Distributed Programming with

# Cloud Haskell

Andres Löh

14 June 2013, Big Tech Day 6 — 

Well-Typed
The Haskell Consultants

# 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

# Multicore in Haskell

**Many approaches**

Different problems have different requirements / cost models.

# Multicore in Haskell

## Many approaches

Different problems have different requirements / cost models.

### Concurrency

- threads and locks ( MVar s)
- aynchronous computations ( Async s)
- software transactional memory
- …

# Multicore in Haskell

## Many approaches

Different problems have different requirements / cost models.

### (Deterministic) Parallelism

- evaluation strategies
- dataflow-based task parallelism
- flat and nested data parallelism
- …

# Multicore in Haskell

**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.

# Cloud Haskell Example

```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
```

# Haskell

# Pure Functions

$$\mathsf{dist} :: \mathsf{Floating}\ \mathsf{a} \Rightarrow \mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{a}$$
$$\mathsf{dist}\ \mathsf{x}\ \mathsf{y} = \mathsf{sqrt}\ (\mathsf{x} * \mathsf{x} + \mathsf{y} * \mathsf{y})$$

# Pure Functions

```haskell
dist :: Floating a ⇒ a → a → a
dist x y = sqrt (x ∗ x + y ∗ y)
```

```haskell
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
```

# Pure Functions

```haskell
dist :: Floating a => a -> a -> a
dist x y = sqrt (x * x + y * y)
```

```haskell
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
```

```haskell
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

```
dist   :: Floating a ⇒ a → a → a
size   :: Tree a → Int
search :: Eq a ⇒ Tree a → a → Bool
```

# Function calls

```
dist :: Floating a ⇒ a → a → a
```

```
dist x y
dist 2 3
dist (2 + x) (3 + x)
```

Well-Typed

# IO

```haskell
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 ++ "?"
```

# IO

```
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
```

Well-Typed

# 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
...
```

# 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.

# Concurrency

forkIO :: IO () $\rightarrow$ IO ThreadId

# Concurrency

```
forkIO :: IO () → IO ThreadId
```

```
threadDelay :: Int → IO ()
forever     :: Monad m ⇒ m a → m b   -- here:  IO a → IO b
```

Well-Typed

# Concurrency

```haskell
forkIO :: IO () -> IO ThreadId
```

```haskell
threadDelay :: Int -> IO ()
forever     :: Monad m => m a -> m b   -- here:  IO a -> IO b
```

```haskell
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

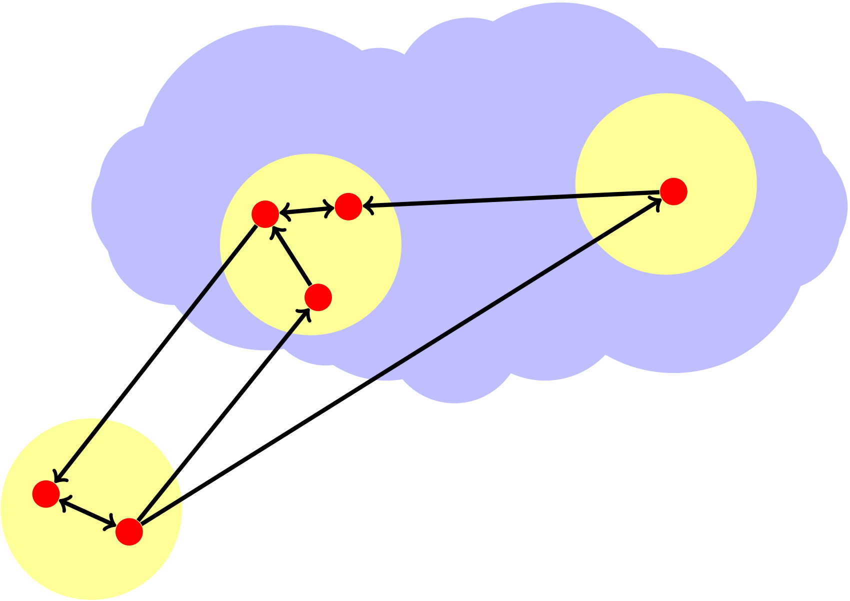
```
server :: Process ()
server = forever $ do
  () ← expect
  liftIO $ putStrLn "ping"

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

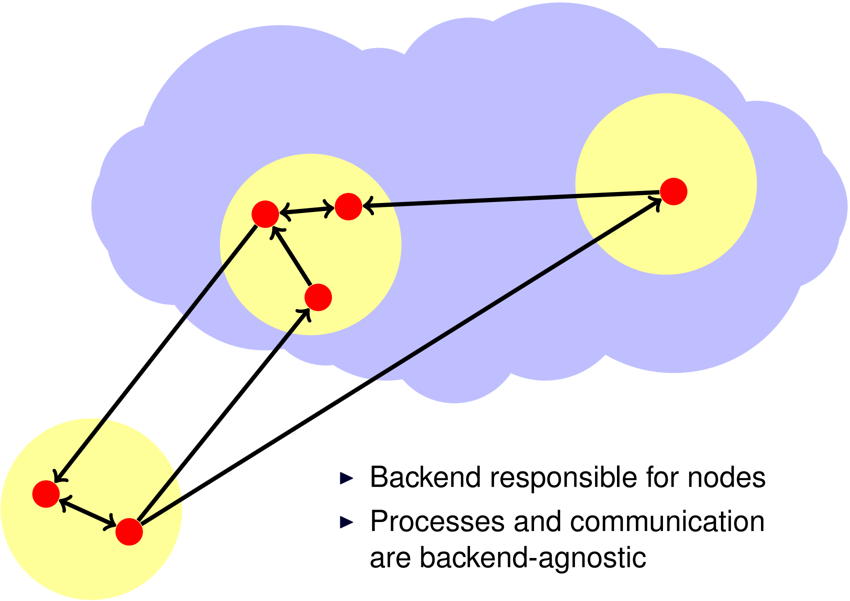
Over-simplified:

| |
|---|
| User application |
| Higher-level libraries |
| Distributed process core library |
| Backend (simplelocalnet, Azure, EC2, ...) |
| Transport (TCP, in-memory, SSH, ZeroMQ, ...) |
| System libraries |

# Nodes, Processes, Communication



# Nodes, Processes, Communication



- Backend responsible for nodes
- Processes and communication are backend-agnostic

# Spawning and running processes

$$
\begin{aligned}
&\mathsf{spawnLocal} :: \mathsf{Process}\ () \rightarrow \mathsf{Process}\ \mathsf{ProcessId} \\
&\mathsf{spawn} \quad\ \ :: \mathsf{NodeId} \rightarrow \mathsf{Closure}\ (\mathsf{Process}\ ()) \\
&\qquad\qquad\qquad\qquad \rightarrow \mathsf{Process}\ \mathsf{ProcessId}
\end{aligned}
$$

For the main process:

$$\mathsf{runProcess} :: \mathsf{LocalNode} \rightarrow \mathsf{Process}\ () \rightarrow \mathsf{IO}\ ()$$

# 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
...
```

# Serializable

$$\text{Serializable a} = (\text{Typeable a}, \text{Binary a})$$

# Serializable

$$\text{Serializable a} = (\text{Typeable a}, \text{Binary a})$$

```
Typeable a      -- has a run-time type representation
Binary a        -- has a binary representation
```

# Static and dynamic typing

## Haskell's typing discipline

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

# Static and dynamic typing

**Haskell's typing discipline**

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

$$
\begin{array}{ll}
\mathsf{typeOf} & :: \mathsf{Typeable}\ a \Rightarrow a \to \mathsf{TypeRep} \\
\mathsf{toDyn} & :: \mathsf{Typeable}\ a \Rightarrow a \to \mathsf{Dynamic} \\
\mathsf{fromDynamic} & :: \mathsf{Typeable}\ a \Rightarrow \mathsf{Dynamic} \to \mathsf{Maybe}\ a
\end{array}
$$

GHC can "derive" an instance of Typeable for any datatype automatically.

# Binary representation

$$\text{encode} :: \text{Binary } a \Rightarrow a \rightarrow \text{ByteString}$$
$$\text{decode} :: \text{Binary } a \Rightarrow \text{ByteString} \rightarrow a$$

- 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

# How to reply

## 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:

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

# Adapting the client

Old client:

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

# Adapting the client

Old client:

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

  forever $ do
    send serverPid ()


    liftIO $ threadDelay (1 ∗ 10^6)
```

# Adapting the client

New client:

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

# 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 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`:

```
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`):

```
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

```
startSlave  :: Backend → IO ()   -- does nothing
startMaster :: Backend → ([NodeId] → Process ()) → IO ()
```

Well-Typed

# Automatic detection of slaves

```
startSlave  :: Backend → IO ()    -- does nothing
startMaster :: Backend → ([NodeId] → Process ()) → IO ()
```

Master gets node ids of all slaves.

# Spawning functions remotely

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

# Spawning functions remotely

$$
\begin{array}{l}
\mathsf{master} :: [\mathsf{NodeId}] \to \mathsf{Process}\ ()\\
\mathsf{master}\ \mathsf{slaves} = \mathbf{do}\\
\quad \mathsf{serverPid} \leftarrow \mathsf{getSelfPid}\\
\quad \mathsf{forM\_}\ \mathsf{slaves}\ \$\\
\qquad \lambda \mathsf{nid} \to \mathsf{spawn}\ \mathsf{nid}\ \boxed{(\$(\mathsf{mkClosure}\ {'}\mathsf{client})\ \mathsf{serverPid})}\\
\quad \mathsf{server}
\end{array}
$$

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 $
    λnid → spawn nid ($(mkClosure 'worker) masterPid)
  forM_ (zip inputs (cycle workerPids)) $
    λ(input, workerPid) → send workerPid input
  r ← collectResults (length inputs)
  liftIO $ 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 $
    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

```haskell
...
  r ← collectResults (length inputs)
  liftIO $ print r
...
```

```haskell
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`