Continuations in Haskell

Ryan Orendorff (ryan@orendorff.io)

12 May, 2015

What we are going to cover today

- The basics of CPS style
- The `Cont` monad
- Call with Current Continuation (`callCC`)
- An motivating use case

The Basics of CPS Style

How do we compute things?

Computing a value is normally quite simple.

```haskell
type Radius = Double
type Area = Double

areaCircle :: Radius → Area
areaCircle r = pi * r * r

areaCircle 1 — π
```

How do we compute things?

Computing a value is normally quite simple.

```
type Radius = Double
type Area = Double

areaCircle :: Radius → Area
areaCircle r = pi * r * r

areaCircle 1 — π
```

But what if we have some value (radius/width/etc), but we don't know what area function to use yet.

How do we make a value wait for a function?

Normally we have a function wait for a value.

```
($) :: (a → b) → a → b
f $ x = f x
```

How do we make a value wait for a function?

Normally we have a function wait for a value.

```
($) :: (a → b) → a → b
f $ x = f x
```

What if we flip the arguments?

```
(£) :: a → (a → b) → b
(£) = flip ($)
— or (£) = \x → (\f → f x)
```

Now we can do all sorts of things to the same radius

```haskell
type Height = Double; type Volume = Double; type Diameter = Double

volumeCylinder :: Radius → Height → Volume
volumeCylinder r h = areaCircle r * h

diameter :: Radius → Diameter
diameter r = 2*r

doSomethingToR :: (Radius → r) → r
doSomethingToR = (1.0 £)
```

Now we can do all sorts of things to the same radius

```haskell
type Height = Double; type Volume = Double; type Diameter = Double

volumeCylinder :: Radius → Height → Volume
volumeCylinder r h = areaCircle r * h

diameter :: Radius → Diameter
diameter r = 2*r

doSomethingToR :: (Radius → r) → r
doSomethingToR = (1.0 £)

doSomethingToR areaCircle          — π

— doSomethingToR volumeCylinder :: Height → Volume
doSomethingToR volumeCylinder 2 — 2π

doSomethingToR diameter            — 2
```

Suspended computation

`f` creates what is known as a suspended computation.

`areaCircle` — `Waiting for an input`

Suspended computation

`£` creates what is known as a suspended computation.

`areaCircle` — `Waiting for an input`

`areaCircle 1` — `Represents a value`

Suspended computation

`£` creates what is known as a suspended computation.

`areaCircle` — `Waiting for an input`

`areaCircle 1` — `Represents a value`

`(1 £)` — `Do something to the value 1 later`

Suspended computation

£ creates what is known as a suspended computation.

`areaCircle` — `Waiting for an input`

`areaCircle 1` — `Represents a value`

`(1 £)` — `Do something to the value 1 later`

`(areaCircle 1 £)` — `Wait to do something with the`
                                  `— result of areaCircle 1`

Suspended computation

£ creates what is known as a suspended computation.

`areaCircle` — Waiting for an input

`areaCircle 1` — Represents a value

`(1 £)` — Do something to the value 1 later

`(areaCircle 1 £)` — Wait to do something with the
— result of areaCircle 1

`areaCircle 1 £ id` — Execute 'areaCircle 1',
— pass the result to id

Standard Notation for Suspended Computations

In the literature/online, you will likely see turning a value into a
suspended computation written as follows.

```
cpsify :: a → (a → r) → r
cpsify x k = k x
— or cpsify x = \k → k x
```

where

- **r** is the final result of evaluating the computation.
- **a → r** is what you are doing to do to **x** later. This is
  represented here as **k**.
- **k** is also called the "continuation", or "continue on with the
  value you pass to me".

Currently we can't do much

We can create suspended computations quite easily using `cpsify`.
What we really need is something to *chain* our suspended
computations together.

Chaining Training Wheels

Say we have two suspended computations.

```
suspendArea :: Radius → (Area → r) → r
suspendArea r = \k → k (areaCircle r)

— Calculate the volume of any extrusion.
suspendExtrude :: Area → Height → (Volume → r) → r
suspendExtrude a h = \k → k (a * h)
```

Chaining Training Wheels

Say we have two suspended computations.

```
suspendArea :: Radius → (Area → r) → r
suspendArea r = \k → k (areaCircle r)

— Calculate the volume of any extrusion.
suspendExtrude :: Area → Height → (Volume → r) → r
suspendExtrude a h = \k → k (a * h)
```

What we would like is something like this

```
suspendVolume :: Radius → Height → (Volume → r) → r
```

Chaining Training Wheels

We could try this.

```
suspendVolume :: Radius → Height → (Volume → r) → r
suspendVolume r h = \k → ?    — k :: (Volume → r) → r
```

Chaining Training Wheels

We could try this.

```
suspendVolume :: Radius → Height → (Volume → r) → r
suspendVolume r h = \k →        — k :: (Volume → r) → r
  suspendArea r                 — needs a (Area → r)
```

Chaining Training Wheels

We could try this.

```
suspendVolume :: Radius → Height → (Volume → r) → r
suspendVolume r h = \k →          — k :: (Volume → r) → r
  suspendArea r                   — needs a (Area → r)
  (\area → ?)                     — needs to return r
```

Chaining Training Wheels

We could try this.

```
suspendVolume :: Radius → Height → (Volume → r) → r
suspendVolume r h = \k →        — k :: (Volume → r) → r
  suspendArea r                 — needs a (Area → r)
  (\area →                      — needs to return r
    suspendExtrude area h ?)    — needs a (Volume → r)
```

Chaining Training Wheels

We could try this.

```
suspendVolume :: Radius → Height → (Volume → r) → r
suspendVolume r h = \k →        — k :: (Volume → r) → r
  suspendArea r                 — needs a (Area → r)
  (\area →                      — needs to return r
    suspendExtrude area h       — needs a (Volume → r)
      k)                        — k is a (Volume → r)!
```

suspendVolume seems to work

We get a suspended computation that calculates the volume and
waits for us to do something with it.

```
suspendVolume 1 2        — :: (Volume → r) → r
suspendVolume 1 2 id     — 2π
suspendVolume 1 2 (*2)   — 2 * 2π = 4π
```

Let's break it down to make sure we are calculating what we want.

Breakdown suspendVolume

— suspendArea r = \k → k (areaCircle r)
— suspendExtrude a h = \k → k (a * h)

```
suspendVolume :: Radius → Height → (Volume → r) → r
suspendVolume r h = \k →        — k :: (Volume → r) → r
  (suspendArea r)               — needs a (Area → r)
  (\area →                      — needs to return r
    (suspendExtrude area h)     — needs a (Volume → r)
      k)                        — k is a (Volume → r)!
```

Expand suspendArea.

Breakdown suspendVolume

— suspendArea r = \k → k (areaCircle r)
— suspendExtrude a h = \k → k (a * h)

```
suspendVolume :: Radius → Height → (Volume → r) → r
suspendVolume r h = \k →        — k :: (Volume → r) → r
  (\k → k (areaCircle r)        — needs a (Area → r)
  (\area →                      — needs to return r
    (suspendExtrude area h)     — needs a (Volume → r)
      k)                        — k is a (Volume → r)!
```

Evaluate the (\k → k (areaCircle r)).

Breakdown suspendVolume

```
— suspendArea r = \k → k (areaCircle r)
— suspendExtrude a h = \k → k (a * h)

suspendVolume :: Radius → Height → (Volume → r) → r
suspendVolume r h = \k →        — k :: (Volume → r) → r
  (\area →                      — needs to return r
    (suspendExtrude area h)     — needs a (Volume → r)
      k)                        — k is a (Volume → r)!
  (areaCircle r)                — areaCircle applied to
                                —   \area → ...
```

Substitude in area.

Breakdown **suspendVolume**

— suspendArea r = \k → k (areaCircle r)
— suspendExtrude a h = \k → k (a * h)

suspendVolume **::** Radius → Height → (Volume → r) → r
suspendVolume r h = \k →   — k **::** (Volume → r) → r
  (suspendExtrude (areaCircle r) h) — needs (Volume → r)
      k)   — k is a (Volume → r)!

Expand **suspendExtrude**.

Breakdown **suspendVolume**

— suspendArea r = \k → k (areaCircle r)
— suspendExtrude a h = \k → k (a * h)

suspendVolume **::** Radius → Height → (Volume → r) → r
suspendVolume r h = \k →   — k **::** (Volume → r) → r
  (\k → k **$** (areaCircle r) * h) — needs (Volume → r)
     k)   — k is a (Volume → r)!

Apply k.

Breakdown suspendVolume

— `suspendArea r = \k → k (areaCircle r)`

— `suspendExtrude a h = \k → k (a * h)`

```
suspendVolume' :: Radius → Height → (Volume → r) → r
suspendVolume' r h = \k →      — k :: (Volume → r) → r
  k $ areaCircle r * h
```

Well that is certainly correct!

Manual chaining is suspended rear pain

We can do better. We have a common pattern here

```
suspendA :: (a → r) → r
suspendB :: a → (b → r) → r
suspendA (\a → (suspendB a) (\b → ...))
```

Let's turn this pattern into a function[1]

```
chain :: ((a → r) → r) → (a → ((b → r) → r)) →
         ((b → r) → r)
```

---

[1]From The Haskell Wikibook, although there it is called `chainCPS`

Manual chaining is suspended rear pain

We can do better. We have a common pattern here

```
suspendA :: (a → r) → r
suspendB :: a → (b → r) → r
suspendA (\a → (suspendB a) (\b → ...))
```

Let's turn this pattern into a function[1]

```
chain :: ((a → r) → r) → (a → ((b → r) → r)) →
         ((b → r) → r)

chain sA aTosB = \k →        — k :: (b → r)
  sA                         — needs a (a → r)
  (\a →
    aTosB a                  — needs a (b → r)
    k)                       — k is a (b → r)!
```

---

[1]From The Haskell Wikibook, although there it is called `chainCPS`

Making **suspendVolume** is much easier now

```
— suspendArea r = \k → k (areaCircle r)
— Flip arguments to chain easier.
suspendExtrude' :: Height → Area → (Volume → r) → r
suspendExtrude' h a = suspendExtrude a h

suspendVolume'' :: Radius → Height → (Volume → r) → r
suspendVolume'' r h = suspendArea r `chain`
                        suspendExtrude' h

— chain for us looks like
chain :: ((Area → r) → r) → (Area → ((Volume → r) → r)) →
        ((Volume → r) → r)
```

I smell a monadic burrito (link)

cpsify (or (£)) is acting like `return`, and `chain` is acting like `bind` ((≫=)). Lets make a type for these suspended computations then.

```
— Already defined in Control.Monad.Trans.Cont
newtype Cont r a = Cont {runCont :: (a → r) → r}
```

I smell a monadic burrito (link)

cpsify (or (£)) is acting like `return`, and `chain` is acting like `bind`
((>>=)). Lets make a type for these suspended computations then.

```
— Already defined in Control.Monad.Trans.Cont
newtype Cont r a = Cont {runCont :: (a → r) → r}

instance Monad (Cont r) where
    return :: a → Cont r a
    return x = Cont (\k → k x)
 — cpsify x =        \k → k x
```

I smell a monadic burrito (link)

`cpsify` (or `(£)`) is acting like `return`, and `chain` is acting like `bind` (`(>>=)`). Lets make a type for these suspended computations then.

```haskell
— Already defined in Control.Monad.Trans.Cont
newtype Cont r a = Cont {runCont :: (a → r) → r}

instance Monad (Cont r) where
    return :: a → Cont r a
    return x = Cont (\k → k x)
 — cpsify x =        \k → k x

    (>>=) :: Cont r a → (a → Cont r b) → Cont r b
    m >>= f = Cont
        (\k → runCont m (\a → runCont (f a) k))
 — chain m f = \k →  m (\a →          (f a) k)
```

The **Cont** monad

do sweetness is now possible

Cont is a monad, which means we can get some nice **do** syntactic sugar.

```
— Control.Monad.Trans.Cont exports cont, not Cont
cont = Cont

suspendAreaCont :: Radius → Cont r Area
suspendAreaCont r = cont (\k → k (areaCircle r))

suspendExtrudeCont :: Height → Area → Cont r Volume
suspendExtrudeCont h a = cont (\k → k (a * h))

suspendVolumeCont :: Radius → Height → Cont r Volume
suspendVolumeCont r h = do
  area ← suspendAreaCont r
  suspendExtrudeCont h area
```

What happens if we forget the k?

The **k** in the prior functions represented "the rest of the computation". So what happens when we forget the k?

```
— I wonder where the r went....
suspendAreaOops :: Double → Cont Double Double
suspendAreaOops r = cont (\_ → areaCircle r)

suspendVolumeOops :: Radius → Height → Cont Double Double
suspendVolumeOops r h = do
  area ← suspendAreaOops r
  suspendExtrudeCont h area
```

What is the result of the following?

```
oops :: Double
oops = runCont (suspendVolumeOops 1 2) id
```

What happens if we forget the k?

The **k** in the prior functions represented "the rest of the computation". So what happens when we forget the k?

```
— I wonder where the r went....
suspendAreaOops :: Double → Cont Double Double
suspendAreaOops r = cont (\_ → areaCircle r)

suspendVolumeOops :: Radius → Height → Cont Double Double
suspendVolumeOops r h = do
  area ← suspendAreaOops r
  suspendExtrudeCont h area
```

What is the result of the following?

```
oops :: Double
oops = runCont (suspendVolumeOops 1 2) id

— oops == π
— (wait, that is not (π * 1²) * 2 = 2 * π)
```

The **k** meant "keep going!"

Normally we have this kind of scenario.

```
area → extrude → suspended computation
—         or (area → ((extrude → suspended computation))
—  k is "conceptually" ^————————————^
```

When we did not call **k**, we forgot to keep going forward and instead did this.

```
area → suspended computation
```

What happens if we double the use of k?

```
suspendAreaDbl :: Radius → Cont [r] Area
suspendAreaDbl r = cont (\k → k (areaCircle r) ++
                                k (areaCircle (r + 2)))


suspendVolumeDbl :: Radius → Height → Cont [r] Volume
suspendVolumeDbl r h = do
  area ← suspendAreaDbl r
  suspendExtrudeCont h area
```

What is the result of the following?

```
dbl :: [Volume]
dbl = runCont (suspendVolumeDbl 1 2) return
```

What happens if we double the use of k?

```
suspendAreaDbl :: Radius → Cont [r] Area
suspendAreaDbl r = cont (\k → k (areaCircle r) ++
                                k (areaCircle (r + 2)))


suspendVolumeDbl :: Radius → Height → Cont [r] Volume
suspendVolumeDbl r h = do
  area ← suspendAreaDbl r
  suspendExtrudeCont h area
```

What is the result of the following?

```
dbl :: [Volume]
dbl = runCont (suspendVolumeDbl 1 2) return
```

— dbl = [6.283185307179586,56.548667764616276]
— dbl = [2π] ++ [18π]

We execute the remaining computation twice, and throw the result
into \k → k (areaCircle r) ++ k (areaCircle (r + 2))

What do we currently have?

We have a few tools to make suspended computations.

- ▶ A way to create suspended computations (`cpsify` or `return`)
- ▶ A way to chain those computations (`chain` or (`>>=`))
- ▶ All of this wrapped in a type `Cont r a`

In addition, we have seen how to break out of the standard control flow.

- ▶ Forget `k`: do not process any more lines in the continuation.
- ▶ Use `k` twice: repeat the continuation twice, combine results.

Call with Current Continuation (`callCC`)

What else could we want?

- We have a way to stop what we were doing (forget `k`)
- We have a way to "branch" (use `k`) twice
- We didn't talk about it but if `k :: (r → r)` then we can run a function on itself again (`\k → k (k (...))`)

Once we are in `Cont`, using `k` is somewhat tedious (it needs to be explicitly brought out).

An eject button would be nice

What if we brought out **k** only when we needed it? Where k is "skip whatever else we were going to do in this continuation and keep going with the next one".

Ejecting with `callCC`

There is a function that does this called "call with current continuation", or `callCC` in Haskell.

```haskell
callCC :: ((a → Cont r b) → Cont r a) → Cont r a

calc :: Double → Double → String
calc a b = (`runCont` id) $ do

  ans ← callCC $ \eject → do
          c ← return $ a*2 + b + 3
          when (c == 0) (eject "oops dividing by 0")
          d ← return $ a + b
          return $ show (d / c)

  return $ "Answer: " ++ ans
```

Ejecting with `callCC`

There is a function that does this called "call with current continuation", or `callCC` in Haskell.

```haskell
callCC :: ((a → Cont r b) → Cont r a) → Cont r a

calc :: Double → Double → String
calc a b = (`runCont` id) $ do

  —   |— Calling `eject` is the same as
  —   v   `ans ← return "oops dividing by 0"`
  ans ← callCC $ \eject → do
          c ← return $ a*2 + b + 3
          when (c == 0) (eject "oops dividing by 0")
          d ← return $ a + b
          return $ show (d / c)

  return $ "Answer: " ++ ans
```

Ejecting with `callCC`

There is a function that does this called "call with current continuation", or `callCC` in Haskell.

```haskell
callCC :: ((a → Cont r b) → Cont r a) → Cont r a

calc :: Double → Double → String
calc a b = ('runCont' id) $ do

  —   |— 'eject' is often called 'k'; 'k' is the same as
  —   v   'ans ← return "oops dividing by 0"'
  ans ← callCC $ \k → do
          c ← return $ a*2 + b + 3
          —              k :: String → Cont String ()
          when (c == 0) (k "oops dividing by 0")
          d ← return $ a + b
          — String → Cont String String
          return $ show (d / c)

  return $ "Answer: " ++ ans
```

Guess the answers

What happens when we call `calc`

```
calc :: Double → Double → String
calc a b = (`runCont` id) $ do

  ans ← callCC $ \k → do
          c ← return $ a*2 + b + 3
          when (c == 0) (k "oops dividing by 0")
          d ← return $ a + b
          return $ show (d / c)

  return $ "Answer: " ++ ans

calc_1 = calc 1 2     — ?
```

Guess the answers

What happens when we call `calc`

```haskell
calc :: Double → Double → String
calc a b = (`runCont` id) $ do

  ans ← callCC $ \k → do
          c ← return $ a*2 + b + 3
          when (c == 0) (k "oops dividing by 0")
          d ← return $ a + b
          return $ show (d / c)

  return $ "Answer: " ++ ans

calc_1 = calc 1 2     — "Answer: 0.42857142857142855"
```

Guess the answers

What happens when we call `calc`

```haskell
calc :: Double → Double → String
calc a b = ('runCont' id) $ do

  ans ← callCC $ \k → do
          c ← return $ a*2 + b + 3
          when (c == 0) (k "oops dividing by 0")
          d ← return $ a + b
          return $ show (d / c)

  return $ "Answer: " ++ ans

calc_1 = calc 1 2    — "Answer: 0.42857142857142855"
calc_2 = calc 0 (−3) — ?
```

Guess the answers

What happens when we call `calc`

```haskell
calc :: Double → Double → String
calc a b = (`runCont` id) $ do

  ans ← callCC $ \k → do
          c ← return $ a*2 + b + 3
          when (c == 0) (k "oops dividing by 0")
          d ← return $ a + b
          return $ show (d / c)

  return $ "Answer: " ++ ans

calc1 :: String
calc1 = calc 1 2    — "Answer: 0.42857142857142855"

calc2 :: String
calc2 = calc 0 (−3) — "Answer: oops dividing by 0"
```

Passing k around

Shortcutting is useful when we realize that *there is nothing more to do.* For example, the product of a list containing 0 is 0.[2]

```haskell
prod :: (Eq a, Num a) ⇒ [a] → a
prod l = ('runCont' id) $ callCC (\k → loop k l)
 where
   loop _ []     = return 1
   loop k (0:_) = k 0
   loop k (x:xs) = do
       n ← loop k xs
       return (n*x)
```

---
[2]From the Haskell Wiki delcont page

Passing k around

Shortcutting is useful when we realize that *there is nothing more to do.* For example, the product of a list containing 0 is 0.[3]

```
prod :: (Eq a, Num a) ⇒ [a] → a
prod l = ('runCont' id) $ callCC (\k → loop k l)
 where
   loop _ []     = return 1
   loop k (0:_)  = k 0
   loop k (x:xs) = do
       n ← loop k xs
       return (n*x)

prod [1, 2, 3, 4]    — ?
```

[3]From the Haskell Wiki delcont page

Passing k around

Shortcutting is useful when we realize that *there is nothing more to do.* For example, the product of a list containing 0 is 0.[4]

```
prod :: (Eq a, Num a) ⇒ [a] → a
prod l = ('runCont' id) $ callCC (\k → loop k l)
 where
   loop _ []    = return 1
   loop k (0:_) = k 0
   loop k (x:xs) = do
       n ← loop k xs
       return (n*x)

prod [1, 2, 3, 4]    — 24
```

[4]From the Haskell Wiki delcont page

Passing k around

Shortcutting is useful when we realize that *there is nothing more to do.* For example, the product of a list containing 0 is 0.[5]

```haskell
prod :: (Eq a, Num a) ⇒ [a] → a
prod l = (`runCont` id) $ callCC (\k → loop k l)
 where
   loop _ []    = return 1
   loop k (0:_) = k 0
   loop k (x:xs) = do
       n ← loop k xs
       return (n*x)

prod [1, 2, 3, 4]    — 24
prod [1, 2, 0, 3, 4] — ?
```

[5]From the Haskell Wiki delcont page

Passing k around

Shortcutting is useful when we realize that *there is nothing more to do.* For example, the product of a list containing `0` is `0`.[6]

```haskell
prod :: (Eq a, Num a) ⇒ [a] → a
prod l = (`runCont` id) $ callCC (\k → loop k l)
 where
   loop _ []     = return 1
   loop k (0:_)  = k 0
   loop k (x:xs) = do
       n ← loop k xs
       return (n*x)

prod [1, 2, 3, 4]    — 24
prod [1, 2, 0, 3, 4] — 0 by shortcut
```

What we have done so far

- We have a way to stop what we were doing (forget **k**)
- We have a way to "branch" (use **k**) twice
- We didn't talk about it but if **k :: (r → r)** then we can run a function on itself again (**\k → k (k (...))**)
- We can eject from a function (`callCC`)

What you can build with continuations

Lots of different control structures, including

- Exceptions
- Coroutines
- Generators/Iterators

Thanks

References I

All bullets are clockable links.

Tutorials

- The Mother of all Monads
- Continuation Passing Style: Haskell Wikibooks
- Continuations in Haskell
- Haskell for All: The Continuation Monad (on modular development using continuations and sum types)
- Understanding Continuations (this one can be a bit tricky for new Haskell users).
- Understanding the callCC example in above

CallCC

- call/cc implementation StackOverflow
- Example callCC usage (in Scheme)

References II

- ▶ callCC patterns (in Scheme)

Real World Use

- ▶ CPS is great! CPS is terrible! (on the use in attoparsec)

Delimited Continuations

- ▶ Delimited Continuations in Haskell tutorial
- ▶ Example code for delimited continuations in haskell
- ▶ Library CC-delcont examples
- ▶ Delimited Continuations and co-monads video

Extra Code

Set Example

We can define sets as a list with a careful insertion function.

```haskell
newtype Set a = Set {getSet :: [a]} deriving (Show)

empty :: Set a
empty = Set []

insert :: Eq a ⇒ Set a → a → Set a
insert   (Set [])    x = Set [x]
insert s@(Set (y:ys)) x = if x == y
                          then s
                          else Set $ y : getSet (insert (Set ys) x)
```

What is problematic when **insert** is given a value already in the set?

(Modified problem from Okasaki's "Purely Functional Data Structures", problem 2.3)

insert allocates unneeded nodes

Given a set **s**.

s = a **:** b **:** c **:** d **:** e **:** f **:** **[]**

insert allocates unneeded nodes

Given a set s.

```
s = a : b : c : d : e : f : []
              ^——————^
                     |
insert s d = a : b : c : |
```

insert allocates a (mostly) new Set.

A "smarter" insert

Let's use a continuation to return **s** when **x** is in **s**.

```
insert' :: Eq a ⇒ Set a → a → Set a
insert' s' x = ('runCont' id) $ callCC (\k → insertShortcut k s')
    where
        insertShortcut _ (Set [])    = return $ Set [x]
        insertShortcut k (Set (y:ys)) =
            if x == y
                then k s'
                else insertShortcut k ⟹ setcons y $ Set ys

        setcons y (Set ys) = return $ Set (y : ys)
```

insert' uses less memory

Given a set **s**.

```
s = a : b : c : d : e : f : []
    ^—————————————^
                 |
insert' s d = |
```

insert' returns the same **s** if an element **x** is in **s**.

```haskell
{——————————–
- Testing with IO -
——————————–}

io1 :: Int → ContT () IO String
io1 x = ContT $ \k → do
    putStrLn "io1 start"
    k $ show x
    putStrLn "io1 end"


io2 :: String → ContT () IO String
io2 s = ContT $ \k → do
    putStrLn "io2 start"
    k (s ++ " io2")
    putStrLn "io2 end"


ioExample :: Int → ContT () IO String
ioExample = io1 ⟹ io2
```

From the "Understanding Continuations" FP article.

```
fpExample :: IO ()
fpExample = flip runContT return $ do
    lift $ putStrLn "alpha"
    (k, num) ← callCC $ \k → let f x = k (f, x)
                             in return (f, 0 :: Integer)
    lift $ putStrLn "beta"        — k
    lift $ putStrLn "gamma"       — j
    if num < 5
        then void (k (num + 1))
        else lift $ print num     — l
```