Functional Pearl: Algebraic composition of impure asynchronous effects

We reformulate the continuation monad in a more simple way that makes it more intuitive to use. The choice is not without some trade-offs, which we justify here. Instead of using the raw power of callCC, we use it to define less powerful but lawful, orthogonal primitives that implement effects like reactive, parallelism, concurrency, exceptions, backtracking, threading and in general any asynchronous effect that is non algebraically composable under other monads.

With and these primitives and the usage of standard monadic, applicative alternative, monoidal combinators it is possible to create infinitely composable expressions that implement a mix of these effects.

CCS Concepts:•**Theory of computation** → **Parallel computing models; Functional constructs;** *Concurrent algorithms;*•**Software and its engineering** → *Publish-subscribe / event-based architectures; Interoperability;*•**Computing methodologies** → *Vector / streaming algorithms;*

Additional Key Words and Phrases: Haskell, reactive, parallelism, concurrency, exceptions, Backtracking, non-determinism, threading

ACM Reference Format:
Anonymous *ACM Trans. Program. Lang. Syst.* V, N, Article A (January YYYY), 22 pages.
DOI: 0000001.0000001

1. INTRODUCTION

This is an unusual submission for a candidate of a ”functional pearl”. It uses IO calls, it forks threads and uses *unsafeCoerce* (although that is arguably justified). But the relative conciseness and the problem that it tackles, composition of hard, impure effects like threading, concurrency, event handling, exceptions, reactive and streaming using a single monad, the continuation monad, makes it a worthy candidacy in my humble opinion.

When a program need to use several threads or there are callbacks, exceptions, asynchronous communications etc it is very difficult or impossible, with current techniques, to codify a program as if it were a single expression. The continuation monad can potentially solve the problem but it has not responded to what was expected in practical terms. There was a great effort, theoretical and practical a few decades ago to use continuations but now it has been put aside in favor of simpler solutions to solve the problem of composition in presence of impure asynchronous effects.

These solutions like futures (async-await) or promises, are imperfect solutions to alleviate that problem. But the fact is that the continuation monad could theoretically solve it completely. Why it is not done? The reason is that this monad has a complex and non intuitive formulation. Furthermore, when toy examples of continuations are made, it is to demonstrate their power by making strange effects that are not easy to understand because they do not conform to educational reasoning or the effects are not orthogonal nor intuitive, so that they can't be combined to produce easily predictable result.

DOI: 0000001.0000001

First, we will highlight some problems of the continuation monad, then we will define a simpler form of continuation monad, with some instances that allows asynchronous effects. Then we will define some orthogonal and composable primitives that implement the above mentioned effects, we will create programs that compose these primitives with standard Haskell binary operators to create programs that have a mix of these effects. Finally we will compose these programs to create a single demo.

There is a demo [Anonymous 2018] online which run all code snippets contained in this paper.

1.1. Enter the monad

The Monad class is defined as:

```
class Monad m where
   return :: m a
   (>>=) :: m a -> (a -> m b) -> m b -- bind
```

The second term of *(>>=)* is a lambda. It could be considered also a kind of continuation. It could be seen also as a callback: The bind operator can be read as this: when *'m a'* (the first term) is executed, apply the second term as a callback, which will receive the result of the first term.

```
do x <- mx
   y <- my
   z
```

... is equivalent to:

```
do x <- mx
   do y <- my
      z
```

... desugars to:

```
mx >>= (\x ->
              my >>= (\y ->
                             z ))
```

1.2. Enter the continuation monad

However we want to define *mx my* etc as computations that know their continuations, so they receive their continuations as a parameter. This allows the creation of primitives that modify the execution flow in ways not permitted by other monads.

This is the original Continuation Monad. In this monad, each computation *'mx'*, *'my'* is a lambda whose parameter *a ->m r* is the continuation *c*, in which *a* is the value returned by the previous term.

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

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

```
instance Applicative (Cont r m) where
    pure x  = Cont (\c ->c x) -- Cont ($ x)
    f <*> v = Cont $ \ c -> runCont f $ \ g -> runCont v $ \t -> c $ g t

instance Monad (Cont r m) where
  return x = Cont (\c -> c x) -- Cont ($ x)
  -- (>>=) :: Cont r m a -> (a -> Cont r m b) -> Cont r m b
  m >>= k  = Cont $ \ c -> runCont m $ \ x -> runCont ( k x ) c
```

1.3. Problems of the Continuation monad

However the Cont monad is weird. It needs an extra parameter *r* which is the final result. Its type depend on a final continuation that is outside of the expression itself, since

```
Cont r m b === Cont ((b -> m r) -> m r)
```

does not materialize in a result. It needs a final lambda 'b ->m r' to produce *'m r'*. Usually the computation *(return . id)* is used to get it. this leads to type coercions when the computation is used as part of more complex expressions. Continuations can only be modeled fully within Indexed monads [Wadler 1994]. The symptom of that problem is the extra parameter *r*.

The monad below eliminates the extra parameter since, by construction, the parameter *'b'* of the continuation of the second term is of the type of the result of the first term of the bind, so we can coerce types with confidence and stay within a normal monad instead of an indexed monad.

1.4. A simpler, but controversial, although effective continuation monad

We define a dynamic parameter in the continuation, which receives the changing types and values managed by the monadic computation when it is applied to different terms.

```
type Dyn= ()
```

The new data definition for the new *Cont* monad:

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

Now the type of the value returned by this kind of continuation is of type a which appears in the data definition. there is no need of a last step outside of the expression.

As a side note, all the code of this paper could be made with the standard continuation monad unchanged, but it need coercion in the application code. This need of coercion in any case is the motivation for the creation of a simpler monad which includes the necessary coercion inside and is clean outside.

For various purposes, we need some state being carried out by the monad; Some primitives, specially backtracking need them. It is also convenient to define an alternative instance as we will see. The details of the state structure will be justified later.

```
type SData= ()

data Stat = Stat
```

```
  { mfData      :: M.Map TypeRep SData
  , emptyOut    :: Bool
  } deriving Typeable
```

For now, it is enough to say that *mfData* will encode a map of data values indexed by types. and *emptyOut* will be used by the alternative instance. The monad will carry on an state. So it uses the state monad transformer

```
type StateIO = StateT Stat IO
```

1.5. Class instances

Let's load our final monad with the payload state we defined above and define the monad instance

```
type ContIO  = Cont  StateIO
```

to increase readability of the type erasure necessary, i define convenient synonymous:

```
ety :: a -> b
ety=   dontWorry
tdyn :: a -> Dyn
tdyn=  dontWorry
fdyn :: Dyn -> a
fdyn = dontWorry

dontWorry= unsafeCoerce


instance Monad ContIO  where
    return  = pure

    m >>= k  = Cont $ \c -> ety $ runCont m (\x -> ety $
                      runCont ( k $ fdyn x) c)
```

The instance, type coercion apart, is identical to the standard continuation. But.. now we know that the result is the result of the second term. A further lambda/continuation/callback is not needed.

More standard instances coming. Nothing special to say. *callCC* is also standard, with the exception of type coercion added:

```
instance MonadState  Stat ContIO where
    get= lift get
    put= lift . put

instance MonadTrans Cont  where
    lift m = Cont ((ety m) >>=)
```

```
instance MonadIO ContIO  where
    liftIO = lift . liftIO


instance Functor (Cont  m) where
    fmap f m = Cont $ \c -> ety $ runCont m $ \ x -> ety c $ f $ fdyn x


callCC :: ((a -> Cont  m b) -> Cont  m a) -> Cont  m a
callCC f = Cont $ \ c -> runCont (f (\ x -> Cont $
                                      \ _ ->  ety $ c $ tdyn x)) c
```

Now, the runner of the monad with an state *st*. Note that we add a final computation (return . id) but in this case it is not arbitrary, but enforced by the return type of the monadic computation.

```
runContState :: Stat -> ContIO  a ->  IO ( a, Stat)
runContState st t= runStateT  (runcst t) st
  where
  runcst :: ContIO a -> StateIO  a
  runcst t=   runCont  t  (return . ety id )
```

Run a Cont computation with a default initial state:

```
runCont :: ContIO a -> IO ( a, Stat)
runCont t = do

    runContState (emptyStat th label childs) t
    where
    emptyStat :: ThreadId -> IORef (LifeCycle, BS.ByteString) ->
                   MVar [Stat] -> Stat
    emptyStat th label childs =
        Stat { mfData     = mempty
               , emptyOut   = False }
```

Now we verify that executing the continuation in *callCC* behaves as a continuation monad.

```
callCCTest= runCont $ do

    r <- callCC $ \ret -> do
        ret  100
        liftIO  $ print "hello"
        return 1
    liftIO $ print r
    liftIO $ print "world"
```

Will produce:

```
> main= callCCTest
```

```
100
"world"
```

1.6. An special alternative instance

Now an unexpected twist to the story: we need the alternative instance for our continuation. It is perfectly possible to let each monadic term to return a *Maybe* value so we can define *empty*, but since we are tacking with exceptions and other impure effects, let's use exceptions for the early finalization of a computation and the execution of a possible alternative computation; It may be more efficient than having an extra constructor.

Moreover adding an additional Maybe constructor would duplicate the number of lines of the instances and would make definitions longer and boring. I did it, and it is not as elegant.

We need an *Empty* exception which carries out a computation state. this exception can be catched by the alternative computation, which continue the execution.

```
newtype Empty= Empty Stat deriving Typeable
instance Show Empty where show _= "Empty"
instance Exception Empty

instance  Alternative ContIO where
  empty= get >>= \st -> liftIO . throw $ Empty st

  f <|> g= callCC $ \k -> do
      st <- get
      liftIO $ (runContState st (f >>= k)
            `catch` (\(Empty st) -> runContState st (g >>= k) ))
      empty
```

That is the straight definition: invoke *f* and the continuation. if it fails with empty run *g* followed by the continuation. The state *st* is propagated trough. Since exceptions are defined in the *IO* Monad, we need to run the terms naked in IO, using *runContState*, and dress the result again with *liftIO*

However empty exception in the continuation '*cont*' of '*f*' in '*f* >>= *cont*' would trigger the execution of the alternative computation '*g*'. This is not what is needed. we need it only when *empty* is triggered in '*f*'. To do it that way, an state variable *emptyOut* is used to detect when empty is called in the continuation. In that case, the *Empty* exception is ignored and re-thrown.

```
  f <|> g= callCC $ \k -> do
       st <- get
       liftIO $ io st f k `catch` \(Empty st) -> do
                    let c =  emptyOut  st
                    when c $ throw (Empty st)
                    io st g k
       empty

       where
       io st f cont= runContState st{emptyOut=False}  (f >>= cont' )
               where cont' x= do modify $ \st ->st{emptyOut=True} ; cont x
```

1.7. A parallel-and-concurrent-ready applicative instance

From easier to more difficult instances, the applicative is the most complicated one. Firstly, following the continuation monad, with the necessary type coercion, let's define the straight instance:

```
instance Applicative ContIO  where
    pure a  = Cont ($ tdyn a)
    f <*> v = ety $ Cont $ \ k -> ety $ runCont f $ \ g ->
            ety $ runCont v $ \t -> k $ (ety g) t
```

That is the standard definition, translated from the standard Cont monad. But we need to give the opportunity to execute both terms in parallel so we define it as the composition of two alternative computations:

```
    f <*> v =   do
          r1 <- liftIO $ newIORef Nothing
          r2 <- liftIO $ newIORef Nothing
          (fparallel r1 r2)  <|> (vparallel  r1 r2)
```

To allow parallel execution of both terms, two mutable variables store the result of each term. The first executes *f*, the other executes *v*. Each term store his result and inspect if the other has finished. This happens if his mutable variable has a result. If it has not finished, it trows empty and the thread finish. If has finished, evaluate the result and execute the continuation *k* with the result:

```
      where
      fparallel :: IORef (Maybe(a -> b)) -> IORef (Maybe a) -> ContIO b
      fparallel r1 r2= ety $ Cont $ \k  ->
          runCont f $ \g -> do
                (liftIO $ writeIORef r1  $ Just (fdyn  g))
                mt <- liftIO $ readIORef r2
                case mt of
                  Just t -> k $ (fdyn g) t
                  Nothing -> get >>= liftIO . throw . Empty

      vparallel :: IORef (Maybe(a -> b)) -> IORef (Maybe a) -> ContIO b
      vparallel  r1 r2=  ety $ Cont $ \k  ->
          runCont v $ \t ->  do
                 (liftIO $ writeIORef r2 $ Just (fdyn t))
                 mg <- liftIO $ readIORef r1
                 case mg of
                   Nothing -> get >>= liftIO . throw . Empty
                   Just g ->  k $ (ety g) t
```

If we manage to run each term in different threads, we could achieve parallelism and concurrency as we will see below.

Now, the standard monoid definition. Since is defined in terms of Applicative, it allows parallel execution of the terms.

```
instance Monoid a => Monoid (ContIO a) where
  mappend x y = mappend <$> x <*> y
  mempty      = return mempty
```

1.8. Numeric algebra

We define a Num instance in terms of applicative, so it allows parallel execution!

```
instance (Num a, Eq a) => Num (ContIO a) where
    fromInteger = return . fromInteger
    mf + mg     = (+) <$> mf <*> mg
    mf * mg     = (*) <$> mf <*> mg
    negate f    = f >>= return . negate
    abs f       = f >>= return . abs
    signum f    = f >>= return . signum
```

In the same way, for example, a parallel relational algebra could be constructed too.

How is this parallel execution permitted? Because we can create threads that execute continuations. These threads execute all terms in parallel, since all combinators are constructed with Applicative operators which are parallel enabled. We will see that they can be also what can be understood as reactive, that is, they can be activated by events.

1.9. Asynchronous jobs

async execute an *IO* operation, then create a thread and initiates the execution of the continuation of the whole computation within it. after that, it leaves the current thread with *empty* so an alternative computation can use the original thread.

```
async :: IO a -> ContIO a
async io= callCC $ \ret -> do
              st <-  get
              liftIO $ forkIOE $ do
                        runContState st ( liftIO io >>= ret )
                             `catch` exceptBack st
                        return ()
              empty

forkIOE x= forkIO $ (x >> return ())  `catch` \(Empty _)-> return ()
```

exceptBack is used to implement backtracking and exceptions. It will be explained later.

This *async* has no await: the thread created executes all the rest of the computation (unless *empty* stop it). Since threads may die with an empty exception, we need to keep the program running. We block the main thread for that purpose.

```
mexit= unsafePerformIO $ newEmptyMVar
```

```
keep mx= do
    forkIOE $ runCont mx
    takeMVar mexit
```

An example with some more complicated expression, showing that asynchronous and synchronous terms may combine well with the defined operators.

```
combination=  do
    r <-  ( async (threadDelay 10000 >> return "hello ")  <>  return "world")
        <|> return "world2"
    liftIO $ putStrLn  r

main= keep combination
```

Produces:

```
world2
hello world
```

1.10. streaming

For multithreaded streaming, we can do something similar than *'async'* but this time, executing the *IO* operation in a loop. each time it return a value, it creates a new thread that continues the execution.

```
waitEvents :: IO a -> ContIO a
waitEvents io= callCC $ \ret -> do
      st <- get
      loop ret st
      where
      loop ret st=  do
              liftIO $ forkIOE $ do
                        runContState st  (liftIO io >>=  ret )
                           `catch` exceptBack st
                        return ()
              loop ret st
```

This simple example uses *waitEvents* with *getLine* to inject strings as events:

```
testWaitEvents= do
  r <- waitEvents getLine
  liftIO $ putStr "received: " >> print r

main= keep testWaitEvents
```

Will produce

```
> hello
received: "hello"
> world
received: "world"
> ssds
received: "ssds"
```

Asynchronous programs can be combined algebraically with any binary operator. For example, this is an IRC client that uses alternative composition:

```
main = do
  h <- withSocketsDo $ connectTo "irc.freenode.net"
              $ PortNumber $ fromIntegral 6667

  keep $ (waitEvents getLine >>= liftIO . hPutStrLn h) <|>
         (waitEvents (hGetLine h) >>= liftIO . putStrLn )
```

1.11. Threaded non-determinism

And now some threaded, non-deterministic, list-like processing. *choose* executes as many alternative 'async' operation as there are values in a list. So each value is returned to the continuation, which is executed in a different thread. Therefore it executes the rest of the computation for all the values in parallel.

```
choose :: [a] -> ContIO a
choose xs = foldl (<|>) empty $ map (async . return) xs
```

This example test the behaviour of this primitive

```
choosetwo=  do
  r  <- choose [1..3]
  r' <- choose ['a'..'c']
  th <- liftIO  myThreadId
  liftIO $ print (r,r', th)

main= keep choosetwo
```

produces:

```
 (2,'a',ThreadId 79)
 (2,'b',ThreadId 80)
 (1,'a',ThreadId 78)
 (1,'b',ThreadId 82)
 (1,'c',ThreadId 83)
 (2,'c',ThreadId 84)
 (3,'a',ThreadId 85)
 (3,'b',ThreadId 86)
 (3,'c',ThreadId 87)
```

This example return the combinations that fit the pythagoras theorem

```
pythagoras =  do
  x <- choose [1..10]
  y <- choose ([1 .. x] :: [Int])
  z <- choose [1 .. round $ sqrt(fromIntegral $ 2*x*x)]

  guard (x*x + y*y == z*z)
  th <- liftIO  myThreadId
  liftIO $ print (x, y, z, th)

main= keep  pythagoras
```

Produces:

```
 (4,3,5,ThreadId 173)
 (8,6,10,ThreadId 565)
```

1.12. Event handling, reactive

A continuation is a callback. We can ”cheat” a callback handler of a framework by giving it our continuation. So instead of

```
do
   ....
   setCallback ourCallback -- our logic is interrupted here


ourCallback value= do
      foo value;           -- and continues here
      .....
```

Instead of that, now we can write:

```
do
    ....
    value <- react setCallback  (return ())
    foo value                          -- code is not broken
    ...
```

the term *return()* is an additional computation that return something to the event handler. It is usually a void value. But some frameworks assign some meaning to the return to the event handler setter. For example in the Web browsers, the event handler interpret ’true’ as stopping bubbling-up events. The definition of *react* is:

```
react
  :: ((eventdata ->  IO response) -> IO ())
  -> IO  response
  -> ContIO eventdata
react setCallback iob= callCC $ \ret -> do
            st <- get
            liftIO $ setCallback $  \x ->  do
                  runContState st (ret x) `catch` exceptBack st
                  iob
            empty
```

Let's define a framework that use callbacks for handling console input and scheduling jobs depending on the content. Unlike ‘getLine‘ which blocks a thread, this framework would feed different processes without blocking. To do so, we can create a thread which will input from the keyboard. Other threads may set callbacks which this console input thread could call when some input is entered.

```
rcb= unsafePerformIO $ newIORef []

setCallback :: String -> (String ->  IO ()) -> IO ()
setCallback name cb= atomicModifyIORef rcb $ \cbs ->
                         (reverse $ (name,cb) : cbs,())

delCallback name= atomicModifyIORef rcb $ \cbs ->
                         (filter ((/=) name . fst ) cbs,())
```

This is the thread that execute the callbacks in a loop

```
consoleLoop =  do
    x   <- getLine
    mbs <- readIORef rcb
    -- for each string entered, execute all the callbacks
    mapM (\(n,cb) -> cb x `catch` \(Empty _) -> return())  mbs
    consoleLoop
```

reactOption set his continuation as callback using *react*, when the continuation is invoked and the input matches the string *resp* then it returns that value, so further lines in the continuation are executed. Otherwise, *empty* stops from doing further actions.

```
reactOption :: String -> String -> ContIO String
reactOption resp message = do
    liftIO $ do
      putStr "enter "
      putStr resp
      putStr "\t to:"
      putStrLn message
    x <- react (setCallback resp) (return ())
    if  x /= resp then empty else
          return resp
```

This example test the composability of our small framework and the ‘react‘ primitive:

```
mainReact = do
    fork consoleLoop
    r <- (reactOption "hello" "hello")  <|>  (reactOption "world" "world")
    liftIO $ putStr "received: " >> print r

    where
    fork f= (async f >> empty) <|> return()

main= keep mainReact
```

This interactive program produces:

```
enter hello      to:hello
enter world      to:world
> hello
received: "hello"
> world
received: "world"
> hello
received: "hello"
```

1.13. backtracking and exceptions

Exceptions pose another problem for composability specially in long running programs where exceptional conditions should free resources or undo actions before returning to the normal execution flow. There may be a stack of handlers that must be executed to free resources when a computation fails. Here is where backtracking and exceptions are related. Take for example this hopefully auto-explained example:

```
emarket= productNavigation >>= reserve  >>= payment

  where
  reserve book = updateDB1 book >> updateDB2 book

  productNavigation= do
    liftIO $ putStrLn "Navigating the list of products"
    return "book"
```

If payment fails, undoing the reservation involves two cancellations, one in each database. A mechanism using normal exceptions would clutter the code with obscure complications in the ’reserve’ code and would force the coder of this computation to decide what to do next. The code should be broken in pieces too since exception primitives like *catch* do fork the execution flow in two branches, one for the normal flow and another for the exceptional condition. In the other side, not using exceptions by using conditional code in *payment* would force this computation to know about reservation details.

All that is a problem for composability, modularity, maintainability, separation of concern etc. It is a engineering problem derived from a computer science problem: the lack of composability.

Let's define a backtracking effect which works among monadic statements and executes backtracking handlers in reverse order. At any time the programmer can decide either executing further handlers or continue forward using the continuation of the exception handler. In the previous example, imagine that *payment* fail, but we want to give more opportunities. Lets imagine some primitives like *onException*, which register an exception handler which will be called by a backtracking mechanism, and there is a *continue* primitive that would stop backtracking and resume execution forward. Using the primitives and their semantic, we can add code to the example so that the client could do two more payment attempts after which the program will unreserve the book in both databases and terminate:

```
data CardFailed= CardFailed  deriving Show
data CardThirdAttemptFailed = CardThirdAttemptFailed deriving Show

instance Exception CardFailed
instance Exception CardThirdAttemptFailed

data Counter= Counter (IORef Int) deriving Typeable


payment book= do
        setState newCounter
        pleaseEnterCard `onException` (\(e :: CardFailed ) ->
             do
                  Counter rn <- getState <|> return newCounter
                  n<- liftIO $ readIORef rn

                    if n==2 then liftIO $ throw CardThirdAttemptFailed else do
                        liftIO $ writeIORef rn $ n+1
                        pleaseEnterCard
                        continue)
        pay
        where
        pay= throw CardFailed   -- make each attempt fail
        newCounter=  Counter (unsafePerformIO $ newIORef 0)

        pleaseEnterCard = liftIO $ print "Please enter Card"




updateDB1 book=  update 1 book `onException`
        \(e :: CardThirdAttemptFailed) -> unreserve 1 book

updateDB2 book=  update 2 book `onException`
       \(e :: CardThirdAttemptFailed) -> unreserve 2 book

update n _= liftIO $ putStr "Updating database" >> print n
```

```
unreserve n _= do
    liftIO $  putStr "unreserving book in database"
    print n
```

We use some primitives like *getState* and *setState* not yet detailed. They retrieve and store state.

In the code above, each computation only mind in his own business. The structure of the program does not change by the fact that we have exceptions and exception handlers. At any point we can resume execution with *continue* which will execute his own continuation, so we have to manage not only exception handlers, but also their corresponding continuations. We need to define state with a data structure that contains both.

To make this backtracking effect work, we generalize the 'backtracking' data not only for exceptions but for any kind of data types, later we will particularize for exceptions: We need to store, in the state, a stack of handlers and their continuations.

```
data Backtrack b= forall a c.
         Backtrack{backtracking :: Maybe b
                  ,backStack :: [(b ->ContIO  c,c -> ContIO  a)] }
                  deriving Typeable
```

The first field contains the data transported by the backtracking being carried out. In the case of exceptions, this is the exception data. The second contains the list of handlers and continuations.

First we need to define some utility functions; *backCut* delete all the undo actions registered till now for the given backtracking type.

```
backCut :: (Typeable b, Show b) => b -> ContIO ()
backCut reason=delData $ Backtrack (Just reason)  []
```

onBack run the action in the first parameter and register the second parameter, (the handler). When the backtracking is called, the handler is called with the backtracking data as argument.

```
onBack :: (Typeable b, Show b) => ContIO a
       -> ( b -> ContIO a) -> ContIO a
onBack ac back =   registerBack  ac back

   where

   registerBack :: (Typeable a, Show a) => (a -> ContIO a)
                -> a -> ContIO a
   registerBack  ac back = callCC $ \k -> do
     md <- getData `asTypeOf` (Just <$> (backStateOf $ typeof back))
     case md of
        Just (bss@(Backtrack b (bs@((back',_):_)))) ->
           setData $  Backtrack b  ( (back,  k):   unsafeCoerce bs)
        Just (Backtrack b []) -> setData $ Backtrack b  [(back , k)]
        Nothing ->  do
```

```
          setData $ Backtrack mwit  [  (back , k)]
     ac


   typeof :: (b -> ContIO a) -> b
   typeof = undefined
   mwit= Nothing `asTypeOf` (Just $ typeof back)
```

forward is a generalized form of *continue*. It tells *back* to resume execution forward invoking the handler continuation with the result returned by the handler.

```
forward :: (Typeable b, Show b) => b -> ContIO ()
forward reason=  do
    Backtrack _ stack <- getData `onNothing`  (backStateOf reason)
    setData $ Backtrack(Nothing `asTypeOf` Just reason)  stack
```

back start the backtracking process. It executes all the handlers registered till now in reverse order. A handler can use *forward* to stop the backtracking process and resume the execution forward. If there are no more undo actions registered then the execution stops

```
back :: (Typeable b, Show b) => b -> ContIO a
back reason = do
  Backtrack _ cs <- getData  `onNothing`  backStateOf  reason
  let  bs= Backtrack (Just reason) cs
  setState bs
  goBackt bs

  where

  goBackt (Backtrack _ [] )= empty
  goBackt (Backtrack Nothing _ )= error "goback: no reason"

  goBackt (Backtrack (Just reason) ((handler,cont) : bs))= do

        x <- unsafeCoerce handler reason

        Backtrack mreason _ <- getData `onNothing`  backStateOf  reason

        case mreason of
                  Nothing    ->  ety $ cont x
                  justreason -> do
                        let backdata= Backtrack justreason bs
                        setData backdata
                        goBackt backdata
                        empty

backStateOf :: (Monad m, Show a, Typeable a) => a -> m (Backtrack a)
backStateOf reason= return $ Backtrack (Nothing `asTypeOf` (Just reason)) []
```

1.14. Exception handling trough backtracking

Now we apply the general backtracking mechanism for exceptions. To manage an exception as data that will be backtracked with the above primitives, first we need to catch every exception which happens in the continuation.

onException Install an exception handler. Handlers are executed in reverse (i.e. last in, first out) order when such exception happens in the continuation. Note that multiple handlers can be installed in sequence for the same exception type.

The semantic is thus very different than the one of the standard *onException* defined for the *IO* monad.

```
onException :: Exception e => ContIO a -> (e -> ContIO a)
             -> ContIO a
onException mx f= onAnyException mx $ \e ->
    case fromException e of
       Nothing -> return $ error "this should not be evaluated"
       Just e'  -> f e'
  where
  onAnyException :: ContIO a -> (SomeException
                   ->ContIO a) -> ContIO a
  onAnyException mx f= ioexp `onBack` f
    where
    ioexp = callCC $ \cont -> do
       st <- get
       ioexp' $ runContState st (mx >>=cont ) `catch` exceptBack st

    ioexp' mx= do
      (mx,st') <- liftIO  mx
      put st'
      case mx of
        Nothing -> empty
        Just x  -> return x
```

onException uses *onBack* to register a backtracking handler. then wraps a *catch* handler around the computation and his continuation. The handler call *back*, which perform the backtracking. *exceptBack* is the computation that catches any exception and call the backtracking mechanism:

```
exceptBack st = \(e ::SomeException) -> do  -- recursive catch itself
                      if (isNothing (fromException e:: Maybe Empty))
                            then runContState st  (back e )
                                `catch` exceptBack st
                            else throw $ Empty st
```

it ignores *Empty* exceptions. We define *backCut* and *forward* specific for exceptions:

```
cutExceptions :: ContIO ()
cutExceptions= backCut  (undefined :: SomeException)
```

```
continue :: ContIO ()
continue = forward (undefined :: SomeException)
```

catcht is semantically similar to *catch*. it catches an exception in a Cont block, but the computation and the exception handler can be multithreaded, reactive etc.

```
catcht :: Exception e => ContIO a -> (e -> ContIO a) -> ContIO a
catcht mx exc= do
      rpassed <- liftIO $ newIORef False
      sandbox  $ do
         delData $ Backtrack (Just (undefined :: SomeException))  []

         r <- onException'  mx $ \e -> do
                  passed <- liftIO $ readIORef rpassed
                  if not passed then unsafeCoerce continue >> exc e  else empty
         liftIO $ writeIORef rpassed True
         return r

   where
   sandbox :: ContIO a -> ContIO a
   sandbox  mx= do
     exState <- getData `onNothing` backStateOf (undefined :: SomeException)
     mx   <* setState exState
```

finally *throwt* throws an exception in the Cont monad invoking the backtracking mechanism.

```
throwt :: Exception e => e -> ContIO a
throwt= back . toException
```

As seen above, the exception handling primitives *onException* and *catcht* are defined so that any exception thrown withing the IO monad is also captured by the backtracking mechanism.

1.15. Undoing transactions trough backtracking and exceptions

Now lets execute the computation *emarket* with the primitives defined above, to see the exception logic working:

```
Navigating the list of products
Updating database1
Updating database2
"Please enter Card"
"Please enter Card"
"Please enter Card"
unreserving book in database2
unreserving book in database1
```

1.16. Extensible state

Finally we need an extensible state management, in a Rich Hickey style adapted to Haskell. We need state to transport data structures for composing effects, like the backtracking mechanism and the alternative mechanism but also for any need of the application programmer. It is a type-indexed map with convenience accessors. The backtracking mechanism demonstrates how state and continuations can be combined to 'edit' the flow of the program. Let's give to the application programmer leveraging this power with a build-in extensible state.

getData look in the state for a data of the desired type. If the data is found, a *Just* value is returned. Otherwise, *Nothing* is returned.

```
getData :: (MonadState Stat m, Typeable a) => m (Maybe a)
getData = resp
  where resp = do
          list <- gets mfData
          case M.lookup (typeOf $ typeResp resp) list of
            Just x  -> return . Just $ unsafeCoerce x
            Nothing -> return Nothing
        typeResp :: m (Maybe x) -> x
        typeResp = undefined
```

getState Retrieve a previously stored data item of the given data type from the monad state. The data type to retrieve is implicitly determined from the equested type. If the data item is not found, *empty* is executed.

```
getState :: Typeable a => ContIO a
getState =    do
        mx <- getData
        case mx of
          Nothing -> empty
          Just x  -> return x
```

Remember that empty stops the monad computation. If you want to print an error message or a default value in that case, you can use an 'Alternative' composition. For example:

```
getState <|> error "no data"
getInt = getState <|> return (0 :: Int)
```

setData stores a data item in the monad state which can be retrieved later using 'getSData'. Stored data items are keyed by their data type, and therefore only one item of a given type can be stored. A newtype wrapper can be used to distinguish two data items of the same type.

```
setData :: (MonadState Stat m, Typeable a) => a -> m ()
setData x = modify $ \st -> st { mfData = M.insert t (unsafeCoerce x) (mfData st) }
  where t = typeOf x
```

Example usage:

```
data Person = Person
    { name :: String
    , age :: Int
    } deriving Typeable

main = keep $ do
      setData $ Person "Peter"  55
      Person name age <- getSData
      liftIO $ print (name, age)
```

Finally, to delete the state data:

```
delState :: (MonadState Stat m, Typeable a) => a -> m ()
delState x = modify $ \st -> st { mfData = M.delete (typeOf x) (mfData st) }
```

1.17. All together now

Now we combine some of these pieces that implement asynchronicity, non-determinism, event management, threading etc to demonstrate the composability of the DSL whe have defined.

```
main= keep examples

examples= keep $ do
      fork consoleLoop
      (reactOption "menu" "show the menu" >> return()) <|> return ()
      combination' <|> testAlternative'
                   <|> chooseTwo'
                   <|> pythagoras'
                   <|> emarket'

      where
      fork f= (async f >> empty) <|> return()

      testAlternative'= do
         reactOption "alt" "alternative parallel example"
         testAlternative

      combination'= do
         reactOption "comb" "parallel combination of alternative and monoid"
         combination

      chooseTwo'= do
         reactOption "two" "parallel list processing"
         choosetwo

      pythagoras'= do
         reactOption "pyt"  "pythagoras triangle"
         pythagoras
```

```
emarket'= do
   reactOption "mkt"  "emarket: example of backtracking and exceptions"
   emarket
```

This is an example run session:

```
$ stack runghc contEffects.hs
enter menu       to:show the menu
enter comb       to:parallel combination of alternative and monoid
enter alt        to:alternative parallel example
enter two        to:parallel list processing
enter pyt        to:pythagoras triangle
enter mkt        to:emarket: example of backtracking and exceptions
mkt
"mkt"
navigating the list of products
Updating database1
Updating database2
"Please enter Card"
"Please enter Card"
"Please enter Card"
unreserving book in database2
unreserving book in database1

menu
"menu"
enter comb       to:parallel combination of alternative and monoid
enter alt        to:alternative parallel example
enter two        to:parallel list processing
enter pyt        to:pythagoras triangle
enter mkt        to:emarket: example of backtracking and exceptions

menu
"menu"
enter comb       to:parallel combination of alternative and monoid
enter alt        to:alternative parallel example
enter two        to:parallel list processing
enter pyt        to:pythagoras triangle
enter mkt        to:emarket: example of backtracking and exceptions
comb
"comb"
world2
hello world

two
"two"
(1,'a',ThreadId 114)
(3,'b',ThreadId 117)
(1,'c',ThreadId 119)
```

```
(2,'a',ThreadId 115)
(2,'b',ThreadId 120)
(2,'c',ThreadId 121)
(3,'a',ThreadId 116)
(3,'c',ThreadId 122)
(1,'b',ThreadId 118)

py
pyt
"pyt"
(4,3,5,ThreadId 325)
(8,6,10,ThreadId 650)
```

The code of this paper with this main program can be obtained and executed from [Anonymous 2018]

1.18. Conclusions and future work

A continuation monad with the help of state allows the ”edition” of the execution flow at run time and allows the composition of impure asynchronous effects.

There are a lot to consider to evolve this model in theoretical and practical terms: more effects, but also more details, more analysis of bibliography, comparison of similar approaches not considered here for lack of time at this stage.

REFERENCES

Anonymous. 2018. impure asynchronous effects demo. (2018). https://gist.github.com/anonymous/5638b9a8f620e83d41fe42a9de9cdacc

Philip Wadler. 1994. Monads and composable continuations. *LISP and Symbolic Computation* 7, 1 (1994), 39–55. DOI:http://dx.doi.org/10.1007/BF01019944