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 (\texttt{>>=} and \texttt{return})
- MonadPlus class (\texttt{mzero} and \texttt{mplus})
- do-notation
- Maybe, List and State monads
- IO monad
- Monadic Prelude
Overview

1. Cont monad

2. Monadic Transformers
Outline

1. *Cont* monad

2. Monadic Transformers
Continuations

**square example 1**

No continuations:

```haskell
square :: Int -> Int
square x = x ^ 2
main = do
  let x = square 4
  print x
```
Continuations

square example 2

Continuation-passing style:

\[
square :: \text{Int} \rightarrow (\text{Int} \rightarrow a) \rightarrow a
\]
\[
square x k = k (x \uparrow 2)
\]
\[
\text{main} = \text{square 4 print}
\]
Cont
definition

newtype Cont r a = Cont {runCont :: (a -> r) -> r}

square example 3
Continuation hidden behind a monad:

square :: Int -> Cont r Int
square x = return (x \^ 2)
main = runCont (square 4) print
Cont

Cont definition

```haskell
newtype Cont r a = Cont {runCont :: (a -> r) -> r }
instance Monad (Cont r) where
  return a = \k -> k a
  m >>= f = \k -> m (\a -> f a k)
```


Cont

Cont definition

newtype Cont r a = Cont { runCont :: (a -> r) -> r }

instance Monad (Cont r) where
  return a = Cont $ \k \to k a
  (Cont c) >>= f =
    Cont $ \k \to c (\a \to runCont (f a) k)

Since Cont is a newtype!
square example 4
What is the result?

\[
\begin{align*}
\text{square} &:: \text{Int} \to \text{Cont r Int} \\
\text{square } x &= \text{return } (x \uparrow 2) \\
\text{addThree} &:: \text{Int} \to \text{Cont r Int} \\
\text{addThree } x &= \text{return } (x + 3) \\
\text{main} &= \text{runCont } (\text{square } 4 \gg= \text{addThree}) \text{ print}
\end{align*}
\]
**Cont**

**square example 4**

What is the result?

```
square :: Int → Cont r Int
square x = return (x ↑ 2)
addThree :: Int → Cont r Int
addThree x = return (x + 3)
main = runCont (square 4 >>= addThree) print
```

**Output**

Main> main
19
**callCC**

**callCC definition**

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

\[
\text{callCC} :: ((a \rightarrow \text{Cont } r \ b) \rightarrow \text{Cont } r \ a) \rightarrow \text{Cont } r \ a
\]
**callCC**

**callCC example 1**

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

\[ \text{callCC} :: ((a -> \text{Cont} r b) -> \text{Cont} r a) -> \text{Cont} r a \]

\[ \text{bar} :: \text{Cont} r \text{Int} \]

\[ \text{bar} = \text{callCC} \ \& \ \lambda k \rightarrow \text{do} \]

\[ \text{let } n = 5 \]

\[ k \ n \]

\[ \text{return} 25 \]

\[ \text{main} = \text{runCont} \ \text{bar} \ \text{print} \]

Always prints 5.
**callCC**

**callCC example 2**

```haskell
foo :: Int → Cont r String
foo n =
    callCC $ λk → do
        let n' = n \(↑\) 2 + 3
        when (n' > 20) $ k "over twenty"
        return (show $ n' - 4)
```

**Output**

Main> runCont (foo 5) print
over twenty
Main> runCont (foo 4) print
15
Exceptions are simpler than continuations:

```haskell
divExcpt x y handler =
callCC $ \lambda ok \rightarrow do
  err \leftarrow callCC $ \lambda notOk \rightarrow do
    when (y \equiv 0) $ notOk "Denominator 0"
    ok $ x 'div' y
  handler err
```

Output

```
Main> runCont (divExcpt 10 2 error) id
5
Main> runCont (divExcpt 10 0 error) id
*** Exception: Denominator 0
```
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` example 4

<table>
<thead>
<tr>
<th>Input (n)</th>
<th>Output</th>
<th>List Shown</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-9</td>
<td>n</td>
<td>none</td>
</tr>
<tr>
<td>10-199</td>
<td>number of digits in ((n/2))</td>
<td>digits of ((n/2))</td>
</tr>
<tr>
<td>200-19999</td>
<td>(n)</td>
<td>digits of ((n/2))</td>
</tr>
<tr>
<td>20000-1999999</td>
<td>((n/2)) backwards</td>
<td>none</td>
</tr>
<tr>
<td>(\geq 2000000)</td>
<td>sum of digits of ((n/2))</td>
<td>digits of ((n/2))</td>
</tr>
</tbody>
</table>
class \(\text{Monad } m\) ⇒ \(\text{MonadCont } m\) where
\[
\text{callCC} :: ((a \rightarrow m\ b) \rightarrow m\ a) \rightarrow m\ a
\]

instance \(\text{MonadCont } (\text{Cont } r)\) where
\[
\text{callCC} f = \\
\text{Cont} \ \$ \ \lambda k \rightarrow \text{runCont} \ (f \ (\lambda a \rightarrow \text{Cont} \ \$ \ \lambda \_ \rightarrow k\ a))\ k
\]
Outline

1. Cont monad

2. Monadic Transformers
# 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
Example 1

```haskell
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
```
### Example 1

<table>
<thead>
<tr>
<th>Input (n)</th>
<th>Output</th>
<th>List Shown</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-9</td>
<td>n</td>
<td>none</td>
</tr>
<tr>
<td>10-199</td>
<td>number of digits in ((n/2))</td>
<td>digits of ((n/2))</td>
</tr>
<tr>
<td>10-19999</td>
<td>n</td>
<td>digits of ((n/2))</td>
</tr>
<tr>
<td>20000-1999999</td>
<td>((n/2)) backwards</td>
<td>none</td>
</tr>
<tr>
<td>(\geq 20000)</td>
<td>sum of digits of ((n/2))</td>
<td>digits of ((n/2))</td>
</tr>
</tbody>
</table>
Example 2

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

```haskell
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
```
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

\[
\text{toIO} :: a \rightarrow \text{IO} \ a
\]

\[
\text{toIO} \ x = \text{return} \ x
\]
Adding IO

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

\textit{toIO} definition

\begin{align*}
\text{toIO} :: & \ a \rightarrow \text{IO} \ a \\
\text{toIO} \ x &= \text{return} \ x
\end{align*}
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 -> IO a
toIO x = return x
```
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
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
Adding IO 2

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

\textit{liftIO} definition

\textit{liftIO} allows to run IO code inside a monad.

\begin{verbatim}
class (Monad m) \Rightarrow MonadIO m where
  liftIO :: IO a \rightarrow m a
\end{verbatim}
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.

```haskell
class (Monad m) ⇒ MonadIO m where
    liftIO :: IO a → m a
```
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.

```
class (Monad m) ⇒ MonadIO m where
    liftIO :: IO a → m a
```
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.

```
class (Monad m) => MonadIO m where
  liftIO :: IO a -> m a
```
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
Example 5

```haskell
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

class MonadTrans t where
  lift :: (Monad m) ⇒ m a → t m a

class (Monad m) ⇒ MonadIO m where
  liftIO :: IO a → m a

Instances

<table>
<thead>
<tr>
<th>Monad</th>
<th>Transformer</th>
<th>Original</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>ErrorT</td>
<td>Either e a</td>
<td>m (Either e a)</td>
</tr>
<tr>
<td>State</td>
<td>StateT</td>
<td>s → (a, s)</td>
<td>s → m (a, s)</td>
</tr>
<tr>
<td>Reader</td>
<td>ReaderT</td>
<td>r → a</td>
<td>r → m a</td>
</tr>
<tr>
<td>[]</td>
<td>ListT</td>
<td>[a]</td>
<td>m [a]</td>
</tr>
<tr>
<td>Cont</td>
<td>ContT</td>
<td>(a → r) → r</td>
<td>(a → m r) → m r</td>
</tr>
</tbody>
</table>
Transformers

**StateT definition**

```
newtype StateT s m a =
  StateT{runStateT :: (s → m (a, s))}

instance (Monad m) ⇒ Monad (StateT s m) where
  return a = StateT $ λs → return (a, s)
  (StateT x) >>= f = StateT $ λs → do
    (v, s') ← x s
    (StateT x') ← return $ f v
  x' s'

instance (Monad m) ⇒ MonadState s (StateT s m) where
  get = StateT $ λs → return (s, s)
  put s = StateT $ λ_ → return (((), s)

instance MonadTrans (StateT s) where
  lift c = StateT $ λs → c >>= (λx → return (x, s))
```
Transformers

Intermission

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

Intermission

- Transformers wrap monads to create combined monads
- Transformer combined with Identity monad is same as original. E.g. \( \text{StateT} \ s \ \text{Identity} \) is same as \( \text{State} \ s \)
- Order is important. \( \text{StateT} \ s \ (\text{Either} \ e) \) with type \( s \rightarrow \text{Either} \ e \ (a, s) \) is different from \( \text{ErrorT} \ e \ (\text{State} \ s) \) with type \( s \rightarrow (\text{Either} \ e \ a, s) \)
- Transformer bind is combined, so all monads end up bound. E.g. \( \text{StateT} \ s \ [] \) 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. \( \text{ContT} \ r \ \text{IO} \ a \) will produce \( (a \rightarrow \text{IO} \ r) \rightarrow \text{IO} \ r \), so we need to \( \text{runContT} \) first and bind IO later
- \( \text{liftIO} \) is just \( \text{lift} \) 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 $s \rightarrow Either e (a, s)$ 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 → Either e (a, s) is different from ErrorT e (State s) with type s → (Either e a, s)
- Transformer bind is combined, so all monads end up bound. E.g. StateT s [] with type s → [(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 → IO r) → 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. \( \text{StateT} \ s \ \text{Identity} \) is same as \( \text{State} \ s \)
- Order is important. \( \text{StateT} \ s \ (\text{Either} \ e) \) with type \( s \rightarrow \text{Either} \ e \ (a, s) \) is different from \( \text{ErrorT} \ e \ (\text{State} \ s) \) with type \( s \rightarrow (\text{Either} \ e \ a, s) \)
- Transformer bind is combined, so all monads end up bound. E.g. \( \text{StateT} \ s \ [] \) 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. \( \text{ContT} \ r \ \text{IO} \ a \) will produce \( (a \rightarrow \text{IO} \ r) \rightarrow \text{IO} \ r \), so we need to \( \text{runContT} \) first and bind IO later
- \( \text{liftIO} \) is just \( \text{lift} \) 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 → Either e (a, s) is different from ErrorT e (State s) with type s → (Either e a, s)
- Transformer bind is combined, so all monads end up bound. E.g. StateT s [] with type s → [(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 → IO r) → IO r, so we need to runContT first and bind IO later
- liftIO is just lift specialized for IO monad
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.