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
- ...

Well-Typed
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.
Cloud Haskell Example

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
Haskell
Pure Functions

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

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

data Tree a = Leaf a | Node (Tree a) (Tree a)
size :: Tree a → Int
size (Leaf n) = 1
size (Node l r) = size l + size r
dist :: Floating a ⇒ a → a → a
dist x y = sqrt (x * x + y * y)

data Tree a = Leaf a | Node (Tree a) (Tree a)
size :: Tree a → Int
size (Leaf n) = 1
size (Node l r) = size l + size r

search :: Eq a ⇒ Tree a → a → Bool
search (Leaf n) x = n == x
search (Node l r) x = search l x || search r x
Type signatures

\[
\begin{align*}
\text{dist} & : \text{Floating } a \Rightarrow a \rightarrow a \rightarrow a \\
\text{size} & : \text{Tree } a \rightarrow \text{Int} \\
\text{search} & : \text{Eq } a \Rightarrow \text{Tree } a \rightarrow a \rightarrow \text{Bool}
\end{align*}
\]
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
<table>
<thead>
<tr>
<th>Monad</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maybe a</td>
<td>-- possibly failing</td>
</tr>
<tr>
<td>State s a</td>
<td>-- state-maintaining</td>
</tr>
<tr>
<td>Random a</td>
<td>-- depending on a PRNG</td>
</tr>
<tr>
<td>Signal a</td>
<td>-- time-changing</td>
</tr>
<tr>
<td>Par a</td>
<td>-- annotated for parallelism</td>
</tr>
<tr>
<td>IO a</td>
<td>-- arbitrary side effects</td>
</tr>
<tr>
<td>STM a</td>
<td>-- logged transactions</td>
</tr>
<tr>
<td>Process a</td>
<td>-- Cloud Haskell processes</td>
</tr>
</tbody>
</table>

...
<table>
<thead>
<tr>
<th>Monads</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maybe a</td>
<td>-- possibly failing</td>
</tr>
<tr>
<td>State s a</td>
<td>-- state-maintaining</td>
</tr>
<tr>
<td>Random a</td>
<td>-- depending on a PRNG</td>
</tr>
<tr>
<td>Signal a</td>
<td>-- time-changing</td>
</tr>
<tr>
<td>Par a</td>
<td>-- annotated for parallelism</td>
</tr>
<tr>
<td>IO a</td>
<td>-- arbitrary side effects</td>
</tr>
<tr>
<td>STM a</td>
<td>-- logged transactions</td>
</tr>
<tr>
<td>Process a</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.
Concurrent computation is achieved using the `forkIO` function. This function takes an `IO` action and returns an `IO ThreadId`. Here is an example:

```haskell
forkIO :: IO () → IO ThreadId

-- Create two child threads
printForever msg = forever$ do
  putStrLn msg
  threadDelay (1 * 10^6)

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

The `printForever` function prints a message and delays the execution for a second (in milliseconds) before freezing the thread. The `main` function creates two child threads using `forkIO` and then prints a message from the parent thread.

The code is well-typed, ensuring that all operations are properly defined and that the program behaves as expected.
Concurrency

forkIO :: IO () → IO ThreadId

threadDelay :: Int → IO ()

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

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
Cloud Haskell example revisited

```
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:

<table>
<thead>
<tr>
<th>Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>User application</td>
</tr>
<tr>
<td>Higher-level libraries</td>
</tr>
<tr>
<td>Distributed process core library</td>
</tr>
<tr>
<td>Backend (simplelocalnet, Azure, EC2, …)</td>
</tr>
<tr>
<td>Transport (TCP, in-memory, SSH, ZeroMQ, …)</td>
</tr>
<tr>
<td>System libraries</td>
</tr>
</tbody>
</table>
Nodes, Processes, Communication

Backend responsible for nodes, processes, and communication are backend-agnostic.
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 :: Nodeld \rightarrow Closure (Process (\()) \rightarrow \text{Process ProcessId}}
\]

For the main process:

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

Ad-hoc:

\[
\begin{align*}
\text{send} & : \text{Serializable } a \Rightarrow \text{ProcessId} \rightarrow a \rightarrow \text{Process} () \\
\text{expect} & : \text{Serializable } a \Rightarrow \text{Process } a \\
\text{expectTimeout} & : \text{Serializable } a \Rightarrow \text{Int} \rightarrow \text{Process (Maybe } a) \\
\end{align*}
\]

Sending is asynchronous. Receiving blocks.
Sending and receiving messages

Ad-hoc:

\[
\begin{align*}
\text{send} &: \text{Serializable} \ a \Rightarrow \text{ProcessId} \rightarrow a \rightarrow \text{Process} \ () \\
\text{expect} &: \text{Serializable} \ a \Rightarrow \text{Process} \ a \\
\text{expectTimeout} &: \text{Serializable} \ a \Rightarrow \text{Int} \rightarrow \text{Process} \ (\text{Maybe} \ a)
\end{align*}
\]

Sending is asynchronous. Receiving blocks.

Typed channels:

\[
\begin{align*}
\text{newChan} &: \text{Serializable} \ a \Rightarrow \text{Process} \ (\text{SendPort} \ a, \text{ReceivePort} \ a) \\
\text{sendChan} &: \text{Serializable} \ a \Rightarrow \text{SendPort} \ a \rightarrow a \rightarrow \text{Process} \ () \\
\text{receiveChan} &: \text{Serializable} \ a \Rightarrow \text{ReceivePort} \ a \rightarrow \text{Process} \ a \\
&\ldots
\end{align*}
\]
Serializable a = (Typeable a, Binary a)
Serializable

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.

typeOf :: Typeable a ⇒ a → TypeRep
toDyn :: Typeable a ⇒ a → Dynamic
fromDynamic :: Typeable a ⇒ Dynamic → Maybe a

GHC can “derive” an instance of 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 `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:

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

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

\[
\text{client serverPid} = \\
\quad \text{forever} \; \text{do} \\
\quad \quad \text{send serverPid}() \\
\quad \quad \text{liftIO \$ threadDelay} (1 \times 10^6)
\]
Adapting the client

Old client:

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

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
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`.
Conversations

Some rules about exchanging messages:

▶ only one mailbox per process;
▶ we can \textbf{expect} a particular type;
▶ we can \textbf{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
```

Well-Typed
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
```

Well-Typed
Automatic detection of slaves

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

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

startSlave :: Backend $\rightarrow$ IO () -- does nothing
startMaster :: Backend $\rightarrow$ ([Nodeid] $\rightarrow$ Process ()) $\rightarrow$ IO ()

Master gets node ids of all slaves.
Spawning functions remotely

master :: [Node Id] → Process ()
master slaves = do
    serverPid ← getSelfPid
    forM_ slaves $ λnid → spawn nid ($ (mkClosure ‘client) serverPid)
server
Spawning functions remotely

```markdown
master :: [NodeID] → Process ()
master slaves = do
  serverPid ← getSelfPid
  forM_ slaves $
    \lambda nid → 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

\[
\text{master :: [Input] \rightarrow [NodeId] \rightarrow Process ()} \\
\text{master inputs workers = do} \\
\quad \text{masterPid } \leftarrow \text{getSelfPid} \\
\quad \text{workerPids } \leftarrow \text{forM workers} \\
\quad \quad \lambda \text{nid } \rightarrow \text{spawn nid ($($mkClosure 'worker) masterPid$)} \\
\quad \text{forM_ (zip inputs (cycle workerPids))}$ \\
\quad \quad \lambda (\text{input, workerPid}) \rightarrow \text{send workerPid input} \\
\quad r \leftarrow \text{collectResults (length inputs)} \\
\quad \text{liftIO$ print r}
\]
master :: [Input] → [NodeId] → Process ()
master inputs workers = do
  masterPid ← getSelfPid
  workerPids ← forM workers $
    \lambda nid \rightarrow \text{spawn } nid (\$(\text{mkClosure } \textbf{worker}) \text{ masterPid})$
  forM_ (zip inputs (cycle workerPids)) $
    \lambda (input, workerPid) \rightarrow \text{send } workerPid \text{ input}$
  r ← \textbf{collectResults} (length inputs)
  liftIO $\text{print } r$
... workerPids ← forM workers $ λnid → 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

collectResults :: Int → Process Result
collectResults = go emptyResult

where
  go :: Result → Int → Process Result
  go !acc 0 = return acc
  go !acc n = do
      r ← expect -- obtain one result
      go (combineResults acc r) (n - 1)

In 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