Composing in the large

[a beautiful parade of the awkward Squad]

Alberto G. Corona
Axioma Consulting
Juan ABello 2, S.L.Escorial. 28200. Madrid. Spain
agocorona@gmail.com

ABSTRACT

In the year 2010, Simon Peyton Jones [1]. wrote: "Functional programming may be beautiful, but to write real applications we must grapple with awkward real-world issues: input/output, robustness, concurrency.."

The awkward squad includes anything that can not be expressed in beautiful mathematical formulas. Such kind of effects that forces the functional programmer get off the white horse of FP and and get stuck in the mud. Such code that can not be composed and reused with mathematical confidence. Loosely speaking, this includes any code whit IO, concurrency, threading , reactivity, Web requests, blocking, distributed computing, exceptions, interruptions, event handling, mutable state, long running loops and others. Although some of these issues can be reduced or decomposed into a mix of the others, There is a common issue among all of them: asynchronicity

Based on some unexplored techniques for managing logging, state, backtracking, closures and continuations in the language Haskell, I present here a solution that allows for the creation and composition of applications that manage these effect without breaking formulaic composability. The concrete mix of techniques makes a whole that is simple and easy to reason about. Although each technique would need a separate explanation, the coherence would be lost in the details. This article is a summary that demonstrates the expressive power of the mix.

I practical terms, this allows the use of FP techniques at an high level closer to software requirements and the feasibility of creating complex software components that can be combined with mathematical soundness.

Distributed computing is an extreme case that combines most of the components of the awkward squad. I demonstrate how a single Haskell expression can execute in many nodes with heterogeneous architectures, including web browsers, and can be composed with other expressions of similar complexity.
[500]Theory of computation Parallel computing models [500]Theory of computation Distributed computing models [500]Theory of computation MapReduce algorithms [500]Theory of computation Vector / streaming algorithms [500]Theory of computation MapReduce algorithms [500]Theory of computation Functional constructs [300]Theory of computation Concurrent algorithms [500]Software and its engineering Cloud computing [500]Software and its engineering Synchronization [300]Software and its engineering Publish-subscribe / event-based architectures [300]Software and its engineering Interoperability [300]Software and its engineering Checkpoint / restart [100]Software and its engineering Data flow architectures [100]Software and its engineering Requirements analysis [100]Software and its engineering Software design engineering [300]Information systems Web interfaces [300]Computing methodologies Vector / streaming algorithms

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications; D.1.1 [**Programming Techniques**]: Applicative(functiona) Programming; D.1.3 [**Programming Techniques**]: Concurrent Programming; H.5 [**INFORMATION INTERFACES AND PRESENTATION**]: Web-based interaction—*WebSockets*

General Terms

Theory

Keywords

Haskell, distributed, reactive, parallelism, concurrency, exceptions, threading, Web‘‘

1. INTRODUCTION

Naturally people express what they want in a declarative style that involve formulas and recipes. But when these descriptions are transformed into programs, the natural flow must be broken. The reason is a great number of asynchronous things that happen in the program, like events, communication messages, concurrency, exceptions, streaming.. Or at a even higher level, distributed computing or web programming that involves two or more program that communicate asynchronously.

There was a golden age of programming when programs ran for a single user at a time, in a sigle computer, they were single threaded, the only input was the keyboard and there were

only a single IO operation going on at the same time, and the main kind of applications were either numerical calculus for scientific purposes or for the enterprise. At that time it was possible to match the user requirements in the program in formulas and sequences of steps almost line by line. That is the kind of console application programs for which Unix and the rest of the OS's were created. Modern OS's compose blocking IO and CPU computation to create libraries. And libraries were the perfect software components to combine to create programs under these limitations. Personal computers were sold with a language, BASIC, that everyone should be capable to master for their needs.

This is no longer the case. The dominant kind of application is now the Web application. A web application can receive thousands of user requests. Each user can decide to go back and undo steps at any moment. It can receive asynchronous events from the cloud or the database. the input and outputs are not single values, but can be streams of values. Things are much more complex today.The asynchronous effects that these requirements demand, like browser events, requests, concurrency are not composable with current techniques.

I present here a set of techniques, a set of primitives made with these techniques and a set of libraries made with these primitives to overcome these kind of problems, so that the programmer can follow the recipes and formulas expressed by the client without regard for asynchronous things neither blocking neither threading neither distribution of computation, so the mistmatch between the level of the functionality description and the way it is coded is greatly reduced so that programming become more modular maintainable and understandable, and also has the mathematical soundness of functional programming.

A goal of computer science would be the establishment of an universal mathematical model for any kind of software. The main goal of software engineering would be a high level language that could express user requirements and make them run. A language with full reusability 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 easy to follow. 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 step we can go back and restart the execution at the most advanced point possible, leveraging the work already done.

Monads where introduced in Haskell and are not used in many languages to program effects. Informally, the semantic of recipes can be expressed in terms of monads with effects of backtracking, stopping/resuming and event handling.

Monads may implement these effects in a sequential composability in which a component (maybe an algebraic combination of components) produces a result that is an input parameter for the next step.

The question is why software can not be expressed in clear formulas (algebra) and recipes (sequences of steps)?

The main obstacle until now, was the absence of a way to express algebraic and monadic composability when there are breaks of the execution flow by events raised from interruptions, web requests, GUIs, blocking IO or long running loops.

Transient leverages continuations, event handling and multithreading in a way that the execution flow is not broken in disjoint event handlers and the execution does not block even in the presence of events and blocking IO operations, so that these effects can be composed algebraically and monadically as combinations of processes executing parallelism and concurrency.

It can be put in the following terms: let ap1 and ap2 two applications with arbitrary complexity, with all effects including multiple threads, asynchronous IO, indeterminism, events and distributed computing.

Then the combinations:

```
- ap1 <|> ap2          -- Alternative, parallelism
- ap1 >>= \x -> ap2    -- monadic sequence
- ap1 <> ap2           -- monoidal, concurrency
- (,) <$> ap1 <*> ap2  -- Applicative, concurrency
```

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

1.1 Demo

I would like to present an application [**?**] made of composable widgets. Each widget implements a complete distributed application. One of them implements a map-reduce operation using the Apache/Spark[**?**] dataflow model. Another implements a distributed chat using the actor model. Both compose a single web app that runs in all the nodes connected.

2. A REVISIT OF THE MONAD

The essential problem that break the flow of execution are whatever that is asynchronous. The natural way to deal with the asynchronous problems in functional programming has been whatever form of continuation. But continuations are hard to manage. First they are too powerful and generates code that is hard to understand. In the other side due to that power, they usually don't compose well following algebraic rules. The continuation monad has not been

used to implement , concurrency and event handling despite the fact that is theoretically ideal for making asynchronous events synchronous.

As an alternative, there is a parallelism repeated often between monads and continuations. Monads can be the key to manage continuations. If we look at the definition of bind:

```
g >>= f   = ...
```

applied to the execution of this monadic sequence:

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

This is equivalent to:

```
a >>= b >>= c >>= d
```

At the moment of the second bind we can see that g is the result of

```
a>>=b  ===  b x
```

g can be considered as the closure and f is the continuation:

```
f= \x y -> c >>= d
```

In the definition of bind, we can store g and f. That makes it available for any primitive that we may need to manipulate the execution of the monad.

For example, if 'b' in the previous expression contains an asynchronous IO operation, we can re-execute his closure g and his continuation f. With this we have managed to handle an asynchronous event as if they were synchronous, without breaking the monadic composability. We have injected an event in the middle of the computation.

Such primitive that wait for events and inject them in the monadic commputation is called '*waitEvents*

3. HANDLING EVENTS

waitEvents execute the continuation whenever an event happens. It is assumed that events are read by a blocking IO computation that is rescheduled inmediately 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
     forkIO $ do
          msg <- blockingIO
          putInBuffer msg
          (closure,continuation)<- get
          forkIO $ closure >>= continuation
     empty
    else
     empty buffer
     return buffer
```

Why execute the closure again and not just the continuation? because *waitEvents* can be an element of a more complex expression. The first term of bind can be a complex computation involving *applicative*, *alternative* and monadic combinators of which *waitEvents* may be a term.

When the closure is re-executed, textitwaitEvents find the buffer filled, so the closure is recalculated and the continution is re-executed.

Observe that there is a new thread for each event. Also, it return *empty* to the current thread. This allows parallelism and concurrency, as we will see later.

Assuming that *keep* is the runner of the monad,

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

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

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

Has some good properties:

- is Reactive, since respond to multiple events
- it is intuitive since it is console-like with monadic composability.
- It handles an stream of messages.
- Is non blocking
- It is multithreaded and parallel if inputs are entered fast enough.
- It is loop free, but works as if it were in a loop.
- It can be composed algebraically as we will see below.

sectionAlternative processing / indeterminism 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
```

However, within a single thread, the operator \textit{<|>} and \textit{empty} follow the alternative laws, so the framework can't do even the alternative laws,

This composition introduces a kind of non-determinism which is multi-threaded. As a consequence, a *TransIO* computation may return 0, 1 or N results

3.1 IRC client

An example of the last composition is this snippet that uses '*waitEvents*' both for input and output:

```
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= threads 1 $
   waitEvents (hGetLine in)  >>= liftIO . hPutStrLn out
```

There are some variants of *waitEvents* defined:

```
-- 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
```

parallel re-executes the blocking IO computation until there are no more events.

sample execute a polling IO with a certain frequency and inject the result when the value changes.

4. DE-INVERTING EVENT HANDLERS

Event handlers break monadic composition. Every framework has some kind of inversion of control, and the programmer is forced to write his code in a set of disjoint handlers. Little can be done to express his program in a clear sequence. Unless we create some primitive that pass our continuations to and from the framework, yet to follow the alternative laws, so that the program donÂt't get broken. This is done by react:

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

Here *setHandler* is the event handler installer that the frameworks use. It has the signature of a continuation, and in fact it is so. It has a parameter, the event handler.

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

toreturn compute the *a* result to be returned to the framework by the handler. Usually, it is *return()*

```
react
```

de-invert the event handler so that, for the programmer, the flow continues downstream, without breaking composability.

Assuming that setCallback in the installer of callbacks in the framework

```
x <- before
event <- react setCallaback (return ())
y <- after event
after2 y
```

react install the callback if *before* return a result. *after* is called with the event when the callback is called

react has a similar structure than *waitEvents*, but there are no thread creation, since in this case, they are generated by the framework.

Like *waitEvents*, it also allows *applicative* and *alternative* composition.

...

5. CONCURRENCY: MONOID AND APPLICATIVE OPERATORS

If two asynchronous processes are combined with an applicative expression, the result is a single result that run 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 buffer of the other operand. If it is filled, it return the result of the *applicative*. If it is not, this thread that finish first dies. So the thread that finished last is the one that continues execution.

To store and read the result of each term in a mutable variable it is necessary to interpolate the statement that read the result of the other operand before the execution of the continuation. That is the reason why clousure and continuation are stored separated in the state.

6. ALGEBRAIC/MONADIC COMPOSITION WITH PARALLELISM/CONCURRENCY AND EVENTS

With the *applicative* operator we can define algebras of binary operations that describe domain problems without regard for the effects that they must manage.

We can maintain a numeric algebra

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

no matter if the calculation of mathematical formula is distributed among many nodes or some operands are a streams of values injected in the formula by means of callbacks.

For example

```
sold :: TransIO Double
sold = price * quantity

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

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

do
    q <- sold
    liftIO $ putStr "Event: income: " >> print q
```

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

In the same way a relational algebra is possible

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

Or the algebra corresponding to any domain problem. In combination with monadic composition, it is possible to program at a higher level without breaking functional programming style and achieving a level of expression that can match user requirements very closely.

This also allows the creation of complex components that can be reused as easily as any formula in larger algebraic and monadic expressions.

7. CONSOLE INPUT WITH THREADING

Sharing the standard input among different threads is important for composing modules that are controlled by the keyboard. If the input line is put in a buffer, with the aid of '*waitEvents*' we can create an asynchronous console input primitive for the creation of menus that do not block, so that they can be composed while at the same time they are all active waiting for input:

```
option v text = do
    liftIO $ print text
    waitEvents loop
    where
    loop= do  b <- read buffer
              if b==v then  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"
```

The alternative expression, with the operator <|> combines two monadic programs that run in two different threads. Both share the same continuation, that prints "next":

An example of interaction of this program:

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

7.1 Command line input

the input can be read from the command line:

```
>program -p b/a/b
world
next
hello
next
world
next
```

7.2 Input with prompt and validation

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

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

8. THREAD CONTROL

Since *waitEvents* produce a new thread for each input, this can be overkill. Too much parallelism may not be optimal:

We can have a lot of light asyncronous tasks:

```
proc= async foo <|>
      async bar <|>
      async baz <|>
      ...
```

Or we can have a list of inputs that we need to process in parallel:

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

Expression like this may produce an explosion of threads if there is no way to control them.

If the number of threads are limited and the number of threads is exhausted, async primitives do not execute the continuation in a new thread but within the current thread.

```
  do
       x <- threads 2 proc
       liftIO $ print x
```

The above example execute *proc* with two threads

The code below perform a loop within the current thread, since there are no extra thread available:

```
  threads 0 $ do choose[1...10]; liftIO $ print x
```

instead of controlling threads one by one, or in groups using pools, threads can be managed using the precedence hierarchy of the program. This does not introduce additional complexities.

killChilds is a primitive without arguments that kill all the child threads of the current thread. *oneThread* would kill all the sibling threads.

killing threads is sometimes necessary. For example, each execution of *waitEvents* or *option* add a new listener that wait for events in a loop. It is necessary to kill the previous one when a new is started by the previous monadic statement:

```
do
    oneThread $ option "menu" "view menu"
    op <- option "op1" "op1" <|> option "op2" "op2"
    if op== "op1" then longRunning  else longRunningâĂŹ
```

In this example, if "menu" is pressed, *oneThread* will kill the two options and the long running processes that may have been initiated previously.

9. STATE

In real world programming, frequently new state data is needed to implement new functionalities. To make Haskell a productive language it is important to ease the definition, creation and deletion of new state data. I believe that state monad transformers are not flexible enough and demand a level of knowledge of the language that is not reachable by everyone. That is partially responsible for the great demand of extensible records: Programmers often use extensible records to avoid multiple state monad transformers.

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 is inherited and continues trough asynchornous primitives that generate new threads.

state primitives can be used with user-defined types:

```
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= getState <|> return initialValue
```

This last expression above 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 used internally by transient to implement some primitives.

Since the state is pure, it does not "travel" back:

```
do
   setState "hello"
  choose [1..] -- many threads produced
  x <- getState
  liftIO $ print x
    setState "world"
...
```

prints "hello" even if the state is changed downstream

Since transient computations typically have no loops. If we want to communicate changes upstream of the computation, locally defined mutable variables can be used.

10. REACTIVE CHANNELS

as long as it is in scope, mutable variables can be used to communicate data between two branches executing two different threads of a transient expression. If we want to inject events and trigger threads in other branch of the execution, we can construct reactive channels that have the semantic of publish-subscribe.

EVars are unbounded reactive channels defined as such:

```
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* to produce the reactive effect.

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

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

```
 do
   ev <- newEVar
   r  <- readEVar ev <|> readEVar ev <|> (mapM_ (writeEVar
   liftIO $ putStr r

> 112233...
```

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

11. TRANSACTIONS AND EXCEPTIONS

Transactions and exceptions are big computing problems. 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 rogramming in the large, compensations are usually the only technique possible.

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.

The basic primitives that we want:

- execute the backtracking by executing the closures stored for it (*undo*)

- defining a handler (*onUndo*) that may undo changes
- suspend backtracking and retry for 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 is 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 undo stack
       execute action
     else handler

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

‘forward‘ unset the backtracking flag
```

This same mechanism can be used to execute exception handlers with the addition of an extra paramenter: the exception.

onException is similar to onBack but with an extra parameter.

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

Treating exceptions trough this same mechanism also makes monadic composition cleaner, since exception handlers do not need to wrap around all the rest of the computation and the computation has no additional bifurcations of code. It also works with multiple threads, as we have seen.

This code is used to restart a network node when the connection fails:

```
onException $ \(e :: ConnectionError) -> restart node >> co
runAt node foo
```

This general backtracking engine unifies transactions and exceptions as well as resource management. instead of retry, ‘onException‘ can close some resource and left the backtracking mechanism continue executing more exception handlers, as in the case of the book reservation example.

12. LOGGING/RECOVERY

I call to attention an effect not used very much, but which is very powerful: The storage of intermediate execution result in order to use them to recover the computation state in the same or different machine. ‘textitlogged‘ is a primitive that given a computation, add its result to the log state and return the value.

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

The log state is defined as such:

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

 LogElem=  Var IDynamic | Exec | Wait

 data Log= Log Recovery [LogElem]
```

An element of the log is either a result, a seralization, a asynchonous operation (*Wait*) or is being executed and has not finished (*Exec*).

if the Recover flag is on, textitlogged , run in recovery mode and read the next element of the log instead of excuting his argument.

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

> "hello"
```

A pseudocode for the ‘textitlogged‘ 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
```

Note that, before *f* is executed, the log is set to *Exec.* if f is asynchronous the log element would be set to *Wait* and after the execution, it is set to the value returned.

After the execution of *f*, all the log trace that may have been generated internally is discarded. Only his result in put in the log. The log stores only the top level results. It would contain only the minimal "route" necessary to recover the computation state.

13. REMOTE EXECUTION

With the primitives defined above for logging and recovery, it is possible to start a computation in a network node and continue in another node.

Let's hypothesize three primitives: a socket listening primitive: *listen*, a connection primitive *wormhole* and a send primitive *teleport* that allows the execution of *foo* in local node and print the result in a remote node:

```
listen <|> return ()
local (option "s" "start")
r <- local  foo
wormhole node $ do
   teleport
   localIO $ print r
```

Note that, in the example, the program can be executed either for a local user command "s" or by a remote message received by listen, since *return ()* immediately is followed by the execution of *option*.

```
local= logged

listen= do
    log <- waitEvents $ read socket messages
    setData $ Log True log

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


teleport= local $ do
    Log recover log <- getState <|> Log False []
    conn <- getState
    send conn  log
```

wormhole set the new connection until the computation given as parameter finish. The purpose of the operator $<**$ is to execute his 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 textitlogged. Since it is logged, teleport should send it trough the socket that wormhole has opened with the node.

in a more optimized implementation, *listen* and *teleport* so that instead of sending and receiving full logs from the beginning of execution, may contain closure identifiers of the teleport that initiated the remote call plus a shorter log containing what happened in the remote node.

Let's look at 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 his 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..from log
  teleport -------------^       teleport
  local..  <-------.            localIO $ print x
  t <-local $      ^            t <-local $ return $ x + 1
  teleport         ^------------teleport
  liftIO $ print y       .----->liftIO $ print y
  z <- return $  t +3    ^      z <- return $ t + 3
  teleport---------------^      teleport
  localIO $ print z             localIO $ print z
```

Additionally, to complicate it a little more, *choose* introduces threading and non-determinism. The same logging-communication-recovery would serialize the messages of all the threads trough the socket opened by wormhole and would produce an stream of results in both nodes with the values expected.

Since each remote invocation produces a closure that must be stored with his 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.

13.1 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      -- moves to the remote node
      y <- local $ waitEvents foo  -- produces events
      teleport      -- move event y to the initial node
      liftIO $ print y -- print in the initial node
```

13.2 Ping-pong program

This example executes the ping-pong program with a remote node. Since it is a loop it excute endless teleports. In each one, the computation is translated from a node to the other.

```
pingPong node= wormhole node pingPongâĂŹ

pingPong'= do
      localIO $ print "ping"
      teleport
      localIO $ print "pong"
      teleport
      empty the Log
      pingPong'
```

Since the log grows with each invocation and does not carry additional information, an optimization of the code would empty the log before each iteration. This pattern for iteration without log growth could abstracted in a simple primitive.

14. DISTRIBUTED PRIMITIVES

Over the basic primitives, a set of higher level primitives can be constructed.

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

atRemote proc= loggedc $ do
     teleport   -- move to the 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 is no longer the master at the end of it. This must be done even if the procedure return *empty*.

15. MOVING COMPUTATIONS

Since the remote node becomes master, it can execute further wormholes and teleports to other nodes:

```
wormhole node1 $ do
    teleport
    wormhole node2 $ do
        teleport
```

16. 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.

Sequencing of remote actions:

```
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 thanks to the reuse of closures, since they maintain non serializable data in scope between remote invocations.

17. CONNECTING, EXECUTING IN ALL THE NODES

Higher level distributed computing primitives can be constructed using 'runAtÂt'

```
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.

18. INITIALIZATION

for initialization of a distributed application some utility procedures are defined.

The goal is to initiate 'listen' to recive messages in a socket and to set up the list of the nodes, this initialization feed the programs that participate in the distributed computation:

```
main= keep . initNode $ inputNodes <|>
                        program1   <|>
                        program2...
```

inputNodes get the node information using *option* and *input*. So the configuration can be given in the command line or interactive:

```
> program -p start/hostname1/port1/
     add/hostname2/port2/y/add... &
```

The pseudocode of these primitives are:

```
data Node= Node Host Port Services

initNode comp= do
     node <- Node <$> input host <*> input port
     listen node <|> return()


inputNodes= do
     option "add" "add a node"
     node <- Node <$> input host <*> input port
     c <-  local $ input (\x -> x=="y" || x== "n")
            "interchange nodes lists?"
     if c then connect node else addNode node
```

connect synchronize the list of nodes with the new node using distributed primitives.

19. 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 may see, a node running javascript in a browser and a server can interact using distributed primitives.

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

Since it is reactive and streaming, a program could stream local changes of a register to a database and simultaneously receive changes from this service by means of a single *callService*:

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

Since it is composable, this snippet can be composed with any other. That allows to program the application in composable pieces at the level of services and at the level of each components of a service.

```
data Service= Key-Value map

 -- run a node as service
runService :: Service -> Cloud a -> Cloud a
-- call a service by name
callService  :: Key -> Service -> parameters -> Cloud a

-- call a service by node
callService' :: Key -> Node -> parameters -> Cloud a
```

A system of authorizations has been defined discriminating by key, by service data, optionally over TLS communications.

A primary service is the monitor, that installs, execute and re-execute on request

```
main= keep $ runService monitor $ \service -> do
     findInNodes service <|> runHere <|> runThere
     where
     runHere= do
       checkSuitable key service
       install
       run
       return node
     runThere= ask supervisors in other machines
```

The supervisors of all the nodes are connected. New algorithms for distribution of the load can be developed over this.

Depending on the service description, the monitor can execute it if it is installed or can install it. git-cabal, stack, and Docker packages are supported or planned to support.

```
\textit{callService} invokes the local monitor. Also initiates it if necessary

callService pin service params = do
    node <- findInNodes service <|> do
        onException $ (\e :: ConnectionError) -> do
                                          startMonitor
                                          forward
        callServiceâĂŹ pin  monitorService (pin, service)

    add node to local list of nodes
    callServiceâĂŹ node params
```

20. 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 is the most known framework that implement map-reduce for general problems. It brings a functional interface but internally it is not functional neither composable. That is the reason why it is a framework and not a library.

Transient has a module that execute map-reduce in all the nodes connected. It mimic the basic behaviour and primitives of spark. It perform a similar distribution of work among the nodes. Since it is composable monadically, it can be invoked anywhere in a transient program:

```
import Transient.MapReduce

main= keep $ initNode $ inputNodes <|> do
   content <- local $ input (const  True) "enter 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 executes again this sum in the calling node for all the node results.

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

21. WEB PROGRAMMING

Given the completeness of the GHCJS compiler, it is possible to make run transient programs in a web browser. Using websockets communications, browser and server can communicate using distributed primitives. ‘listen‘ is extended so that when it detect a HTTP request it send the compiled JS program. The program then open a websocket connection using wormhole and the distributed interaction can take place

Although the code is the same, browser and server can run different programs:

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

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

This example takes input from a text box. Print in the server, receive a message and display it in the browser:

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

    at "#result" Insert $ rawHtml $ p r
```

22. REFERENCES

[1] S. P. Jones. Tackling the awkward squad:monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. 2010.