Functional Programming

Continuation Monad and Monad Transformers

Jevgeni Kabanov

Department of Computer Science
University of Tartu

Introduction

Previously on Functional Programming

- Monadic laws
- *Monad* class ($\gg=$ and *return*)
- *MonadPlus* class (*mzero* and *mplus*)
- **do**-notation
- Maybe, List and State monads
- IO monad
- Monadic Prelude

Introduction

Previously on Functional Programming

- Monadic laws
- *Monad* class ($\gg=$ and *return*)
- *MonadPlus* class (*mzero* and *mplus*)
- **do**-notation
- Maybe, List and State monads
- IO monad
- Monadic Prelude

Overview

1. *Cont* monad
2. Monadic Transformers

Outline

Continuations

square example 1

No continuations:

$$square :: Int \to Int$$
$$square\ x = x \uparrow 2$$
$$main = \textbf{do}$$
$$\quad \textbf{let}\ x = square\ 4$$
$$\quad print\ x$$

Continuations

square example 2

Continuation-passing style:

$$square :: Int \to (Int \to a) \to a$$
$$square\ x\ k = k\ (x \uparrow 2)$$
$$main = square\ 4\ print$$

Cont

Cont definition

$$\textbf{newtype}\ Cont\ r\ a = Cont\{runCont :: (a \to r) \to r\}$$

square example 3

Continuation hidden behind a monad:

$$square :: Int \to Cont\ r\ Int$$
$$square\ x = return\ (x \uparrow 2)$$
$$main = runCont\ (square\ 4)\ print$$

Cont

Cont definition

$$\textbf{newtype}\ Cont\ r\ a = Cont\{runCont :: (a \to r) \to r\}$$

$$\textbf{instance}\ Monad\ (Cont\ r)\ \textbf{where}$$
$$\quad return\ a = \lambda k \to k\ a$$
$$\quad m \gg\!= f = \lambda k \to m\ (\lambda a \to f\ a\ k)$$

Cont

Cont definition

$$\mathbf{newtype}\ Cont\ r\ a = Cont\{runCont :: (a \to r) \to r\}$$

$$\mathbf{instance}\ Monad\ (Cont\ r)\ \mathbf{where}$$

$$\quad return\ a = Cont\ \$\ \lambda k \to k\ a$$

$$\quad (Cont\ c) \gg\!= f =$$

$$\qquad Cont\ \$\ \lambda k \to c\ (\lambda a \to runCont\ (f\ a)\ k)$$

Since *Cont* is a **newtype**!

Cont

square example 4

What is the result?

$$square :: Int \to Cont\ r\ Int$$
$$square\ x = return\ (x \uparrow 2)$$
$$addThree :: Int \to Cont\ r\ Int$$
$$addThree\ x = return\ (x + 3)$$
$$main = runCont\ (square\ 4 \gg= addThree)\ print$$

Cont

square example 4

What is the result?

$$square :: Int \to Cont\ r\ Int$$
$$square\ x = return\ (x \uparrow 2)$$
$$addThree :: Int \to Cont\ r\ Int$$
$$addThree\ x = return\ (x + 3)$$
$$main = runCont\ (square\ 4 \gg= addThree)\ print$$

Output

```
Main> main
19
```

$callCC$

$callCC$ definition

$callCC$ captures the current continuation and passes it as an argument.

$$callCC :: ((a \to Cont\ r\ b) \to Cont\ r\ a) \to Cont\ r\ a$$

$callCC$

$callCC$ example 1

k is the current continuation, calling k causes immediate return.

$$callCC :: ((a \rightarrow Cont\ r\ b) \rightarrow Cont\ r\ a) \rightarrow Cont\ r\ a$$

$$\begin{array}{l} bar :: Cont\ r\ Int \\ bar = callCC\ \$\ \lambda k \rightarrow \mathbf{do} \\ \quad \mathbf{let}\ n = 5 \\ \quad k\ n \\ \quad return\ 25 \end{array}$$

$$main = runCont\ bar\ print$$

Always prints 5.

$callCC$

$callCC$ example 2

$$
\begin{array}{l}
foo :: Int \rightarrow Cont\ r\ String \\
foo\ n = \\
\quad callCC\ \$\ \lambda k \rightarrow \textbf{do} \\
\quad\quad \textbf{let}\ n' = n \uparrow 2 + 3 \\
\quad\quad when\ (n' > 20)\ \$\ k\ \texttt{"over twenty"} \\
\quad\quad return\ (show\ \$\ n' - 4)
\end{array}
$$

Output

```
Main> runCont (foo 5) print
over twenty
Main> runCont (foo 4) print
15
```

callCC

callCC example 3

Exceptions are simpler than continuations:

$$
\begin{array}{l}
divExcpt\ x\ y\ handler = \\
\quad callCC\ \$\ \lambda ok \rightarrow \mathbf{do} \\
\quad\quad err \leftarrow callCC\ \$\ \lambda notOk \rightarrow \mathbf{do} \\
\quad\quad\quad when\ (y \equiv 0)\ \$\ notOk\ \texttt{"Denominator 0"} \\
\quad\quad\quad ok\ \$\ x\ \text{`}div\text{`}\ y \\
\quad\quad handler\ err
\end{array}
$$

Output

```
Main> runCont (divExcpt 10 2 error) id
5
Main>runCont (divExcpt 10 0 error) id
*** Exception: Denominator 0
```

callCC

callCC example 4

```
fun :: Int -> String
fun n = (`runCont` id) $ do
  str <- callCC $ \exit1 -> do
    when (n < 10) (exit1 (show n))
    let ns = map digitToInt (show (n `div` 2))
    n' <- callCC $ \exit2 -> do
      when ((length ns) < 3) (exit2 (length ns))
      when ((length ns) < 5) (exit2 n)
      when ((length ns) < 7) $ do
        let ns' = map intToDigit (reverse ns)
        exit1 (dropWhile (=='0') ns')
      return $ sum ns
    return $ "(ns = " ++ (show ns) ++ ") " ++ (show n')
  return $ "Answer: " ++ str
```

callCC

callCC example 4

Input (n)	Output	List Shown
0-9	n	none
10-199	number of digits in (n/2)	digits of (n/2)
200-19999	n	digits of (n/2)
20000-1999999	(n/2) backwards	none
>= 2000000	sum of digits of (n/2)	digits of (n/2)

$callCC$

$callCC$ definition 2

$$
\begin{array}{l}
\textbf{class}\ (Monad\ m) \Rightarrow MonadCont\ m\ \textbf{where} \\
\quad callCC :: ((a \to m\ b) \to m\ a) \to m\ a \\
\textbf{instance}\ MonadCont\ (Cont\ r)\ \textbf{where} \\
\quad callCC\ f = \\
\qquad Cont\ \$\ \lambda k \to runCont\ (f\ (\lambda a \to Cont\ \$\ \lambda\_ \to k\ a))\ k
\end{array}
$$

Outline

Monadic Transformers

Outline

- We will begin by simplifying the previous example
- Then we will try to enhance it by adding some IO
- Finally we will generalize the approach to arbitrary monads

Monadic Transformers

Outline

- We will begin by simplifying the previous example
- Then we will try to enhance it by adding some IO
- Finally we will generalize the approach to arbitrary monads

Monadic Transformers

Outline

- We will begin by simplifying the previous example
- Then we will try to enhance it by adding some IO
- Finally we will generalize the approach to arbitrary monads

IO + Cont

Example 1

```
fun :: Int -> String
fun n = (`runCont` id) $ do
  str <- callCC $ \exit1 -> do
    when (n < 10) (exit1 (show n))
    let ns = map digitToInt (show (n `div` 2))
    n' <- callCC $ \exit2 -> do
      when ((length ns) < 5) (exit2 n)
      return $ sum ns
    return $ "(ns = " ++ (show ns) ++ ") " ++ (show n')
  return $ "Answer: " ++ str
```

IO + Cont

Example 1

Input (n)	Output	List Shown
0-9	n	none
~~10-199~~	~~number of digits in (n/2)~~	~~digits of (n/2)~~
10-19999	n	digits of (n/2)
~~20000-1999999~~	~~(n/2) backwards~~	~~none~~
>= 20000	sum of digits of (n/2)	digits of (n/2)

IO + *Cont*

Example 2

The easiest way to add IO is to nest *Cont* inside *IO*:

```
fun :: IO String
fun = do n <- (readLn::IO Int)
  return $ (`runCont` id) $ do
    str <- callCC $ \exit1 -> do
      when (n < 10) (exit1 (show n))
      let ns = map digitToInt (show (n `div` 2))
      n' <- callCC $ \exit2 -> do
        when ((length ns) < 5) (exit2 n)
        return $ sum ns
      return $ "(ns = " ++ (show ns) ++ ") " ++ (show n')
    return $ "Answer: " ++ str
```

$IO + Cont$

Adding IO

- What do we do when we need to use IO inside *Cont* monad?
- We could try to just lift the continuation result value into IO

toIO definition

$$toIO :: a \to IO\ a$$
$$toIO\ x = return\ x$$

$IO + Cont$

Adding IO

- What do we do when we need to use IO inside *Cont* monad?
- We could try to just lift the continuation result value into IO

toIO definition

$$toIO :: a \rightarrow IO\ a$$
$$toIO\ x = return\ x$$

$IO + Cont$

Adding IO

- What do we do when we need to use IO inside *Cont* monad?
- We could try to just lift the continuation result value into IO

toIO definition

$$toIO :: a \rightarrow IO\ a$$
$$toIO\ x = return\ x$$

IO + Cont

Example 1

```
fun :: Int -> String
fun n = (`runCont` id) $ do
  str <- callCC $ \exit1 -> do
    when (n < 10) (exit1 (show n))
    let ns = map digitToInt (show (n `div` 2))
    n' <- callCC $ \exit2 -> do
      when ((length ns) < 5) (exit2 n)
      return $ sum ns
    return $ "(ns = " ++ (show ns) ++ ") " ++ (show n')
  return $ "Answer: " ++ str
```

IO + Cont

Example 4

```
fun :: Int -> IO String
fun n = (‘runCont‘ id) $ do
  str <- callCC $ \exit1 -> do
    when (n < 10) (exit1 $ toIO (show n))
    let ns = map digitToInt (show (n ‘div‘ 2))
    n’ <- callCC $ \exit2 -> do
      when ((length ns) < 5) (exit2 $
        do putStrLn "Enter a number:"
        x <- (readLn::IO Int)
        return x)
      return (toIO (sum ns))
    return $
      do num <- n’
      return $ "(ns = " ++ (show ns) ++ ") " ++ (show num)
  return $ do s <- str
              return $ "Answer: " ++ s
```

$IO + Cont$

Adding IO 2

- This adds useless conversions to/from IO
- We would IO only where actually needed
- This is where monadic transformers come in

liftIO definition

liftIO allows to run IO code inside a monad.

$$\text{class } (Monad\ m) \Rightarrow MonadIO\ m \text{ where}$$
$$\quad liftIO :: IO\ a \rightarrow m\ a$$

$IO + Cont$

Adding IO 2

- This adds useless conversions to/from IO
- We would IO only where actually needed
- This is where monadic transformers come in

$liftIO$ definition

$liftIO$ allows to run IO code inside a monad.

$$\text{class } (Monad\ m) \Rightarrow MonadIO\ m \text{ where}$$
$$\quad liftIO :: IO\ a \to m\ a$$

$IO + Cont$

Adding IO 2

- This adds useless conversions to/from IO
- We would IO only where actually needed
- This is where monadic transformers come in

liftIO definition

liftIO allows to run IO code inside a monad.

$$\text{class } (Monad\ m) \Rightarrow MonadIO\ m \text{ where}$$
$$\quad liftIO :: IO\ a \rightarrow m\ a$$

IO + Cont

Adding IO 2

- This adds useless conversions to/from IO
- We would IO only where actually needed
- This is where monadic transformers come in

liftIO definition

liftIO allows to run IO code inside a monad.

$$\textbf{class}\ (Monad\ m) \Rightarrow MonadIO\ m\ \textbf{where}$$
$$\quad liftIO :: IO\ a \rightarrow m\ a$$

IO + Cont

Example 2

```
fun :: IO String
fun = do n <- (readLn::IO Int)
  return $ (`runCont` id) $ do
    str <- callCC $ \exit1 -> do
      when (n < 10) (exit1 (show n))
      let ns = map digitToInt (show (n `div` 2))
      n' <- callCC $ \exit2 -> do
        when ((length ns) < 5) (exit2 n)
        return $ sum ns
      return $ "(ns = " ++ (show ns) ++ ") " ++ (show n')
    return $ "Answer: " ++ str
```

IO + *Cont*

Example 5

```
fun :: IO String
fun = (`runContT` return) $ do
  n   <- liftIO (readLn::IO Int)
  str <- callCC $ \exit1 -> do
    when (n < 10) (exit1 (show n))
    let ns = map digitToInt (show (n `div` 2))
    n' <- callCC $ \exit2 -> do
      when ((length ns) < 5) $ do
        liftIO $ putStrLn "Enter a number:"
        x <- liftIO (readLn::IO Int)
        exit2 x
      return $ sum ns
    return $ "(ns = " ++ (show ns) ++ ") " ++ (show n')
  return $ "Answer: " ++ str
```

Transformers

MonadTrans

$$\textbf{class } \mathit{MonadTrans}\ t \textbf{ where}$$
$$\quad \mathit{lift} :: (\mathit{Monad}\ m) \Rightarrow m\ a \to t\ m\ a$$
$$\textbf{class } (\mathit{Monad}\ m) \Rightarrow \mathit{MonadIO}\ m \textbf{ where}$$
$$\quad \mathit{liftIO} :: \mathit{IO}\ a \to m\ a$$

Instances

Monad	Transformer	Original	Combined
Error	ErrorT	$\mathit{Either}\ e\ a$	$m\ (\mathit{Either}\ e\ a)$
State	StateT	$s \to (a, s)$	$s \to m\ (a, s)$
Reader	$\mathit{ReaderT}$	$r \to a$	$r \to m\ a$
$[\,]$	ListT	$[a]$	$m\ [a]$
Cont	ContT	$(a \to r) \to r$	$(a \to m\ r) \to m\ r$

Transformers

StateT definition

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{StateT}\ s\ m\ a = \\
\quad \mathit{StateT}\{\mathit{runStateT} :: (s \rightarrow m\ (a, s))\} \\
\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Monad}\ (\mathit{StateT}\ s\ m)\ \textbf{where} \\
\quad \mathit{return}\ a \qquad\qquad\quad = \mathit{StateT}\ \$\ \lambda s \rightarrow \mathit{return}\ (a, s) \\
\quad (\mathit{StateT}\ x) \gg\!= f = \mathit{StateT}\ \$\ \lambda s \rightarrow \textbf{do} \\
\qquad\qquad\qquad\qquad\quad (v, s') \leftarrow x\ s \\
\qquad\qquad\qquad\qquad\quad (\mathit{StateT}\ x') \leftarrow \mathit{return}\ \$\ f\ v \\
\qquad\qquad\qquad\qquad\quad x'\ s' \\
\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadState}\ s\ (\mathit{StateT}\ s\ m)\ \textbf{where} \\
\quad \mathit{get} = \mathit{StateT}\ \$\ \lambda s \rightarrow \mathit{return}\ (s, s) \\
\quad \mathit{put}\ s = \mathit{StateT}\ \$\ \lambda\_ \rightarrow \mathit{return}\ ((), s) \\
\textbf{instance}\ \mathit{MonadTrans}\ (\mathit{StateT}\ s)\ \textbf{where} \\
\quad \mathit{lift}\ c = \mathit{StateT}\ \$\ \lambda s \rightarrow c \gg\!= (\lambda x \rightarrow \mathit{return}\ (x, s))
\end{array}
$$

Transformers

Intermission

- Transformers wrap monads to create combined monads
- Transformer combined with *Identity* monad is same as original. E.g. *StateT s Identity* is same as *State s*
- Order is important. *StateT s* (*Either e*) with type $s \rightarrow Either\ e\ (a, s)$ is different from *ErrorT e* (*State s*) with type $s \rightarrow (Either\ e\ a, s)$
- Transformer bind is combined, so all monads end up bound. E.g. *StateT s* [] with type $s \rightarrow [(a, s)]$ will bind both state and list, producing a list of both values and state on every bind.
- We still need to run inner monads. *ContT r IO a* will produce $(a \rightarrow IO\ r) \rightarrow IO\ r$, so we need to *runContT* first and bind IO later
- *liftIO* is just *lift* specialized for IO monad

Transformers

Intermission

- Transformers wrap monads to create combined monads
- Transformer combined with *Identity* monad is same as original. E.g. *StateT s Identity* is same as *State s*
- Order is important. *StateT s* (*Either e*) with type $s \to Either\ e\ (a, s)$ is different from *ErrorT e* (*State s*) with type $s \to (Either\ e\ a, s)$
- Transformer bind is combined, so all monads end up bound. E.g. *StateT s* [] with type $s \to [(a, s)]$ will bind both state and list, producing a list of both values and state on every bind.
- We still need to run inner monads. *ContT r IO a* will produce $(a \to IO\ r) \to IO\ r$, so we need to *runContT* first and bind IO later
- *liftIO* is just *lift* specialized for IO monad

Transformers

Intermission

- Transformers wrap monads to create combined monads
- Transformer combined with *Identity* monad is same as original. E.g. $StateT\ s\ Identity$ is same as $State\ s$
- Order is important. $StateT\ s\ (Either\ e)$ with type $s \to Either\ e\ (a, s)$ is different from $ErrorT\ e\ (State\ s)$ with type $s \to (Either\ e\ a, s)$
- Transformer bind is combined, so all monads end up bound. E.g. $StateT\ s\ []$ with type $s \to [(a, s)]$ will bind both state and list, producing a list of both values and state on every bind.
- We still need to run inner monads. $ContT\ r\ IO\ a$ will produce $(a \to IO\ r) \to IO\ r$, so we need to $runContT$ first and bind IO later
- $liftIO$ is just $lift$ specialized for IO monad

Transformers

Intermission

- Transformers wrap monads to create combined monads
- Transformer combined with *Identity* monad is same as original. E.g. $StateT\ s\ Identity$ is same as $State\ s$
- Order is important. $StateT\ s\ (Either\ e)$ with type $s \to Either\ e\ (a, s)$ is different from $ErrorT\ e\ (State\ s)$ with type $s \to (Either\ e\ a, s)$
- Transformer bind is combined, so all monads end up bound. E.g. $StateT\ s\ []$ with type $s \to [(a, s)]$ will bind both state and list, producing a list of both values and state on every bind.
- We still need to run inner monads. $ContT\ r\ IO\ a$ will produce $(a \to IO\ r) \to IO\ r$, so we need to $runContT$ first and bind IO later
- $liftIO$ is just $lift$ specialized for IO monad

Transformers

Intermission

- Transformers wrap monads to create combined monads
- Transformer combined with *Identity* monad is same as original. E.g. $StateT\ s\ Identity$ is same as $State\ s$
- Order is important. $StateT\ s\ (Either\ e)$ with type $s \to Either\ e\ (a, s)$ is different from $ErrorT\ e\ (State\ s)$ with type $s \to (Either\ e\ a, s)$
- Transformer bind is combined, so all monads end up bound. E.g. $StateT\ s\ [\,]$ with type $s \to [(a, s)]$ will bind both state and list, producing a list of both values and state on every bind.
- We still need to run inner monads. $ContT\ r\ IO\ a$ will produce $(a \to IO\ r) \to IO\ r$, so we need to $runContT$ first and bind IO later
- $liftIO$ is just $lift$ specialized for IO monad

Transformers

Intermission

- Transformers wrap monads to create combined monads
- Transformer combined with *Identity* monad is same as original. E.g. $StateT\ s\ Identity$ is same as $State\ s$
- Order is important. $StateT\ s\ (Either\ e)$ with type $s \to Either\ e\ (a, s)$ is different from $ErrorT\ e\ (State\ s)$ with type $s \to (Either\ e\ a, s)$
- Transformer bind is combined, so all monads end up bound. E.g. $StateT\ s\ [\,]$ with type $s \to [(a, s)]$ will bind both state and list, producing a list of both values and state on every bind.
- We still need to run inner monads. $ContT\ r\ IO\ a$ will produce $(a \to IO\ r) \to IO\ r$, so we need to $runContT$ first and bind IO later
- *liftIO* is just *lift* specialized for IO monad

Transformers

Example 6

The Kalotans are a tribe with a peculiar quirk: their males always tell the truth. Their females never make two consecutive true statements, or two consecutive untrue statements. An anthropologist (let's call him Worf) has begun to study them. Worf does not yet know the Kalotan language. One day, he meets a Kalotan (heterosexual) couple and their child Kibi. Worf asks Kibi: "Are you a boy?" The kid answers in Kalotan, which of course Worf doesn't understand. Worf turns to the parents (who know English) for explanation. One of them says: "Kibi said: 'I am a boy.'" The other adds: "Kibi is a girl. Kibi lied." Solve for the sex of Kibi and the sex of each parent.