Algebraic and Monadic Composition of Concurrent and Distributed Applications*

A Beautiful Orchestration of the Awkward Squad†

Alberto G. Corona
Software Engineer
Axioma Co.
Juan Abello 2
S. Lorenzo Escorial, Madrid 28200, Spain
agocorona@gmail.com

Abstract

In 2010, Simon Peyton Jones [10] presented how to handle the awkward squad in Haskell. The awkward squad included the fundamental attributes of real-world applications like input/output, concurrency and exceptions. Concurrent and distributed applications are now the order of the day and present the next level challenge of composing applications having arbitrary complexity with mathematical soundness. In this paper, we present a monadic abstraction that allows the programmer to algebraically compose computations that may themselves be composed of multiple concurrent, local or remote computations. The abstraction affords the programmer a single, sequentially composed view of a concurrent and distributed program; just the way a single threaded, single node program is written. It overcomes the composability challenge posed by asynchronicity in event driven applications. From parallel and distributed map reduce applications to unified client and server side web applications can be expressed effortlessly using this abstraction.

CCS Concepts •**Theory of computation → Parallel computing models; Distributed computing models; MapReduce algorithms; Vector / streaming algorithms; MapReduce algorithms; Functional constructs;** *Concurrent algorithms;* •**Software and its engineering → Cloud computing; Synchronization;** *Publish-subscribe / event-based architectures; Interoperability; Checkpoint / restart;* Data flow architectures; Requirements analysis; Software design engineering; •**Information systems →** *Web interfaces;* •**Computing methodologies →** *Vector / streaming algorithms;*

Keywords Haskell, Cloud, distributed, reactive, parallelism, concurrency, exceptions, threading, Web

*with title note
†with subtitle note

with paper note.
HS'17, Oxford, United Kingdom
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

ACM Reference format:
Alberto G. Corona. 2017. Algebraic and Monadic Composition of Concurrent and Distributed Applications. In *Proceedings of Haskell Symposium, Oxford, United Kingdom, September , 2017 (HS'17),* 12 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

A goal of Software Engineering would be a high level language that could express user requirements and make them run by developing from this high-level description down to the details. All with re-usability of each component and easy maintainability.

From the practical point of view, there is nothing more reusable than mathematical formulas and recipes. Mathematical formulas include binary operations:

```
A -> A -> A
```

With these binary operators it is possible to combine any number of elements, two at a time.
Ideally, these binary operations applied to programming at the highest level would conform an algebra for software components which may allow unrestricted composability.

On the other side, recipes are sequences of instructions that are easy to follow. They are also declarative since they do not impose how to execute them: When cooking a recipe, the task can be interrupted and restarted. We can follow the recipe in a distributed way, by collaboration among many people. If we fail, at any moment we can go back and restart the execution at the most advanced point possible, leveraging the work already done.
Informally, the semantic of recipes can be expressed in terms of a monad with effects of multi-threading, backtracking, stopping/resuming, event handling and distributed computing.

In the other side, formulas can be assimilated to applicative, monoid and alternative combinations. Once this is recognized as the goal, the difficulty lies in the management of these effects within the monad while maintain composability.

let's define the goal in the following terms using some combinators of the language Haskell: Let *ap1* and ap2 two applications with arbitrary complexity, with all effects including threading, asynchronous IO, indeterminism, events and distributed computing.

Then the combinations:

```
ap1 <|> ap2
ap1 >>= \x -> ap2
ap1 <> ap2
(,) <$> ap1 <*> ap2
```

are possible if the types match, and generate new applications that are composable with the same combinators as well.

We present here a set of techniques, a set of primitives made with these techniques and a set of libraries[4] made with these primitives, as well and some examples made with these libraries that achieve that goal. Although each technique would need to be discussed in more detail, this article demonstrate the expressive power of the mix that otherwise would be lost in the details.

As a proof of concept we developed a composite application [5] made of different applications that are combined using the above mentioned combinators. One of the applications implements a map-reduce operation using the Apache/Spark[9] dataflow model. Another implements a distributed chat following the architecture of the Actor Model. Both compose a single web app that runs in all the nodes connected and visualize the results in a web browser as well as in the node terminals.

2 Continuations

By definition, anything asynchronous break the flow of a process. To deal with asynchronicity, the straightforward solution is to embrace it in the state machine model. In the pure formulation of Object Oriented Programming, each object is a state machine that receives messages and send messages to other state machines. The first OOP languages, Simula and Smalltalk[15], were designed around this basic idea in order to manage discrete events.

The specific way to deal with the asynchronous problems in functional programming has been the use of continuations. But continuations are hard to manage. Delimited continuations has not been widely adopted for event management despite the fact that it is theoretically ideal for hiding the asynchronous nature of many effects. Delimited continuations are too powerful and generates code that is hard to understand. Due to that power, continuations don't compose following laws that would make the code easy to reason about.

Additionally, there are domain problems in which the continuation must be serialized and deserialized; Such is the case of distributed computing, or Web navigation. Only in fully dynamic languages like schema or Smalltalk, serialization is guaranteed. But it is may be non practical since the size may be huge[3]. serialization of closures in managed languages like Java is discussed in the paragraph about distributed computing.

Futures is the name of a technique in which a thread wait for the result of one or many parallel tasks. They are a form of stack-less continuations. It is used in languages like Scala and Rust as well as in Haskell[11]. This model can perform parallelism and concurrency of simple tasks with limited composability. They need to impose unneeded concurrency since there is a single "master" thread that continue the execution, when the problem would admit more parallelism.

3 A revisit of the Monad

To overcome the problem of continuations, Instead of unlawful operations like *shift* and *reset* present in delimited continuations, we developed a more intuitive notion of continuation constructed over the bind operation of a monad. Although this is no formal demonstration of this claim, we demonstrate by examples that kind of continuation follow the rules of the monadic combinators. Also, we use a different serialization method that is architecture and language independent.

There is a parallelism often mentioned informally between the power of monads and continuations. However although the continuation monad can be used to implement operations like *shift* and *reset*, P. Wadler [14] suggest that continuations can not be fully managed internally by a monad since the type of bind is not rich enough to express the necessary types. He suggest an extension of monad, the indexed monad that can manage delimited continuations fully.

We approached the problem in a different way. We "constructed" the notion of continuation from the bind operation. In this article we use Haskell pseudocode. Bind is defined as:

```
(>>=) :: m a -> (a -> m b) -> m b
g >>= f   = ...
```

Where *m* is the Monad. This monadic sequence:

```
do x <- a; y <- b x; z <- c y; d z
```

Desugared as:

```
a >>= (\x ->b >>= (\y -> c >>= \z -> d))
```

When the second bind is executed, we can see that:

```
g=  \xvalue -> b

f= (\y -> c >>= d)
```

Where *xvalue* is the result of the execution of 'a'.
The real representation of g and f depend on the evaluation strategy and the implementation. It may be interpreted and use variable substitution, or it may be compiled in a thunk, It is not in the scope of this article to detail it. Independently of the evaluation engine, when the second bind is executed, all the variables of g and f that referred to x point to a defined value, generated by the evaluation of a. In the case of a lazy language, like Haskell, g is probably in a thunk that contains that calculated value. His body b is still unevaluated. This is what the pseudo-lambda expression for g means.
In the last bind, both terms are:

```
g= \xvalue -> (\yvalue -> c)

f= \z -> d
```

Let's call g "the closure". It contains all the variables calculated in previous binds that are in scope for the computation when a new bind is called. f is the continuation that executes within the scope of these variables. Most likely, if the evaluation strategy of the compiler is efficient, it has not been touched by the evaluation of the previous statements. If we define bind so that we capture g and f in a state monad then we can use it to create primitives that make use of them. The pseudocode of a bind that stores its own computation could be:

```
g >>= f= Transient $ do
      get state
      store g and f in the state
      r <- g
      reset state
      f r
```

Since the types of the closure and the continuation are different in each bind, only an indexed monad can manage the types involved, as [14] suggested.
To keep it under the framework of an standard monad and get the benefits of *do* syntactic sugar, some type erasure via *unsafePerformIO* is necessary. But there are some guarantees; By construction we know that that closure and continuation stored in the state will type match and the result has the type of the whole computation.
during the time that g is executing, in any part of it, we can access to the whole g and f; In the above example, if we want to receive events in b, we can use the closure and its continuation stored in the state to continue after this event, without breaking composability.

4 Handling events

waitEvents execute the closure and the continuation whenever an event happens. It is assumed that events are read by a blocking IO computation which is rescheduled immediately after receiving an event, to read the next.

```
do msg <- waitEvents blockingIO   -- g
process msg                       -- f

The pseudocode of \textit{waitEvents} would be:

 waitEvents blockingIO= do
  if not $  buffer filled
    then do
     (closure,continuation)<- get
     liftIO $ forkIO $ loop $ do
          msg <- blockingIO
          putInBuffer msg
       forkIO $ runMonad $ closure >>= continuation
     empty
    else
     empty buffer
     return buffer
```

When the closure is re-executed, the *waitEvents* sentence is executed again, but this time there is a value in the buffer, so the value is returned and the continuation is executed.
The closure is reexecuted because g may be any complex computation involving *applicative*, *alternative* and monadic combinators of which *waitEvents* may be only a term.
The continuation may be also complex. Each term in the above example can be an arbitrarily complex expression. suppose that:

```
c = \x -> e >>= h
```

Then, when the computation is in that internal bind of c, e is the closure, but the continuation is not only h, but also the continuation of the outer computation

```
h >>= d
```

We have to modify the state accordingly.

We dont want to re-execute everything when a term receives an event; The continuations should be delimited so that the programmer decide about the propagation of events. That role is played by *empty*, which introduce the effect of early termination and the execution of alternative computations. *empty* delimits the continuation. *runCont* produce a result whose type is the type of the whole computation where it is. Both play the rules. This is enough for our purposes.
Observe that in *waitEvents*, there is a new thread for each event. Also, it return *empty* to the current thread. This allows

parallelism as well as non-determinism. As a consequence, a *Transient* computation may return 0, 1 or N results
If we assume that *keep* is the runner of the monad:

```
keep: TransIO a -> IO a
```

Since *waitEvents* executes the blocking IO call in different threads, then this expression:

```
main= keep $ waitEvents getLine
                >>= liftIO . print
```

Has some good properties:

- It is reactive, since it respond to keyboard events
- it is intuitive since it "follows the recipe", with monadic composability.
- It processes a stream of messages.
- It is non blocking, despite using a blocking IO call
- parallelize the processing of inputs if they are entered fast enough.
- It is loop free, but works like a loop
- It can be composed algebraically, as we will see below.

5 Alternative processing

Since *waitEvents* return *empty* to the current thread, the *alternative* instance can be used to execute more than one simultaneous blocking IO actions and yet maintain composability:

```
main= keep $ do
    r <- waitEvents foo <|> waitEvents bar <|> ...
    liftIO $ print r

foo,bar :: IO a
```

5.1 IRC client

The above snippet is composed two times, for input and output, in this example:

```
import           Transient.Base
import           Network
import           System.IO
import           Control.Monad.IO.Class
import           Control.Applicative

main = do
    h <- connectTo host 6667
    keep' $ process h stdout <|>
            process stdin h
  where
  process in out=
    waitEvents (hGetLine in)  >>=
                          liftIO . hPutStrLn out
```

The alternative operator is defined as usual, but since *waitEvents* return *empty* to the base thread, the second term is executed too. The result are two threads running in parallel: one for input and another for output.
We can define some variants of the asynchronous primitive *waitEvents*:

```
-- wait events permanently
waitEvents :: IO a -> TransIO a

-- wait a single event
async :: IO a -> TransIO a

-- sampling with maximum frequency
sample :: IO a -> Time -> TransIO a

-- wait events temporally
parallel :: IO (StreamData a) -> TransIO a

data StreamData a= SDone
                 | SLast a
                 | SMore a
                 | SError SomeException

abduce :: TransIO ()
abduce = async $ return ()
```

parallel re-executes the blocking IO computation until there are no more events.
sample execute a polling IO action with a certain frequency and inject the result when the value changes.
abduce return empty in the current thread and continue the execution in new thread.

6 Event handlers

Event handlers break composition. Every framework has some kind of inversion of control. The programmer is forced to write the logic of the program in a set of disjoint handlers. It is not possible to express his program in a clear sequence unless we create some primitive that pass our continuations as the event handlers to the framework, so that our program continues when it is called from it. This is what the *react* primitive does :

```
react setHandler toreturn= do
    case buffer in state of
        Nothing -> do
            liftIO $ setHandler $ \event ->do
              put event in buffer
              (closure,continuation)<- get
             runTrans $ closure >>= continuation
```

```
            toreturn
          empty
      buffer filled -> do
          empty buffer
          return buffer content
```

Here *setHandler* is the framework-specific installer of event handlers.

```
setHandler :: (Event -> IO a) -> IO ()
```

The return type of the handler *a* is defined by the particular framework. The second parameter, *toreturn* computes it. Usually, it is *return()*
Since the program has set its event handler, his continuation will be called when an event occurs. The execution continues downstream after the event, without breaking composability. Assuming that *setCallback* is the installer of callbacks in the framework:

```
x <- foo
event <- react setCallaback
              (return ())
y <- bar event
...
```

react install the callback when *foo* return a result. *bar* and the rest of the monadic sequence is called with the event value when the callback is invoked by the framework.

react has a similar structure than *waitEvents*, but there are no thread creation, since the threads are generated by the framework. It also allows *applicative* and *alternative* composition.

...

7 Concurrency: monoid and applicative

If two asynchronous processes are combined with an applicative expression, the result is a single result in a single thread. This is a form of concurrency:

```
do
    r <- (,) <$> async (return "hello")
             <*> async (return "world")
    liftIO $ print r

> ("hello","world")
```

Since the *monoid* instance is defined in terms of *applicative*, the latter also implements concurrency:

```
do
    r <- async (return "hello ") <>
         async (return "world")
    liftIO $ print r

> "hello world"
```

The *applicative* instance stores the result of each operand in a mutable variable. When the other operand finalizes execution it inspects the other buffer. If it is filled, it return the result of the *applicative*. If it is not, the thread will finish. So the slowest thread is the one that find the other result and continues execution.
To implement the applicative instance, it is necessary to read the result of the other operand before the execution of the continuation. That is one of the reasons why closures and continuations are stored separated in the state.

In contrast:

```
do
    r <- async (return "hello") <|>
         async (return "world")
    liftIO $ print r

> "hello"
"world"
```

There is no concurrency here. The alternative operator, combined with an asynchronous primitive produces parallel tasks that execute the continuations independently.

8 Algebraic composition

With the *Applicative* operators we can define algebras of binary operations that describe the domain problem if the domain admit such mathematical structure:
We can use a numeric algebra:

```
mx + my=  (+) <$> mx <*> m
mx * my=  (*) <$> mx <*> m
```

It does not matter if the calculation is distributed among many nodes or if some operands are streams of values injected in the formula by means of callbacks.

For example:

```
amount= price * quantity

realTimePrintSold= do
    newAmount <- amount
    liftIO $  print newAmount

price :: TransIO Double
price= react setPriceChangeCallback (return ())
```

```
quantity :: TransIO Double
quantity= do
    units <- waitEvents quantityChannel
    return $ fromIntegral units
```

The above expression would return paid quantities of a product whose price variations are read from a callback and whose units sold are read from a channel. The formula *price * quantity* remain unchanged.
In the same way, a relational algebra with parallelism and concurrency, or even distributed, is possible:

```
query1 and query2= intersect <$> query1 <*> query2
```

It preserves the mathematical properties of a domain problem independently from effects. It is possible to program at a higher level without abandoning equational reasoning. Once more there is no formal proof of this claim. Only a reasonable assumption coming from the examples.
It also facilitate the creation of complex components that can be reused as easily as any formula in larger applications made of algebraic and monadic expressions.

9 Console input with threading

Sharing the standard input among different threads is important for composing modules that are controlled by a single input stream. If the standard input is stored in a buffer by an independent thread, with the aid of *waitEvents* we can create an asynchronous console input primitive for the creation of menus that do not block, so that menu options can be composed:

```
option v text = do
  liftIO $ print text
  waitEvents loop
  where
  loop= do
    b <- read buffer
    if b==v
       then
           clean the buffer
           return v
       else loop
```

option can be used to compose two programs that wait for console input:

```
main= keep $ do
 (do option "a" "say hello"
     liftIO $ print "hello") <|>
        (do option "b" "say world"
            liftIO $ print "world")
  liftIO $ print "next"

Enter "a"  to:  say hello
Enter "b"  to:  say world
>a
hello
next
```

9.1 Command line input

Asynchronous, non blocking parsers can be implemented in the same way:

```
n <- input (< 5) "give a number < 5 :"
```

10 Thread control

Since *waitEvents* produce a new thread for each input, this can be overkill. Too much parallelism may not be optimal. We can receive a lot of events simultaneously:

```
choose list=
 foldr (<|>) empty $ map (async . return) list
```

If there is no way to limit them, expression like this may produce an explosion of threads. It is better to reduce the threads and have a form of implicit queuing.
We can extend the asynchronous primitives to check for a counter of threads in the state. If it is 0, it should not execute the continuation in a new thread but within the current one. When a thread finish the counter is increased.

```
do
     x <- threads 2 choose [1..100]
     liftIO $ print (x * x)
```

The above example executes in two threads. Each async operation wait his turn until the previous has been processes. There is an implicit queuing.
The code below perform a loop within the current thread, since there are no other available:

```
threads 0 $ choose[1...10] >>= liftIO . print
```

Instead of controlling threads one by one or in groups using pools explicitly identified, threads can be managed using the hierarchy of the program. That avoids additional complexities: if threads hierarchy may be maintained by the state, it is possible to navigate it. If we identify the threads with labels, we can display such hierarchy for monitoring and debugging purposes.

11 State

The transient monad run over a state monad that includes a map of types to values in a Data.Map structure. This allows the management of an arbitrary number of pure states.
This state is inherited and continues trough asynchronous primitives that generate new threads.
These primitives manage user-defined data:

```
setState (1 :: Int)
setState  MyRegister{....}
```

getState perform a lookup for data of the type expected in the map. If it does not exist it returns *empty*.

```
getMydata :: TransIO MyData
getMyData= getState <|> return initialValue
```

This last expression reproduces the semantic of the *get* method of the state monad. The advantage is that it can be defined for any kind of data that the user may need. It is also used internally by transient to implement some primitives that will be described later.

12 reactive channels

If we want to inject events and trigger threads in other branches of the monad, we can construct channels that have the semantic of publish-subscribe.

EVars are unbounded reactive channels defined as:

```
data EVar a= EVar (TChan a)

readEVar :: EVar a -> TransIO a
readEVar (EVar ev)=
 waitEvents . atomically $ readTChan ev
```

An EVar is a broadcast channel that is attached to *waitEvents*.

```
writeEVar (Evar ev) x=
 liftIO . atomically $ writeTChan tv x
```

This computation has three parallel branches. One of them inject values, the other two extract and print them:

```
do
 ev <- newEVar
 r  <- readEVar ev <|>
       readEVar ev <|>
      (mapM_ (writeEVar . ev) [1..10] >> empty)
 liftIO $ putStr r
```

```
> 112233...
```

Mailboxes are used in distributed computing for asynchronous communications. They are stored *EVar*s indexed by a serializable identifier.

13 Transactions and exceptions

Tentatively, a transaction could be defined as an special exception treatment in which the handler either retry the execution until there is no exception or rollback and abandon. In both case the state of the system must be coherent. Isolation can be achieved with a two phase commit. If there is no isolation and the transaction has been performed, a compensation can undo the changes. When programming in the large, compensations are usually the only technique possible, since isolation can not be achieved.
If we record the continuations of the points that we want to retry, we can re-execute them. The possible new threads created during the process can inherit such state information, so any of them can retry.
We want primitives that may:

- Insert a handler (*onUndo*) that is stored in the stack
- Execute the backtracking by executing the handlers stored for it (*undo*)
- Suspend backtracking and retry from this point on (*forward*).

An example of usage of these primitives is this snippet that may be self-explanatory:

```
do  select book
    enter card `onUndo` (enter card >> forward)
    reserve book `onUndo` unreserve book
    r <- paymentOf book
    when (r == Failure) undo
    processOrder
```

onUndo handlers are executed in reverse order. When the payment fail, *undo* execute the handlers, stored in the state. the first *onUndo* unreserve the book, then the second undo ask the user for a different card. if this is accomplished (maybe in another thread since the introduction of a new card may be asynchoronous) the computation will retry until all the transaction is done or it is rolled back if the card introduction is aborted. In any case the book database stores a coherent state with the account database.

```
onUndo action handler= do
 if not bactracking
   then  do
     store (closure,continuation) in the stack
     execute action
   else handler
```

```
undo= do
  set backtracking flag
  (hander, continuation):_ <- getState
  flag <- execute first handler in the stack
  if backtracking flag
   then proceed with next handler in the stack
   else execute continuation

fordward= unset the backtracking flag.
```

This same mechanism can be used to execute exception handlers, with the addition of an extra paramenter transported by the backtracking: the exception. onException is equivalent to *onUndo*.

```
onException :: SomeException e
            => e
            -> TransIO a
```

This mechanism also makes monadic composition cleaner, since exception handlers do not need to wrap around all the rest of the computation and the handler does not introduce additional bifurcations of code. The mechanism run "in the main track" instead of creating side tracks. It also works even if the program fork multiple threads.
This code is used to restart a network node when the connection fails:

```
onException $ \(e :: ConnError) ->
               do
                restart node
                continue
runAt node foo
```

This general backtracking engine unifies transactions and exceptions as well as resource management.

14 Logging/recovery

logged is a primitive that, given a computation, store its result in the state and return it.

```
logged :: Serializable a
       => TransIO a
       -> TransIO a

data IDynamic=
   forall a.Serializable a =>
           IDyn a |
           IDyns ByteString

LogElem= Var IDynamic
       | Exec
       | Wait

data Log= Log Recovery [LogElem]
```

A log element can be in a serialized or deserialized state. it also can denote an asynchonous operation whose value is unknown (*Wait*) or it may be being executed and has not yet finished (*Exec*).
if the recover flag is on, *logged* , run in recovery mode and read the next element of the log instead of executing its argument.

```
do
  setState $ Log True ["hello"]
  r <- logged  foo  -- dont execute x
  liftIO $ print "world"

> "hello"
```

A pseudocode for the *logged* primitive would be:

```
logged f= do
     Log replay log <- getState <|>
                       return Log False []
     if replay
       then
         case head log of
          IDyn x ->   return x
          IDyns s ->   return $ deserialize s
          Wait -> empty
          Exec -> f
       else do
          setState $ Log False (Exec:log)
          x <- f
          setState $ Log False (IDyn x:log)
          return x
```

After the execution of *f*, all the results that *f* may have put in the state internally is discarded. Only its result remain in the log. The log stores only the top level results. It would contain only the minimal "route" necessary for recovering the computation state.

15 Distributed computing

To pipeline tasks that involve a big amount of data distributed in the cloud it is convenient to bring high level, functional interfaces for programmers. In distributed computing, the functional approach demand the usage of some kind of closure serialization.
Managed environments like the JVM and the NET support serialization of closures with some restrictions. Big Data frameworks like spark [9] and mbrace [12] use the serialization services of the virtual machines to transport state and code with the caveats and errors that the engineers and theorists are trying to solve [13] For relatively static distributed

task like map-reduce, it is feasible to serialize, move and deserialize big closures, since this is realized at a setup phase previous to the real calculation, but it is not efficient in more dynamic scenarios where the topology of the distributed application changes and programs communicate dynamically in the middle of a transient computation that generate new closures.
Machine learning poses similar difficulties. The tensor operations involved in ML are elegantly expressed algebraically, but they should be decomposed in terms of the framework execution model. TensorFlow is a parallel and distributed graph execution framework. There is a work in progress to abstract a functional interface over this dataflow model[1]
A single functionality of a program may need to use:

- a map-reduce engine for database manipulation.
- an actor model for sending activity information to other users.
- a distributed tensor operation to suggest content for the users.
- a distributed, balanced set of application servers to execute the application.
- different client GUI frameworks for different devices.

That common case demand the integration, configuration and maintenance of very different frameworks.
Additionally, with the exception of *mbrace* which makes and excellent usage of monads [6], these frameworks are programmed at a low level and the functional interface is a tiny layer on top. This makes dynamic composable programming not possible out of some ad-hoc coarse grained primitives like map and reduce. The creation of new processing primitives implies more low-level tweaking.
Static closures [8] are pointers whose addresses are know at compilation time. Cloud Haskell[7] uses this mechanism to communicate identical binary programs. In the other side, dynamic closures are graphs generated at run time, and includes results of previous computations mixed with different static pointers to code that has not been reduced yet to results. For this reason we understand that static closures do not allow the required composability.
In the other side, the Actor model, used for some distributed architectures, is an OOP pattern that embrace asynchronous messaging where the execution flow is difficult to reason about. Using continuations, we implement asynchronous messaging while maintaining composability.
Hereafter, we define three basic distributed primitives that allows the programming of map-reduce without losing functional composition. As a consequence, map-reduce is a module in the library[2] that may be called anywhere in a haskell program.

16 Remote execution

Using the primitives defined for logging and recovery, it is possible to start a computation in a network node and continue in another node. We need primitives for connecting, sending and receiving it.
Hereafter, we define three basic distributed primitives that allows the programming of map-reduce without losing functional composition. As a consequence, map-reduce is a module in the library[2] that may be called anywhere in a haskell program.
Let's hypothesize three primitives: a socket listening primitive: *listen*, a connection primitive *wormhole* and a log send primitive *teleport* that, with *local*, allows the execution of *foo* in local node and print the result in a remote node:

```
local= logged

listen= do
  (hisClosid,myClosid,log)
      <- waitEvents $ read socket messages

  setState $ Log True log
  setState hisClosure
  closure <- lookup myClosureid
  execute closure

wormhole node proc= local $ do
  conn <- open/reuse a connection with the node
  setState conn
  proc  <**  setState previous connection


teleport= local $ do
  Log _ log <- getState <|> Log False []
  yourClosureid <- getState <|> Closure 0
  connection <- getState
  send connection (myclosid,hisClosid, log)
```

wormhole set the new connection until the computation given as parameter finish. The purpose of the operator

<**

is to execute its second operand even if the first return *empty*.
listen generates a thread for each log that arrives trough the communication socket and set it in the state with the recovery flag on.
local is another name for *teleport* should send the log generated trough the socket that *wormhole* has opened.
listen and *teleport* instead of sending and receiving full logs from the beginning of the execution of each node, they may manage closure identifiers of the *teleport* that initiated the remote call. Son when the remote node respond, *listen* receive the closure that must be executed and a log containing what happened in the remote node.

Let's look how a program like the one below is executed in two nodes. There are three teleport calls. Each one of them send info about "who" called him (the identifier of the local closure) and a sequence of result containing its trace of execution since it was called. *listen* identifies the closure that performed the remote call and re-executes such closure with the log received. Since each closure has the previous execution state, it has all the previous variables generated in scope plus the new ones generated remotely, that have been recovered from the log.

```
listen..              .-->listen
wormhole...           ^   wormhole...
 local $ option...    ^    local $ option...
 x <- local$choose[1..10]  -- read 1,2.
 teleport -------------^   tel...
 local..  <-------.        localIO $ print x
 t <-local $...   ^        t <-local$return(x+2)
 tel...           ^--------teleport
 liftIO $ print y      .-->liftIO $ print y
 z <- return $  t +3   ^   z <- return $ t + 3
 teleport--------------^   tel...
 localIO $ print z         localIO $ print z
```

Additionally, to complicate this ping-pong example a little more, *choose* introduces threading and non-determinism. The continuation after *choose* will send the messages of all the threads trough the socket and would produce an stream of results in both nodes with the values expected.
Since each invocation produces a closure that must be stored with its identifier, the hash identifier of the closures must be defined carefully. two closures that correspond to the same path must have the same identifier, so old closures with the same identifier are garbage collected. This avoids space leaks.

16.1 Remote streaming

The above example send a stream of results from the "master" node to the remote node and back. To initiate the steaming from the remote node, simply it is necessary to have a non-deterministic/reactive/asynchronous primitive in the remote node:

```
do
  wormhole node $ do
    teleport
    -- in the remote node
    y <- local $ waitEvents foo
    teleport
    -- back in the local node
    liftIO $ print y
```

17 Distributed computing primitives

Over the basic primitives defined in the previous paragraphs, a set of higher level primitives can be constructed.

```
runAt node proc= wormhole node
                     $ atRemote proc

atRemote proc= loggedc $ do
   teleport   -- move to remote node
   r <- proc  <** setSlave
   teleport   -- back to initial node
   return r
```

atRemote executes a remote computation in a node and get the results back. Since the remote node is master during the time the procedure is executed, it is necessary a flag to notify the remote node that it is no longer the master. This must be done even if the procedure returns *empty*.

18 Moving computations

The remote node becomes master while executing the remote procedure, it can execute further wormholes and teleports in cascade to other nodes and back:

```
runAt node1 $ runAt node2 foo
```

The outer *wormhole* return a value computed in a third node Since streaming from any of the two sides is possible, we can code asynchronous messages of the Actor model synchronously, without breaking the flow of the code and losing composability.

19 Distributed computing algebra

runAt, as defined above, is composable with alternative, applicative and monadic combinators. Since a remote computation is similar to an asynchronous primitive from the point of view of the calling node, The semantic of parallelism and concurrency is identical:
This example sum the results coming from two nodes:

```
 r <- (+) <$> runAt node1 foo
          <*> runAt node2 foo
```

This example call two nodes in parallel. Get the first result and stop the second:

```
r <-  collect 1 $ runAt node1 proc <|>
                  runAt node2 proc
```

collect is a primitive, not detailed here, that kill the rest of the processes when there are the number of results desired. Remote actions can be sequenced using bind:

```
      r <- runAt node1 foo
      runAt node2 $ bar r
```

Mailboxes or *EVars* can be used for asynchronous communications between nodes:

```
runAt remoteNode $
          writeMailBox mboxname data

runAt remoteNode $ writeEVar ev data
```

Since *EVars* are not serializable, they should have been created in the local node previously. Non serializable data can be used in a distributed computation since the execution resume in the same closure after a remote invocation.

20 Connecting, executing in all the nodes

Higher level distributed computing primitives can be constructed using *runAt*

```
clustered proc= do
 nodes <- getNodes
 fold (<|>) empty (\n -> runAt n proc)  nodes

mclustered proc= do
 nodes <- getNodes
 fold (<>) mempty (\n -> runAt n proc)  nodes
```

clustered execute a single procedure in many nodes and return the results back, generating different threads in the calling node. *mclustered* use *monoid* to return an unique result.

21 Services

So far, the different nodes can communicate as soon as they share the same code, or at least the same execution routes. It is not necessary to share the same architecture. As we will see, a node running javascript ,generated from a Haskell program, in a browser can interact with the same program running in the server using distributed primitives. Conditional compilation can be used to eliminate code that is not relevant for each side.

It would be good to communicate completely different programs using a similar mechanism that may retain some properties such is reactivity, streaming and composability. *callService* allows to call a different remote program while retaining these properties.

Maintaining these properties means that a program can update a register by invoking a database service and simultaneously it can receive updates produced by other nodes accessing the same database. All with a single invocation of *callService*:

```
do
 reg <- newValue
 reg <- callService database reg
 localIO $ do
  putStr "register changed by another node : "
  print reg
```

In conventional frameworks that would require two completely different codebases and primitives for each of the two directions.

A primary service is the monitor. It installs, execute and re-execute another program or service when it is requested. The monitors of all the nodes are connected to distributed the load of programs and services.

22 Map-reduce

Map-reduce is a powerful and general distributed-computing pattern that can be used in widely different problems. Also is a hard test for the distributed primitives. Apache Spark[9] is the most known framework that implement map-reduce. As we said in the section of Distributed Computing, it brings a functional interface but internally it is not functional.

Over the primitives already described, we have implemented map-reduce. It distributes the computation among all the nodes connected. It mimic the basic behaviour and primitives of *spark*. Since it is composable it can be invoked anywhere in a transient program. This example makes use of it:

```
import Transient.MapReduce

main= keep $ initNode $ inputNodes <|> do
 content <- local $ input (const True) "text:"

 r<- reduce  (+) . mapKeyB (\w -> (w, 1))
                 $ distribute
                 $ V.fromList $ words content

 localIO $ putStr "result:" >> print r
```

This program executes the basic example that count the number of words in a text. first it distribute the text among the nodes, then the map generate 2-tuples with each world. Then, a local reduce in the node, sum all the tuples of the same word. Finally reduce is executed again in the calling node for all the results.

For performance reasons, it operates with boxed and unboxed vectors and use all the cores in each node. No benchmarks have been done to date.

23 Web programming

Thanks to the GHCJS compiler, it is possible to run transient programs in a web browser. Using WebSockets, browser and

server can communicate using distributed primitives. *listen* has been extended so that when it detect a HTTP request it send the compiled JavaScript program. The program then open a WebSocket connection using *wormhole*. Then the browser-server interaction can take place using the above mentioned primitives.
Browser and server can run different programs:

```
main= keep initNode $
        onBrowser browserprogram <|>
        onServer serverProgram ...

onBrowser f =
  if isBrowserInstance then f else empty
```

But a browser program can invoke the server using distributed primitives and the other way around. Any of them can be the master.
The same transient primitives and tecniques for de-inverting callbacks have been used for managing asynchronous events and managing WebSocket responses in the browser. For the creation of DOM elements in the browser, a sublanguage has been created. It is not in the scope of this article to detail these features.
This example takes input from a text box and print it in the server, the server send a response to the browser, that is displayed it in the browser:

```
browserProgram= onBrowser $ do
  rawHtml $ div ! id "result" $ noHtml
  name <- local $ render $
            getString Nothing fire OnKeyUp
  r <- atRemote $ do
         localIO $ print name
         return $ "hello " ++ name
  at "#result" Insert $ rawHtml $ p r
```

24 Future work

There are no performance comparison with other alternatives that have been discussed in the corresponding paragraphs. Some elements are in research state and must be improved in terms of robustness and performance. It is necessary to formalize and optimize the continuation engine.

Theoretically this model of concurrent and distributed computation can "run the formulas" of tensor algebra used for Machine Learning with parallelism and concurrency.

A set of higher level primitives can manage dynamically the distribution of data and computing to make distributed models like map-reduce a consequence of the availability of nodes. The programmer would write a program without awareness of the distribution.

Since inspecting of the state of each thread is possible, a general algorithm using Simulated Annealing or similar can dynamically modify parameters like the number of threads, buffer sizes in order to optimize certain performance criteria.

Identifying effects and state at the type level can help to make correct programs.

References

[1] Martín Abadi. 2016. TensorFlow: Learning Functions at Scale. *SIGPLAN Not.* 51, 9 (Sept. 2016), 1–1. DOI: http://dx.doi.org/10.1145/3022670.2976746

[2] Apache. 2016. Apache Spark. (2016). https://github.com/transient-haskell/transient-universe/blob/master/src/Transient/MapReduce.hs

[3] Continuation Fest 2008 Tokyo, Japan April 13, 2008 2008. *Clicking on Delimited Continuations.* Continuation Fest 2008 Tokyo, Japan April 13, 2008. http://okmij.org/ftp/Computation/Fest2008-talk-notes.pdf

[4] Albeto G. Corona. 2017. (2017). http://github.com/transient-haskell/transient

[5] Alberto G. Corona. 2017. Transient demo. (2017). https://github.com/transient-haskell/transient-examples/blob/master/distributedApps.hs

[6] Jan Dzik, Nick Palladinos, Konstantinos Rontogiannis, Eirik Tsarpalis, and Nikolaos Vathis. 2013. MBrace: Cloud Computing with Monads. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems (PLOS '13)*. ACM, New York, NY, USA, Article 7, 6 pages. DOI: http://dx.doi.org/10.1145/2525528.2525531

[7] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the Cloud. *SIGPLAN Not.* 46, 12 (Sept. 2011), 118–129. DOI: http://dx.doi.org/10.1145/2096148.2034690

[8] Jeff Epstein et Al. 2016. (2016). https://hackage.haskell.org/package/distributed-static-0.3.5.0

[9] Apache Foundation. (????).

[10] Simon Peyton Jones. 2010. Tackling the Awkward Squad:monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. (2010).

[11] Simon Marlow. 2016. (2016). https://hackage.haskell.org/package/async

[12] Microsoft. 2016. mbrace: Integrated Data Scripting for the Cloud. (2016). http://mbrace.io

[13] Heather Miller, Philipp Haller, and Martin Odersky. 2014. *Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution.* Springer Berlin Heidelberg, Berlin, Heidelberg, 308–333. DOI: http://dx.doi.org/10.1007/978-3-662-44202-9_13

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

[15] Peter Wegner. 1990. Concepts and Paradigms of Object-oriented Programming. *SIGPLAN OOPS Mess.* 1, 1 (Aug. 1990), 7–87. DOI: http://dx.doi.org/10.1145/382192.383004