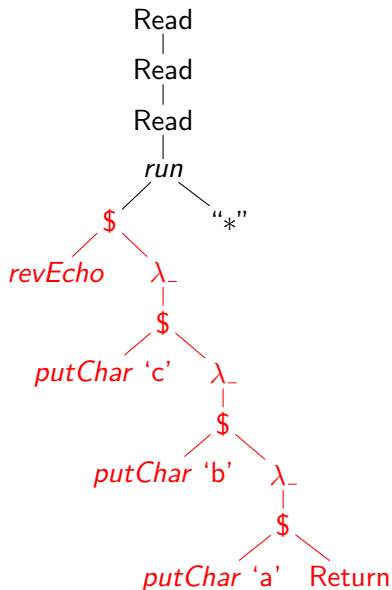


$run \text{ (GetChar } f) (c : cs) = \text{Read } (run \text{ (} f \text{ } c) \text{ } cs)$

$p \gg m = \lambda h \rightarrow p \$ (\lambda_ \rightarrow m \$ h)$

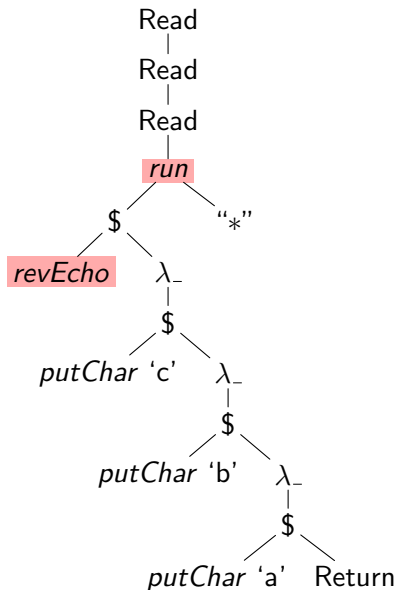
And Then?

An example evaluation:



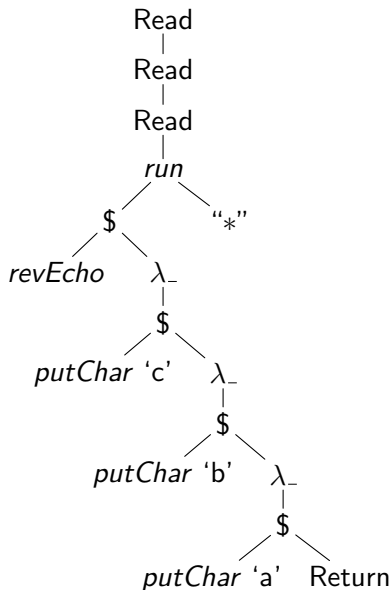
And Then?

An example evaluation:



And Then?

An example evaluation:



Overall, linear time!

More General Considerations

- ▶ Was the transformation semantically correct?

More General Considerations

- ▶ Was the transformation semantically correct?
- ▶ Is something similar possible for other data types?

More General Considerations

- ▶ Was the transformation semantically correct?
- ▶ Is something similar possible for other data types?
- ▶ How to provide the transformation to the programmer?

More General Considerations

- ▶ Was the transformation semantically correct?
⇒ program calculation, monad laws
- ▶ Is something similar possible for other data types?
- ▶ How to provide the transformation to the programmer?

More General Considerations

- ▶ Was the transformation semantically correct?
⇒ program calculation, monad laws
- ▶ Is something similar possible for other data types?
⇒ generic development for arbitrary free monads
- ▶ How to provide the transformation to the programmer?

More General Considerations

- ▶ Was the transformation semantically correct?
⇒ program calculation, monad laws
- ▶ Is something similar possible for other data types?
⇒ generic development for arbitrary free monads
- ▶ How to provide the transformation to the programmer?
⇒ type constructor classes, rank-2 types

More General Considerations

- ▶ Was the transformation semantically correct?
⇒ program calculation, monad laws
- ▶ Is something similar possible for other data types?
⇒ generic development for arbitrary free monads
- ▶ How to provide the transformation to the programmer?
⇒ type constructor classes, rank-2 types

Next:

- ▶ even more monads?

More General Considerations

- ▶ Was the transformation semantically correct?
⇒ program calculation, monad laws
- ▶ Is something similar possible for other data types?
⇒ generic development for arbitrary free monads
- ▶ How to provide the transformation to the programmer?
⇒ type constructor classes, rank-2 types

Next:

- ▶ even more monads?
- ▶ dual concepts?




More General Considerations

- ▶ Was the transformation semantically correct?
⇒ program calculation, monad laws
- ▶ Is something similar possible for other data types?
⇒ generic development for arbitrary free monads
- ▶ How to provide the transformation to the programmer?
⇒ type constructor classes, rank-2 types

Next:

- ▶ even more monads?
- ▶ dual concepts?
- ▶ ...?

References

-  K. Claessen and R.J.M. Hughes.
QuickCheck: A lightweight tool for random testing of Haskell programs.
In International Conference on Functional Programming, Proceedings, pages 268–279. ACM Press, 2000.
-  W. Swierstra and T. Altenkirch.
Beauty in the beast: A functional semantics for the awkward squad.
In Haskell Workshop, Proceedings, pages 25–36. ACM Press, 2007.
-  W. Swierstra.
Data types à la carte.
Journal of Functional Programming, 18(4):423–436, 2008.