Continuations From the Ground Up

06 June 2017

It's difficult to learn functional programming without hearing about continuations. Often they're mentioned while talking about boosting the performance of pure functional code, sometimes there's talk of control flow, and occasionally with 'time-travel' thrown in there to make it all seem more obscure. It's all true, but let's start from the beginning.

This post was generated from a Literate Haskell file using Pandoc, so you can load it up into GHCI and play around if you want.

Continuation Passing Style

```
module CPS where

import Control.Applicative
import Control.Monad.Trans.Class
import Data.IORef
import Data.Maybe

import qualified Data.Map as M
import qualified System.Exit as E
```

The main idea of this style is that the called function has control over how its return value is used. Usually, the caller will pass a function that tells the callee how to use its return value. Here's what that looks like:

```
add :: Num a => a -> a -> (a -> r) -> r
add a b = \k -> k (a + b)
```

`add` takes two numbers, plus a function that will take the result and do something (returning an unknown answer), then pass the result to this function. We call this 'extra function' a *continuation* because it specifies how the program should *continue*.

It's possible to write any program using this style. I'm not going to prove it. As a challenge, let's restrict ourselves to write every function this way, with two exceptions:

```
exitSuccess :: a -> IO ()
exitSuccess _ = E.exitSuccess

exitFailure :: Int -> IO ()
exitFailure = E.exitWith . E.ExitFailure
```

`exitSuccess` and `exitFailure` do not take a continuation, because the program always ends when they are called.

Let's define `mul` and `dvd`:

```
mul :: Num a => a -> a -> (a -> r) -> r
mul a b = \k -> k (a * b)

dvd :: Fractional a => a -> a -> (a -> r) -> r
dvd a b = \k -> k (a / b)
```

Now we can write some programs using this style.

```
-- Exits with status code 5
prog_1 :: IO ()
prog_1 = add 2 3 exitFailure

-- Exits successfully after multiplying 10 by 10
prog_2 :: IO ()
prog_2 = mul 10 10 exitSuccess

-- Exits with status code (2 + 3) * 5 = 25
prog_3 :: IO ()
prog_3 = add 2 3 (\two_plus_three -> mul two_plus_three 5 exitFailure)
```

We can factor out the continuation to make our program more modular:

```
-- Equivalent to \k -> k ((2 + 3) * 5)
prog_4 :: (Int -> r) -> r
prog_4 = \k -> add 2 3 (\two_plus_three -> mul two_plus_three 5 k)

-- Equivalent to \k -> k ((2 + 3) * 5 + 5)
```

```
prog_5 :: (Int -> r) -> r
prog_5 = \k -> prog_4 (\res -> add res 5 k)
```

In these kind of definitions, we'll call the `k` the *current continuation* to stand for *how the program will (currently) continue execution*.

Here's a more complex expression:

```
-- (2 + 3) * (7 + 9) + 5
prog_6 :: Num a => (a -> r) -> r
prog_6 = \k ->
  add 2 3 (\five ->
    add 7 9 (\sixteen ->
      mul five sixteen (\eighty ->
        add eighty 5 k)))
```

In translating programs to continuation passing style, we transform a *tree* of computations into a *sequence* of computations. In doing so, we have *reified* the flow of the program. We now have a data structure in memory that represents the computations that make up the program. In this case, the data structure is a lot like a linked list- there is a head: 'the computation that will be performed next', and a tail: 'the computations that will be performed on the result'. It's this ability to represent the flow of the program as a data structure that sets CPS programs apart from regular programs, which we will see later.

Continuation Passing is Monadic

Right now, writing CPS programs in Haskell is too verbose. Fortunately there are some familiar abstractions that will make it elegant:

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

add' :: Num a => a -> a -> Cont r a
add' a b = Cont $ add a b

mul' :: Num a => a -> a -> Cont r a
mul' a b = Cont $ mul a b

dvd' :: Num a => a -> a -> Cont r a
dvd' a b = Cont $ dvd a b

instance Functor (Cont r) where
  fmap f c = Cont $ \k -> runCont c (k . f)

instance Applicative (Cont r) where
  pure a = Cont ($ a)
  cf <*> ca = Cont $ \k -> runCont cf (\f -> runCont ca (\a -> k (f a)))
```

```
instance Monad (Cont r) where
  ca >>= f = Cont $ \k -> runCont ca (\a -> runCont (f a) k)
```

It turns out that the return type of these CPS programs, `(a -> r) -> r`, is a Monad. If you don't understand these implementations, meditate on them until you do. Here some hand-wave-y English explanations that may help:

Functor

`fmap`: Continue with the result of `c` by changing the result from an `a` to a `b` then sending that result to the current continuation.

Applicative

`pure`: Send an `a` to the current continuation

`<*>`: Continue with the result of `cf` by continuing with the result of `ca` by sending (the result of `cf`) applied to (the result of `ca`) to the current continuation.

Monad

`>>=`: Continue with the result of `ca` by applying it to `f` and passing the current continuation on to the value `f` returned.

So now we can rewrite our previous verbose example:

```
prog_6' :: Cont r Int
prog_6' = do
  five <- add' 2 3
  sixteen <- add' 7 9
  eighty <- mul' five sixteen
  add' eighty 5
```

and run it:

```
prog_7 :: IO ()
prog_7 = runCont prog_6' exitSuccess
```

callCC

Consider the following CPS program:

```
prog_8 :: (Eq a, Fractional a) => a -> a -> a -> (Maybe a -> r) -> r
prog_8 a b c = \k ->
  add b c
    (\b_plus_c ->
      if b_plus_c == 0
        then k Nothing
        else dvd a b_plus_c (k . Just))
```

It adds `b` to `c`, then if `b + c` is zero, sends `Nothing` to the current continuation, otherwise divides `a` by `b + c` then continues by wrapping that in a `Just` and sending the `Just` result to the current continuation.

Because the current continuation is 'how the program will continue with the result of this function', sending a result to the current continuation early cause the function to *exit early*. In this sense, it's a bit like like a `jmp` or a `goto`.

It is conceivable that somehow we can write a program like this using the `Cont` monad. This is where `callCC` comes in.

`callCC` stands for 'call with current continuation', and is the way we're going to bring the current continuation into scope when writing CPS programs. Here's an example of how the previous code snippet should look using `callCC`:

```
prog_8' :: (Eq a, Fractional a) => a -> a -> a -> Cont r (Maybe a)
prog_8' a b c = callCC $
  \k -> do
    b_plus_c <- add' b c
    if b_plus_c == 0
      then k Nothing
      else fmap Just $ dvd' a b_plus_c
```

Here's how `callCC` is defined:

```
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
callCC f = Cont $ \k -> runCont (f (\a -> Cont $ const (k a))) k
```

We can see that the current continuation is permanently captured when it is used in the function passed to `f`, but it is also used when running the final result of `f`. So `k` might be called somewhere inside `f`, causing `f` to exit early, or it might not, in which case `k` is guaranteed to be called after `f` has finished.

Earlier I said that invoking the current continuation earlier is like jumping. This is a lot easier to show now that we can use it in our `Cont` monad. Calling the continuation provided by `callCC` will jump the program execution to immediately after the call to `callCC`, and set the result of the `callCC` continuation to the argument that was passed to the current continuation.

```
prog_9 = do
  five <- add' 2 3
  res <- callCC $ \k ->
    -- current continuation is never used, so `callCC` is redundant
    mul' 4 5
  -- `res` = 20
  add' five res

prog_10 = do
  five <- add' 2 3
  res <- callCC $ \k -> do
    k 5
    mul' 4 5 -- this computation is never run
  -- program jumps to here, `res` = 5
  add' five res

prog_11 = do
  five <- add' 2 3
  res <- callCC $ \k -> do
    if five > 10
      then k 10 -- branch A
      else mul' 4 5 -- branch B
  -- if branch A was reached, `res` = 10
  -- if branch B was reached, `res` = 20
  add' five res
```

Another level of abstraction

We can also embed arbitrary effects in the return type of `Cont`. In other words, we can create a monad transformer.

```
newtype ContT r m a = ContT { runContT :: (a -> m r) -> m r }

callCC' :: ((a -> ContT r m b) -> ContT r m a) -> ContT r m a
callCC' f = ContT $ \k -> runContT (f (\a -> ContT $ const (k a))) k

instance Functor (ContT r m) where
  fmap f c = ContT $ \k -> runContT c (k . f)

instance Applicative (ContT r m) where
  pure a = ContT ($ a)
  cf <*> ca = ContT $ \k -> runContT cf (\f -> runContT ca (\a -> k (f a)))

instance Monad (ContT r m) where
  ca >>= f = ContT $ \k -> runContT ca (\a -> runContT (f a) k)
```

```
instance MonadTrans (ContT r) where
  lift ma = ContT $ \k -> ma >>= k
```

Notice that the `Functor`, `Applicative` and `Monad` instances for `ContT r m` don't place any constraints on the `m`. This means that any type constructor of kind `(* -> *)` can be in the `m` position. The `MonadTrans` instance, however, does require `m` is a monad. It's a very simple definition- the result of running the lifted action is piped into the current continuation using `>>=`.

Now that we have a fully-featured CPS monad, we can start doing magic.

The future, the past and alternate timelines

The continuation that `callCC` provides access to is the current future of program execution as a single function. That's why this program-as-a-linear-sequence is so powerful. If you could save the current continuation and call it at a later time somewhere else in your (CPS) program, it would jump 'back in time' to the point after that particular `callCC`.

To demonstrate this, and end with a bang, here's a simple boolean SAT solver.

```
-- Language of boolean expressions
data Expr
  = Implies Expr Expr
  | Iff Expr Expr
  | And Expr Expr
  | Or Expr Expr
  | Not Expr
  | Val Bool
  | Var String
  deriving (Eq, Show)

-- Reduces a boolean expression to normal form, substituting variables
-- where possible. There are also some equivalences that are necessary to
get
-- the SAT solver working e.g. Not (Not x) = x (I said it was a simple one!)
eval
  :: M.Map String Expr -- ^ Bound variables
  -> Expr -- ^ Input expression
  -> Expr
eval env expr =
  case expr of
    Implies p q -> eval env $ Or (Not p) q
    Iff p q -> eval env $ Or (And p q) (And (Not p) (Not q))
    And a b ->
      case (eval env a, eval env b) of
        (Val False, _) -> Val False
        (_, Val False) -> Val False
        (Val True, b') -> b'
        (a', Val True) -> a'
        (a', b') -> And a' b'
    Or a b ->
      case (eval env a, eval env b) of
        (Val True, _) -> Val True
        (_, Val True) -> Val True
        (Val False, b') -> b'
        (a', Val False) -> a'
        (a', b')
          | a' == eval env (Not b') -> Val True
          | otherwise -> Or a' b'
    Not a ->
      case eval env a of
        Val True -> Val False
        Val False -> Val True
        Not a' -> a'
        a' -> Not a'
    Val b -> Val b
    Var name -> fromMaybe (Var name) (M.lookup name env)

-- Returns `Nothing` if the expression is not satisfiable
-- If the epxression is satisfiable returns `Just mapping` with a valid
-- variable mapping
sat :: Expr -> ContT r IO (Maybe (M.Map String Expr))
sat expr = do
  -- A stack of continuations
  try_next_ref <- lift $ newIORef []

  callCC' $ \exit -> do
    -- Run `go` after reducing the expression to normal form without any
    -- variable values
    res <- go (eval M.empty expr) try_next_ref exit
    case res of
      -- If there was a failure, backtrack and try again
      Nothing -> backtrack try_next_ref exit
      Just vars -> case eval vars expr of
        -- If the expression evaluates to true with the results of `go`,
finish
        Val True -> exit res
        -- Otherwise, backtrack and try again
        _ -> backtrack try_next_ref exit

  where
    -- To backtrack: try to pop a continuation from the stack. If there are
    -- none left, exit with failure. If there is a continuation then enter
it.
    backtrack try_next_ref exit = do
      try_next <- lift $ readIORef try_next_ref
      case try_next of
        [] -> exit Nothing
        next:rest -> do
          lift $ writeIORef try_next_ref rest
          next

    -- It's a tree traversal, but with some twists
    go expr try_next_ref exit =
      case expr of
        -- Twist 1: When we encounter a variable, we first continue as if
it's
        -- true, but also push a continuation on the stack where it is set
to false
        Var name -> do
```

```
          res <- callCC' $ \k -> do
            lift $ modifyIORef try_next_ref (k (Val False) :)
            pure $ Val True
          -- When this program is first run, `res` = True. But if we pop and
          -- enter the result of `k (Val False)`, we would end up back here
          -- again, with `res` = False
          pure $ Just (M.singleton name res)
        Val b -> pure $ if b then Just M.empty else Nothing
        -- Twist 2: When we get to an Or, only one of the sides needs to be
        -- satisfied. So we first continue by checking the left side, but
also
        -- push a continuation where we check the right side instead.
        Or a b -> do
          side <- callCC' $ \k -> do
            lift $ modifyIORef try_next_ref (k b :)
            pure a
          -- Similar to the `Var` example. First ruvn, `side` = a. But if
later
          -- we enter the saved continuation then we will return to this
point
          -- in the program with `side` = b
          go side try_next_ref exit
        And a b -> do
          a_res <- go a try_next_ref exit
          b_res <- go b try_next_ref exit
          pure $ liftA2 M.union a_res b_res

        Not a -> go a try_next_ref exit

        _ -> go (eval M.empty expr) try_next_ref exit
```

The solver sets all the variables to `True`, and if the full expression evaluates to `False` it flips one to `False` and automatically re-evaluates the expression, repeating the process untill either it finally evaluates to `True` or all possible combinations of boolean values have been tested. It's not efficient, but it's a wonderful illustration of the elegance that CPS enables.

How to delete old NixOS boot configurations - Previous | Newest

tags: programming haskell

Isaac Elliott GitHub Gmail