Distributed programming in Haskell

Mathieu Boespflug

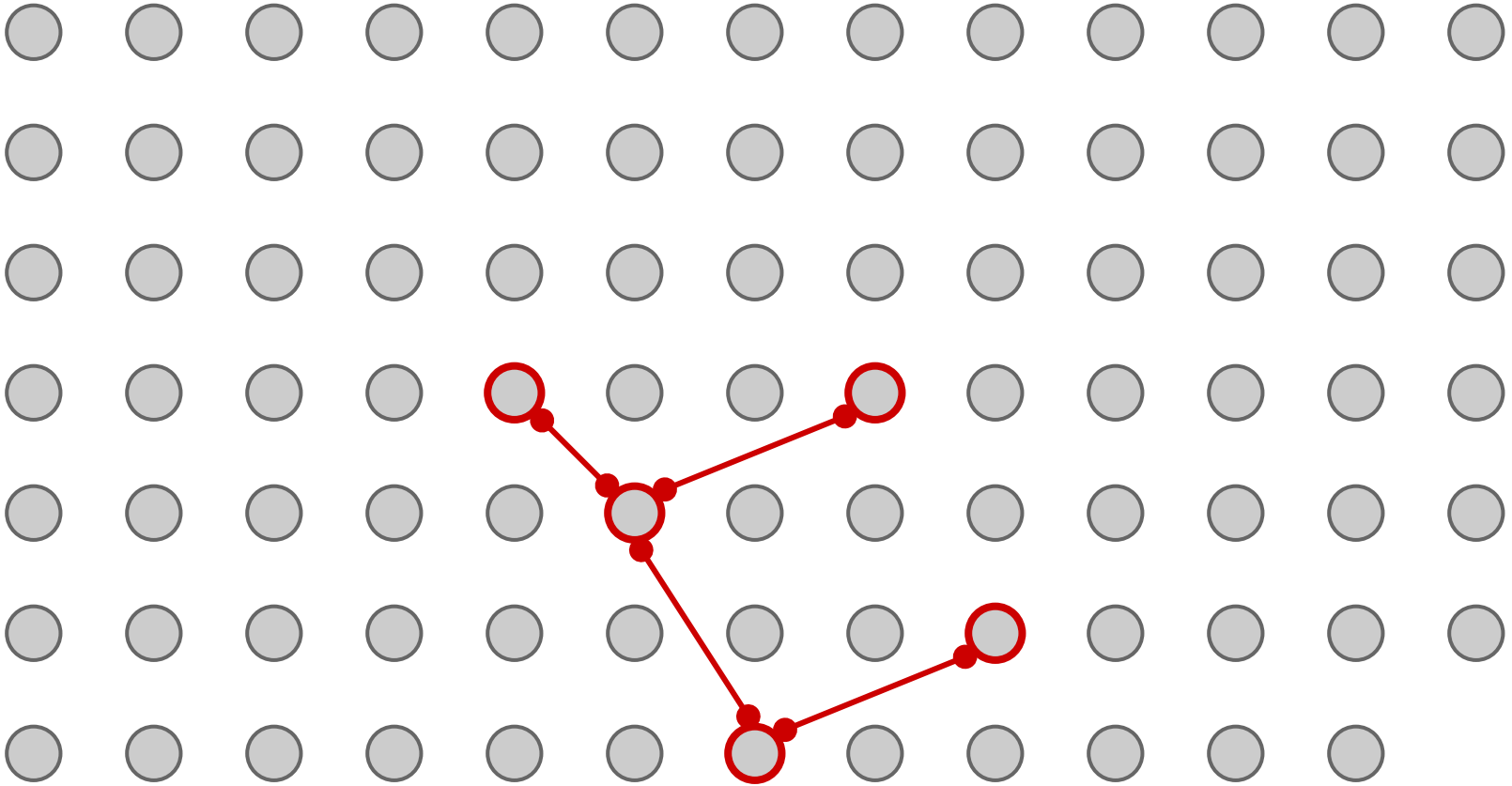
Tweag I/O

Getting started...

```
$ git clone https://github.com/mboes/zurihac15-talk-examples
$ cd zurihac15-talk-examples
$ cabal sandbox init
$ cabal update
$ cabal install
```

Plan

1. Introduction
2. Cloud Haskell from the ground up
3. Code along session
4. Discussion



Distributed programming

- From loosely coupled SOA...
  - REST API, closed set of verbs/resources
  - HTTP request/response
  - Extensible and/or implicit schema (JSON, XML, …)
- ...to tightly coupled distributed systems…
  - RPC, open set of verbs
  - message passing
  - fixed, explicit schemas (strong client/server coupling)
  - … **Focus of this talk**

Why use Haskell?

- ★ (1) Strong library support (warp, http-client, aeson…)
- ★ (2) Strong library support (Cloud Haskell, zmq, …)
- ★ The performance of a mature native compiler
- ★ Typeful programming for robust systems
- ★ Exceptionally powerful array of composition mechanisms for boilerplate-less, high-level programming.

Moving up from the nuts and bolts

- ★ Map/Reduce:
  - map function

$$MR.map :: (k_{in}, v_{in}) \rightarrow [(k_{med}, v_{med})]$$

  - reduce function

$$MR.reduce :: (k_{inter}, [v_{inter}]) \rightarrow [(k_{out}, v_{out})]$$

  - **Reality:**
    - long map/reduce pipelines (Google indexing pipeline: 21 step)
    - 21 steps means 21 mapper classes and 21 reducer classes in Java
- ★ Two directions for improvement:
  - more general data-parallel compute model
  - less boilerplate!

WordCount Hadoop (C++)

```
class WordCountMap: public HadoopPipes::Mapper {
public:
  WordCountMap(HadoopPipes::TaskContext& context){}
  void map(HadoopPipes::MapContext& context) {
    std::vector<std::string> words =
      HadoopUtils::splitString(context.getInputValue(), " ");
    for(unsigned int i=0; i < words.size(); ++i) {
      context.emit(words[i], "1");
    }
  }
};
```

```
class WordCountReduce: public HadoopPipes::Reducer {
public:
  WordCountReduce(HadoopPipes::TaskContext& context){}
  void reduce(HadoopPipes::ReduceContext& context) {
    int sum = 0;
    while (context.nextValue()) {
      sum += HadoopUtils::toInt(context.getInputValue());
    }
    context.emit(context.getInputKey(), HadoopUtils::
toString(sum));
  }
};
```

```
int main(int argc, char *argv[]) {
  return HadoopPipes::runTask(HadoopPipes::
TemplateFactory<WordCountMap,
                                WordCountReduce>());
}
```

WordCount Spark (Java)

```
JavaRDD<String> textFile = spark.textFile("hdfs://...");

JavaRDD<String> words = textFile.flatMap(new FlatMapFunction<String, String>() {
  public Iterable<String> call(String s) { return Arrays.asList(s.split(" ")); }
});
JavaPairRDD<String, Integer> pairs = words.mapToPair(new PairFunction<String, String,
Integer>() {
  public Tuple2<String, Integer> call(String s) { return new Tuple2<String, Integer>(s,
1); }
});
JavaPairRDD<String, Integer> counts = pairs.reduceByKey(new Function2<Integer,
Integer>() {
  public Integer call(Integer a, Integer b) { return a + b; }
});

counts.saveAsTextFile("hdfs://...");
```

Cloud Haskell

Untyped messages

- ★ BSD sockets: low-level streaming API (+ limited datagram oriented API).
- ★ Need higher-level notion of a *message.*
- ★ A message can be arbitrary length.
- ★ For performance and simplicity, multiple segments:
- ★ type Message = [ByteString]

network-transport

- ★ A general purpose networking library for *many-process* communication.
- ★ Provides message abstraction.
- ★ Each node has one or more *endpoints*.
- ★ Each process wants *ordered* communication to any other process on any other node.
- ★ Have each process create a connection (or *open a session*) to every other process to encapsulate connection state.
- ★ Under the hood, *multiplex* process-to-process connections into a single transport-level connection.

network-transport (Idealized API)

```
newEndPoint :: Transport -> IO EndPoint

address     :: EndPoint -> EndPointAddress

connect     :: EndPointAddress
            -> IO Connection

receive     :: EndPoint -> IO Event
```

Processes

```
data Process a
instance Monad Process
instance MonadIO Process

data ProcessId      -- like ThreadId

-- like forkIO
spawnLocal :: Process a -> Process ProcessId
```

Sending/Receiving messages

```
send :: Serializable a
     => ProcessId
     -> a
     -> Process ()

expect :: Serializable a
       => Process a
```

Example

```
data Arith = Plus Double Double
           | Mult Double Double
           | Neg Double

instance Serializable ArithOp
```

```
server :: Process ()
server = forever $ do
  expect >>= say . show . \case of
    Plus x y -> x + y
    Mult x y -> x * y
    Neg x -> negate x
```

```
let'sDoSomeMath :: ProcessId -> Process ()
let'sDoSomeMath there = do
    send there $ Plus 10 2
    send there $ Mult (2^10) (3^10)
    send there $ Neg 1
```

`Arith` type is …

★ … indirect;
★ … error prone;
★ … difficult to extend;
★ … antimodular.

Example

```
plus, mult :: Int -> Int -> Process ()
plus x y = say $ show $ x + y
mult x y = say $ show $ x * y

neg :: Int -> Process ()
neg x = say $ show $ negate x
```

```
let'sDoSomeMath :: NodeId -> Process ()
let'sDoSomeMath there = do
  spawnLocal $ plus 10 2
  spawnLocal $ mult (2^10) (3^10)
  spawnLocal $ neg 1
```

Remote process

- ★ Message of type `Process a` arbitrary action.
- ★ Not directly serializable (w/o heavyweight runtime support).
  - ○ How to spawn processes remotely (not just locally)?
- ★ **Solution:** track in the type system which actions are serializable!

```
data Closure thing
spawn :: Closure (Process a)
      -> Process ProcessId
```

Applicative forms

`(f exp`$_1$ `... exp`$_n$`)`

In the Identity applicative functor:

```
newtype Id a = Id a

pure  :: a -> Id a
(<*>) :: Id (a -> b) -> Id a -> Id b

pure f <*> pure exp1 <*> ... <*> pure expn
```

(Serializable?) (quasi-)Applicatives

$(f\ exp_1\ \ldots\ exp_n)$

```
newtype Closure a = Closure a

pure  :: a -> Closure a
(<*>) :: Closure (a -> b) -> Closure a -> Closure
b
```

(Serializable?) (quasi-)Applicatives

$(f\ exp_1\ \ldots\ exp_n)$

```
newtype Closure a = Closure a

pure  :: Serializable a => a -> Closure a
(<*>) :: Closure (a -> b) -> Closure a -> Closure
b
```

Serializable closures

```
cpure :: Serializable a => a -> Closure a
cap   :: Closure (a -> b)
      -> Closure a
      -> Closure b

-- A portable code pointer. Like a “name” for expressions.
data StaticPointer a

closure :: StaticPointer a -> Closure a
```

-XStaticPointers extension

```
plus, mult :: Int -> Int -> Process ()
plus x y = say $ show $ x + y
mult x y = say $ show $ x * y

plusPtr, mulPtr :: StaticPointer (Int -> Int -> Process ())

plusPtr = static ptr

mulPtr = static ptr
```

Rule: can only `static e` if `e` is closed (no free variables).

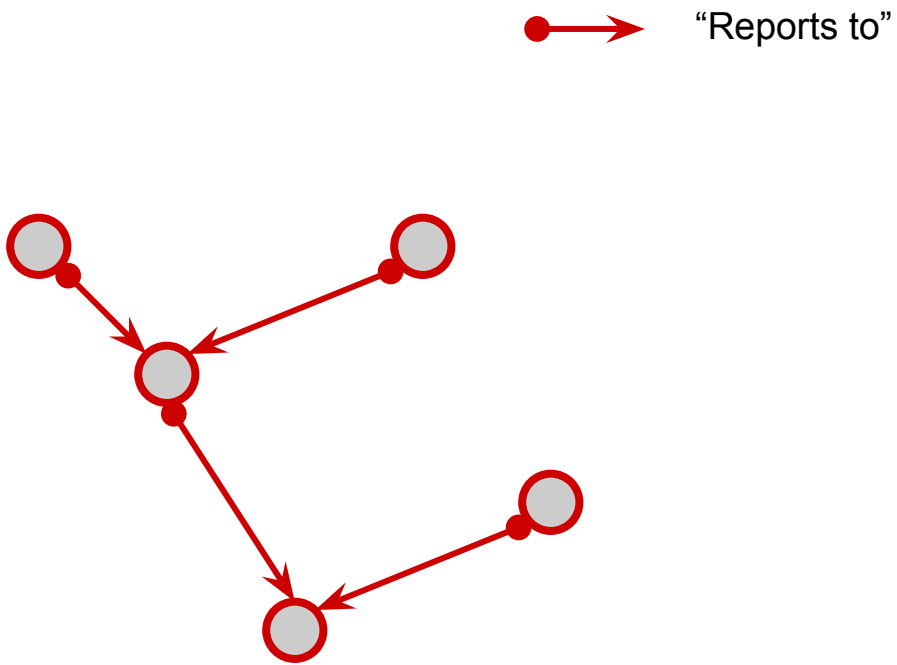
Example

```
plus, mult :: Int -> Int -> Process ()
plus x y = say $ show $ x + y
mult x y = say $ show $ x * y

neg :: Int -> Process ()
neg x = say $ show $ negate x
```

```
let'sDoSomeMath :: NodeId -> Process ()
let'sDoSomeMath there = do
  spawn there $ closure $ static (plus 10 2)
  spawn there $ closure $ static (mult (2^10) (3^10))
  spawn there $ closure $ static (neg 1)
```

Supervision hierarchies



Example

```
plus, mult :: Int -> Int -> Process ()
plus x y = say $ show $ x + y
mult x y = say $ show $ x * y

neg :: Int -> Process ()
neg x = say $ show $ negate x
```

```
link :: ProcessId
     -> Process ()
```

```
let'sDoSomeMath :: NodeId -> Process ()
let'sDoSomeMath there = do
  pid <- spawn there $ closure $ static (plus 10 2)
  link pid
  spawnLink there $ closure $ static (mult (2^10) (3^10))
```

Let’s write some code!

Discussion

Discussion

- ★ Distributed programming on the cheap.
  - Captured a useful notion of remote code execution.
  - Minimal runtime support required.
  - How useful is it?
- ★ No good story yet for heterogeneous clusters.
- ★ Cloud Haskell provides low-level mechanism.
  - Build data-parallel models on top
  - Map/Reduce, distributed NDP, DAG of transformers (Spark)

Thank you!