Distributed Programming with Cloud Haskell

Andres Löh

14 June 2013, Big Tech Day 6 — Copyright © 2013 Well-Typed LLP
Overview

- Introduction
- Haskell
- Cloud Haskell
- Communication
- Going distributed
- Towards Map-Reduce
Introduction
What is Cloud Haskell?

- Framework (a number of related packages) for Haskell
- Message-passing distributed concurrency (Erlang, actors)
- All in libraries; no (specific) compiler support required
Features

- Global view on a distributed program
- Single program runs in potentially many places
- Processes and nodes are first class entities
- Communication via (typed) messages
- Functions can be sent
- Programmable serialization
- Easy to monitor processes (and recover from failure)
- (Draft of) formal semantics
Many approaches

Different problems have different requirements / cost models.
Many approaches

Different problems have different requirements / cost models.

Concurrency

- threads and locks (MVar s)
- asynchronous computations (Async s)
- software transactional memory
- ...
Many approaches

Different problems have different requirements / cost models.

(Deterministic) Parallelism

- evaluation strategies
- dataflow-based task parallelism
- flat and nested data parallelism
- ...
Many approaches

Different problems have different requirements / cost models.

Distributed Concurrency

- Cloud Haskell
- ...
Freedom of choice

- Haskell is great for embedded domain-specific languages.
- GHC has a very capable run-time system.
- You can pick whatever suits the needs of your task.
- All the approaches can be combined!
Freedom of choice

- Haskell is great for embedded domain-specific languages.
- GHC has a very capable run-time system.
- You can pick whatever suits the needs of your task.
- All the approaches can be combined!

Lesson

Rather than picking a language based on the model you want, pick a library based on the problem you have.
server :: Process ()
sserver = forever $ do
  () ← expect
  liftIO $ putStrLn "ping"

cclient :: ProcessId → Process ()
cclient serverPid = forever $ do
  send serverPid ()
  liftIO $ threadDelay (1 * 10^6)

main :: IO ()
main = do
  Right t ← createTransport "127.0.0.1" "201306"
  defaultTCPParameters
  node ← newLocalNode t initRemoteTable
  runProcess node $ do
    serverPid ← getSelfPid
    spawnLocal $ client serverPid
  server

Cloud Haskell Example
Haskell
Pure Functions

dist :: Floating a ⇒ a → a → a
dist x y = sqrt (x * x + y * y)
Pure Functions

\[
\text{dist} :: \text{Floating } a \Rightarrow a \rightarrow a \rightarrow a \\
\text{dist } x \ y = \sqrt{(x \times x + y \times y)}
\]

\[
\textbf{data} \ \text{Tree } a = \text{Leaf } a \mid \text{Node } (\text{Tree } a) (\text{Tree } a)
\]

\[
\text{size} :: \text{Tree } a \rightarrow \text{Int} \\
\text{size } (\text{Leaf } n) = 1 \\
\text{size } (\text{Node } l \ r) = \text{size } l + \text{size } r
\]
Pure Functions

\[
\text{dist} :: \text{Floating } a \Rightarrow a \rightarrow a \rightarrow a \\
\text{dist } x \ y = \sqrt{x \times x + y \times y}
\]

\[
\text{data } \text{Tree } a = \text{Leaf } a \mid \text{Node } (\text{Tree } a) (\text{Tree } a)
\]

\[
\text{size} :: \text{Tree } a \rightarrow \text{Int} \\
\text{size } (\text{Leaf } n) = 1 \\
\text{size } (\text{Node } l \ r) = \text{size } l + \text{size } r
\]

\[
\text{search} :: \text{Eq } a \Rightarrow \text{Tree } a \rightarrow a \rightarrow \text{Bool} \\
\text{search } (\text{Leaf } n) x = n = x \\
\text{search } (\text{Node } l \ r) x = \text{search } l \ x \mid\mid \text{search } r \ x
\]
Type signatures

\[ \text{dist} :: \text{Floating } a \Rightarrow a \to a \to a \]
\[ \text{size} :: \text{Tree } a \to \text{Int} \]
\[ \text{search} :: \text{Eq } a \Rightarrow \text{Tree } a \to a \to \text{Bool} \]
Function calls

dist :: Floating a ⇒ a → a → a

\[
dist x y \\
dist 2 3 \\
dist (2 + x) (3 + x)
\]
conversation :: IO ()
conversation = do
  putStrLn "Who are you?"
  name ← getLine
  putStrLn $ "Hi " ++ name ++ ". Where are you from?"
  loc ← getLine
  putStrLn $ if loc == "Munich"
              then "Oh, I love Munich!"
              else "Sorry, where is " ++ loc ++ "?"
conversation :: IO ()
conversation = do
  putStrLn "Who are you?"
  name ← getLine
  putStrLn $ "Hi " ++ name ++ ". Where are you from?"
  loc ← getLine
  putStrLn $ if loc == "Munich" then "Oh, I love Munich!" else "Sorry, where is " ++ loc ++ "?"

readNLines :: Int → IO [String]
readNLines n = replicateM n getLine
Monads

- Maybe a -- possibly failing
- State s a -- state-maintaining
- Random a -- depending on a PRNG
- Signal a -- time-changing
- Par a -- annotated for parallelism
- IO a -- arbitrary side effects
- STM a -- logged transactions
- Process a -- Cloud Haskell processes
...

"Semicolon" is overloaded. You can define your own "monads". You can decide what the semantics of sequencing in your application should be.
### Monads

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maybe</td>
<td>possibly failing</td>
</tr>
<tr>
<td>State</td>
<td>state-maintaining</td>
</tr>
<tr>
<td>Random</td>
<td>depending on a PRNG</td>
</tr>
<tr>
<td>Signal</td>
<td>time-changing</td>
</tr>
<tr>
<td>Par</td>
<td>annotated for parallelism</td>
</tr>
<tr>
<td>IO</td>
<td>arbitrary side effects</td>
</tr>
<tr>
<td>STM</td>
<td>logged transactions</td>
</tr>
<tr>
<td>Process</td>
<td>Cloud Haskell processes</td>
</tr>
</tbody>
</table>

“Semicolon” is overloaded

You can define your own “monads”. You can decide what the semantics of sequencing in your application should be.
Concurrency

forkIO :: IO () → IO ThreadId

Well-Typed
Concurrency

forkIO :: IO () → IO ThreadId

threadDelay :: Int → IO ()

forever :: Monad m ⇒ m a → m b  -- here:  IO a → IO b

printForever :: String → IO ()
printForever msg = forever $ do
  putStrLn msg
  threadDelay (1 * 10^6)

main :: IO ()
main = do
  forkIO $ printForever "child 1"
  forkIO $ printForever "child 2"
  printForever "parent"
Concurrency

forkIO :: IO () → IO ThreadId

threadDelay :: Int → IO ()

forever :: Monad m ⇒ m a → m b -- here: IO a → IO b

printForever :: String → IO ()
printForever msg = forever $ do
  putStrLn msg
  threadDelay (1 * 10^6)

main :: IO ()
main = do
  forkIO $ printForever "child 1"
  forkIO $ printForever "child 2"
  printForever "parent"
Cloud Haskell
server :: Process ()
server = forever $ do
  () ← expect
  liftIO $ putStrLn "ping"

client :: ProcessId → Process ()
client serverPid = forever $ do
  send serverPid ()
  liftIO $ threadDelay (1 * 10^6)

main :: IO ()
main = do
  Right t ← createTransport "127.0.0.1" "201306"
  defaultTCPParameters
  node ← newLocalNode t initRemoteTable
  runProcess node $ do
    serverPid ← getSelfPid
    spawnLocal $ client serverPid
    server
Layered architecture

Over-simplified:

1. **User application**
2. **Higher-level libraries**
3. **Distributed process core library**
4. **Backend** (simplelocalnet, Azure, EC2, ...)
5. **Transport** (TCP, in-memory, SSH, ZeroMQ, ...)
6. **System libraries**
Nodes, Processes, Communication

Well-Typed
Nodes, Processes, Communication

- Backend responsible for nodes
- Processes and communication are backend-agnostic
Spawning and running processes

\[
\text{spawnLocal :: Process} \; () \rightarrow \text{Process ProcessId}
\]

\[
\text{spawn :: NodeId} \rightarrow \text{Closure (Process} \; ()\text{)} \\
\rightarrow \text{Process ProcessId}
\]

For the main process:

\[
\text{runProcess :: LocalNode} \rightarrow \text{Process} \; () \rightarrow \text{IO} \; ()
\]
Sending and receiving messages

Ad-hoc:

send :: Serializable a ⇒ ProcessId → a → Process ()
expect :: Serializable a ⇒ Process a
expectTimeout :: Serializable a ⇒ Int → Process (Maybe a)

Sending is asynchronous. Receiving blocks.
Sending and receiving messages

Ad-hoc:

send :: Serializable a ⇒ ProcessId → a → Process ()
expect :: Serializable a ⇒ Process a
expectTimeout :: Serializable a ⇒ Int → Process (Maybe a)

Sending is asynchronous. Receiving blocks.

Typed channels:

newChan :: Serializable a ⇒ Process (SendPort a, ReceivePort a)
sendChan :: Serializable a ⇒ SendPort a → a → Process ()
receiveChan :: Serializable a ⇒ ReceivePort a → Process a
...

Well-Typed
Serializable a = (Typeable a, Binary a)
Serializable a = (Typeable a, Binary a)

Typeable a -- has a run-time type representation
Binary a -- has a binary representation
Haskell’s typing discipline

Haskell is a statically typed language, but can be dynamically typed locally, on demand.
Haskell’s typing discipline

Haskell is a statically typed language, but can be dynamically typed locally, on demand.

\[
\text{typeOf} :: \text{Typeable } a \Rightarrow a \rightarrow \text{TypeRep} \\
\text{toDyn} :: \text{Typeable } a \Rightarrow a \rightarrow \text{Dynamic} \\
\text{fromDynamic} :: \text{Typeable } a \Rightarrow \text{Dynamic} \rightarrow \text{Maybe } a
\]

GHC can “derive” an instance of \textbf{Typeable} for any datatype automatically.
Binary representation

encode :: Binary a ⇒ a → ByteString
decode :: Binary a ⇒ ByteString → a

- Haskell has no built-in serialization.
- Automatic generation of sane \textit{Binary} instances for many datatypes possible via datatype-generic or meta-programming.
- Programmer has control – instances can deviate from simply serializing the in-memory representation.
Communication
Idea

Messages can include process ids and channel send ports.
How to reply

Idea

Messages can include process ids and channel send ports.

Old server:

```
server :: Process ()
server = forever $ do
  () ← expect
  liftIO $ putStrLn "ping"
```
How to reply

Idea

Messages can include process ids and channel send ports.

New server:

```haskell
server :: Process ()
server = forever $ do
  clientPid ← expect
  liftIO $ putStrLn $ "ping " ++ show clientPid
  send clientPid ()
```
Adapting the client

Old client:

\[
\text{client} :: \text{ProcessId} \rightarrow \text{Process} ()
\]

\[
\text{client serverPid} =
\]

\[
\text{forever} \quad \text{do}
\]

\[
\text{send serverPid} ()
\]

\[
\text{liftIO} \; \text{threadDelay} \; (1 \times 10^6)
\]
Adapting the client

Old client:

```
client :: ProcessId → Process ()
client serverPid =

  forever $ do
    send serverPid ()

  liftIO $ threadDelay (1 * 10^6)
```
New client:

```haskell
client :: ProcessId → Process ()
client serverPid = do
  clientPid ← getSelfPid
  forever $ do
    send serverPid clientPid
    () ← expect
    liftIO $ putStrLn "pong"
    liftIO $ threadDelay (1 * 10^6)
```

Well-Typed.
Well-Typed.
More about replying

- We can send ids of other processes.
- Forwarding, redirection, broadcasting.
More about replying

- We can send ids of other processes.
- Forwarding, redirection, broadcasting.

For typed channels:

- We can serialize `SendPort`.
- But we cannot serialize `ReceivePort`.
Some rules about exchanging messages:

- only one mailbox per process;
- we can expect a particular type;
- we can receiveWait for specific messages;
- typed channels are separate;
- sane ordering of messages;
- messages may remain undelivered.
Going distributed
Distributed ping-pong

No changes to **server** and **client** are needed.

Old **main**:

```haskell
main :: IO ()
main = do
  Right t ← createTransport "127.0.0.1" "201306"
  defaultTCPParameters
  node ← newLocalNode t initRemoteTable
  runProcess node $ do
    serverPid ← getSelfPid
    spawnLocal $ client serverPid
  server
```
Distributed ping-pong

No changes to server and client are needed.

New main (using distributed-process-simplelocalnet):

```haskell
main :: IO ()
main = do
  args ← getArgs
  let rtbl = __remoteTable initRemoteTable
  case args of
    ["master", port] → do
      backend ← initializeBackend "127.0.0.1" port rtbl
      startMaster backend master
    ["slave", port] → do
      backend ← initializeBackend "127.0.0.1" port rtbl
      startSlave backend
```
Automatic detection of slaves

\[ \text{startSlave} :: \text{Backend} \rightarrow \text{IO} () \quad -- \text{does nothing} \]

\[ \text{startMaster} :: \text{Backend} \rightarrow ([\text{Node Id}] \rightarrow \text{Process} ()) \rightarrow \text{IO} () \]
Automatic detection of slaves

\[
\begin{align*}
\text{startSlave} &:: \text{Backend} \rightarrow \text{IO} () \quad -- \text{does nothing} \\
\text{startMaster} &:: \text{Backend} \rightarrow ([\text{NodeId}] \rightarrow \text{Process} ()) \rightarrow \text{IO} ()
\end{align*}
\]

Master gets node ids of all slaves.
master :: [NodeId] → Process ()

master slaves = do
  serverPid ← getSelfPid
  forM_ slaves $ λnid → spawn nid ($(mkClosure 'client) serverPid)

server
Spawning functions remotely

```
master :: [NodeId] → Process ()
master slaves = do
  serverPid ← getSelfPid
  forM_ slaves $ λnid → spawn nid ($ (mkClosure 'client) serverPid)
server
```

Spawns a function call on a remote node.
Serializing functions

- “Single program assumption”
- Top-level functions are easy
- (Partially) applied functions are turned into closures
Serializing functions

- “Single program assumption”
- Top-level functions are easy
- (Partially) applied functions are turned into closures

- Currently based on a bit of meta-programming.
- In the future perhaps using a (small) compiler extension.
Towards Map-Reduce
Distributing actual work

master :: [Input] → [NodeID] → Process ()
master inputs workers = do
  masterPid ← getSelfPid
  workerPids ← forM workers $
    \lambda \text{nid} \rightarrow \text{spawn} \ \text{nid} ((\text{mkClosure} \ \text{worker}) \ \text{masterPid})$

  forM_ (zip inputs (cycle workerPids)) $
    \lambda \text{input, workerPid} \rightarrow \text{send} \ \text{workerPid} \ \text{input}$

  r ← collectResults (length inputs)
  liftIO $\text{print} \ r$
Distributing actual work

master :: [Input] → [NodeId] → Process ()
master inputs workers = do
  masterPid ← getSelfPid
  workerPids ← forM workers $ λnid → spawn nid ($ (mkClosure worker) masterPid)
  forM_ (zip inputs (cycle workerPids)) $ λ(input, workerPid) → send workerPid input
  r ← collectResults (length inputs)
liftIO $ print r
Workers

... workerPids ← forM workers $ λ nid → spawn nid ($ (mkClosure 'worker) masterPid)
...

worker :: ProcessId → Process ()
worker serverPid = forever $ do
  x ← expect -- obtain function input
  send serverPid (expensiveFunction x)

The expensiveFunction is “mapped” over all inputs.
Collecting results

... 
\[ r \leftarrow \text{collectResults} \text{ (length inputs)} \]
liftIO \$\ print r
... 

\[
\text{collectResults} :: \text{Int} \rightarrow \text{Process Result} \\
\text{collectResults} = \text{go emptyResult}
\]

\textbf{where}

\[
\text{go} :: \text{Result} \rightarrow \text{Int} \rightarrow \text{Process Result} \\
\text{go !acc 0} = \text{return acc} \\
\text{go !acc n} = \textbf{do} \\
\hspace{1em} r \leftarrow \text{expect} \quad -- \text{obtain one result} \\
\hspace{1em} \text{go (combineResults acc r) (n - 1)}
\]

In \text{go} we “reduce” the results.
Abstraction and variation

- Abstracting from `expensiveFunction`, `emptyResult`, `combineResults` (and inputs) yields a simple map-reduce function.
- Can easily use other ways to distribute work, for example work-stealing rather than work-pushing.
- Can use a hierarchy of distribution and reduction processes.
Conclusions

Aspects we hardly talked about:

- User-defined message types
- Matching of messages
- Embrace failure! (Linking and monitoring)
- Combination with other multicore frameworks
Conclusions

Aspects we hardly talked about:

- User-defined message types
- Matching of messages
- Embrace failure! (Linking and monitoring)
- Combination with other multicore frameworks

Remember:

- Cloud Haskell is a library (easy to change, extend, adapt)
- Cloud Haskell is ongoing work
- All of Haskell plus distributed programming
- Watch for exciting new backends and higher-level libraries
Want to try it?

http://haskell-distributed.github.io/

Mini-tutorial blog series by Duncan Coutts and Edsko de Vries:
http://www.well-typed.com/blog/70