Towards Haskell in the Cloud

Jeff Epstein
University of Cambridge
jee36@cam.ac.uk

Andrew P. Black
Portland State University *
black@cs.pdx.edu

Simon Peyton-Jones
Microsoft Research, Cambridge
simonpj@microsoft.com

Abstract

We present Cloud Haskell, a domain-specific language for developing programs for a distributed computing environment. Implemented as a shallow embedding in Haskell, it provides a message-passing communication model, inspired by Erlang, without introducing incompatibility with Haskell's established shared-memory concurrency. A key contribution is a method for serializing function closures for transmission across the network. Cloud Haskell has been implemented; we present example code and some preliminary performance measurements.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed Programming

General Terms Languages, Reliability, Performance

Keywords Haskell, Erlang, message-passing

1. Introduction

Cloud Haskell is a domain-specific language for cloud computing, implemented as a shallow embedding in Haskell. It presents the programmer with a computational model strongly based on the message-passing model of Erlang, but with additional advantages

that stem from Haskell's purity, types, and monads.

The message-passing model, popularized by Erlang [1] for highly-reliable real-time applications and by MPI [6] for high-performance computing, stipulates that concurrent processes have no access to each other's data: any data that needs to be communicated from one process to another are sent explicitly in messages. We choose this model because it makes the costs of communication apparent, and because it makes a concurrent process a natural unit of failure: since processes do not share data, the data of one process cannot be contaminated by a fault in another.

We use the term "cloud" to mean a large number of processors with separate memories that are connected by a network and have independent failure modes. We don't believe that shared-memory concurrency is appropriate for programming the cloud. An effective programming model must be accompanied by a cost model. In a distributed memory system, the most significant cost, in both energy and time, is data movement. A programmer trying to reduce

---

* Research conducted while on sabbatical at Microsoft Research, Cambridge.

Haskell'11, September 22, 2011, Tokyo, Japan.

these costs needs a model in which they are explicit, not one that denies that data movement is even taking place — which is exactly the premise of a simulated shared memory.

One reason that this model of distributed computing has not previously been brought to Haskell is that it requires a way of running code on a remote system. Our work provides this, in the form of a novel method for serializing function closures. Without extending the compiler, our method hides the underlying serialization mechanism from the programmer, but makes serialization itself explicit.

In many ways, failure is *the* defining issue of distributed computation. In a network of hundreds of computers, some of them are likely to fail during the course of an extended computation; if our only recourse were to restart the computation from the beginning, the likelihood of it ever completing would become ever smaller as the system scales up. A programming system for the cloud must therefore be able to tolerate partial failure. Here again, Erlang has a solution that has stood the test of time; we don't innovate in this area, but adopt Erlang's solution (summarized in Section 2.5).

If Erlang has been so successful, you may wonder what Haskell brings to the table. The short answer is: purity, types, and monads. As a pure functional language, data is by default immutable, so the lack of shared, mutable data won't be missed. Importantly, immutability allows the implementation to decide whether to share or copy the data: the choice is semantically invisible, and so can depend on the locations of the processes. Moreover, pure functions are idempotent; this means that functions running on failing hardware can be restarted elsewhere without the need for distributed transactions or other mechanisms for "undoing" effects. Types in general, and monadic types in particular, help to guarantee prop-

erties of programs statically. For example, a function that has an externally-visible effect such as sending or receiving a message *cannot* have the same type as one that is pure. Monadic types make it convenient to program in an effectful style when that is appropriate, while ensuring that the programmer cannot accidentally mix up the pure and effectful code.

The contributions of this paper are as follows.

- A description of Cloud Haskell's interface functions (Sections 2 and 3). Following Erlang, our language provides a system for exchanging messages between lightweight concurrent processes, regardless of whether they are running on one computer or on many. We also provide functions for starting new remote processes, and for fault tolerance, which closely follow Erlang. However, unlike Erlang, Cloud Haskell also allows shared-memory concurrency *within* one of its processes.
- An additional message-passing interface that uses multiple *typed channels* in place of Erlang's single untyped channel (Section 4). Each channel is realized as a pair of ports; while the send port can be transmitted over the network, the receive port cannot.
- A method for serializing function closures that enables higher-order functions to be used in a distributed environment (Section 5). Starting a remote process means sending a representation of a function and its environment across the network; our approach makes the environment *explicit*, and thus gives the programmer control over the cost of the message.

- A demonstration of the effectiveness of our approach in the form of an implementation (discussed in Sections 6 and 7), and a complete example application (Section 8). We also provide performance measurements from the k-means clustering algorithm, an iterative algorithm for partitioning data points into natural groups (Section 9).

2. Processes and messages

We start with an overview of the basic elements of Cloud Haskell: processes, messages, what can be sent in a message and provision for failure. All of the elements of our DSL are listed in Figure 2.

2.1 Processes

The basic unit of concurrency in Cloud Haskell is the *process*: a concurrent activity that has been "blessed" with the ability to send and receive messages. As in Erlang, processes are lightweight, with low creation and scheduling overhead. Processes are identified by a unique process identifier, which can be used to send messages to the new process.

In most respects, Cloud Haskell follows Erlang by favoring message-passing as the primary means of communication between processes. Our language differs from Erlang, though, in that it also supports shared-memory concurrency within a single process. The existing elements of Concurrent Haskell, such as `MVar` for shared mutable variables and `forkIO` for creating lightweight threads, are still available to programmers who wish to combine message passing with the more traditional approach. This is illustrated in Figure 1. Our embedding ensures that mechanisms specific to shared-

memory concurrency cannot be inadvertently used between remote systems. The key idea that makes this separation possible is that not all data types can be sent in a message; in particular, MVars and ThreadIds are not Serializable (see Section 2.3).

2.2 Messages to processes

Any process can send and receive messages. Our messages are asynchronous, reliable, and buffered. All the state associated with messaging (e.g., the message buffer) is wrapped in the ProcessM monad, which is updated with each messaging action. Thus, any code participating in messaging must be in the ProcessM monad.

Cloud Haskell provides two styles of message passing: *untyped messages*, which closely resemble messages in Erlang; and *typed channels*, which leverage Haskell's strong type system to provide static guarantees about the content of messages. In this section, we discuss untyped messages; we describe channels in Section 4.

The primitives for untyped messaging are send and expect[1]:

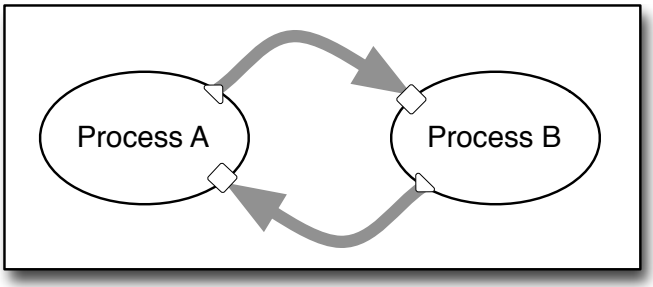
```
send :: Serializable a ⇒ ProcessId → a → ProcessM ()
expect :: Serializable a ⇒ ProcessM a
```

Before we discuss these primitives in detail, let's look at an example of their use. ping is a process that accepts "pong" messages and responds by sending a "ping" to whatever process sent the pong. The data types are:

```
data Ping = Ping ProcessId
data Pong = Pong ProcessId
-- omitted: Serializable instance for Ping and Pong
```

[1] Why expect rather than receive ? The intent is that expect is used when

the program *expects* a message of a particular type. There are more general `receive` functions that allow for type-dependent and conditional receives; these are discussed in Section 3.



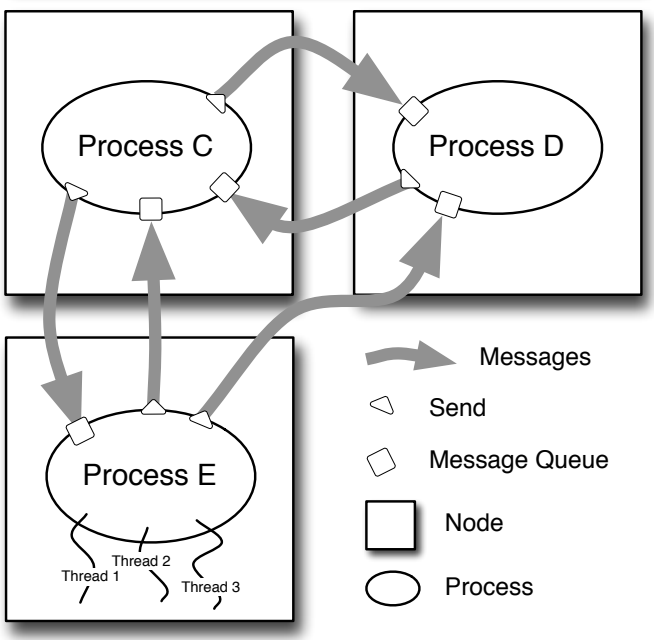


Figure 1. Processes A and B do not share memory, even though they are running on the same physical processor; instead they communicate by sending messages, shown by grey arrows. This makes it easy to reconfigure the application to resemble the situation shown with processes C and D. Process E has created some lightweight threads using Concurrent Haskell's `forkIO` primitive; these threads share memory with Process E, and with each other. However, they cannot send or receive messages from Process E's

channels, because this requires execution in the ProcessM monad; threads, in contrast, execute in the IO monad.

Using send and expect, the code for such a process would be:

```
ping :: ProcessM ()
ping = do Pong partner ← expect
          self ← getSelfPid
          send partner (Ping self)
          ping
```

The equivalent code in Erlang looks like this:

```
ping() → receive
           {pong, Partner} →
             Partner ! {ping, self()}
         end,
         ping().
```

These two programs have similar structure. Both ping functions are designed to be run as processes. They each wait for a specific message to be received; the Haskell expect function matches incoming messages by type, whereas in Erlang, messages are usually pattern-matched against tuples whose first element is a well-known atom. The programs wait for a "pong" message, and ignore all others. The "pong" message contains in its payload the process ID of a "partner", to whom the response message is sent; this message contains the process ID of the ping process (self). Finally, they wait for the next message by repeating with tail recursion.

Note that in the Erlang version, Ping and Pong are atoms, whereas in the Haskell version they are types, and so need to be

declared explicitly. As given, the type declarations are incomplete; Ping and Pong need to be declared to be instances of the class Serializable ; we will discuss this in Section 2.3.

The send function is our general-purpose message-sending primitive; it packages up a chunk of *serializable* data of arbitrary type as a bag of bits, together with a representation of that type (a TypeRep), and transmits both (possibly over the network) to a particular process, selected by the ProcessId argument. Upon receipt, the incoming message will be placed in a message queue associated with the destination process. The send function corresponds to Erlang's ! operator.

At the far end of the channel, the simplest way of receiving a message is with expect, which examines the message queue associated with the current process and extracts the first message whose type matches the (inferred) type of expect — a Ping message in the example. The implementation of expect looks down the queue for a message with the right type representation, dequeues that message, parses the bag of bits into a data item of the right type, and returns it. If there is no message of the appropriate type on the queue, expect waits for one to arrive.

2.3 Serialization

When we said that the data to be transmitted must be serializable, we meant that each item must implement the Serializable type class. This ensures two properties: that it is Binary and that it is Typeable (see Figure 2: Type classes). Binary means that put and get functions are available to encode and decode the data item into binary form and back again; Typeable means that a function typeOf can be used to produce a TypeRep that captures the item's type.

While all of Haskell's primitive data types and most of the common higher-level data structures are Serializable, and can therefore be part of a message, some data types are emphatically *not* serializable. One example is MVar, the type of Haskell's mutable concurrent variables. Since MVars allows communication between threads on the assumption of shared memory, it isn't helpful to send an MVar to a remote process that may not share memory with the current process. Although one can imagine a synchronous distributed variable that mimics the semantics of an MVar, such a variable would have a vastly different cost model from a normal MVar. Since neither MVar's cost model nor its implementation could be preserved in an environment that required communication between remote systems, we prohibit programmers from using MVars in that way. Notice, however, that we do not attempt to stop the programmer from using MVars within a single process: processes are allowed to use Haskell's forkIO function to create *local* threads that can share memory using MVars. The same is true for TVars: the fact that they are non-serializable guarantees that STM transactions do not span processes, but the programmer is free to use STM *within* a process. In fact, our implementation uses STM to protect the message queue, as discussed in Section 7.

2.4 Starting and Locating Processes

To start a new process in a distributed system, we need a way of specifying where that process will run. The question of *where* is answered with Cloud Haskell's unit of location, the node. A node can be thought of as an independent address space. Each node is named by a NodeId, a unique identifier that contains an IP address that can be used to communicate with the node. So, to be able to

start a process, we want a function named spawn that takes two parameters: a NodeId that specifies where the new process should run, and some expression of what code should be run there. Since we want to run code that is able to receive messages, the code should be in the ProcessM monad. The spawn function should then

Basic messaging

```
instance Monad ProcessM
instance MonadIO ProcessM
send   :: Serializable a ⇒ ProcessId → a → ProcessM ()
expect :: Serializable a ⇒ ProcessM a
```

Channels

```
newChan :: Serializable a
        ⇒ ProcessM (SendPort a, ReceivePort a)
sendChan :: Serializable a
         ⇒ SendPort a → a → ProcessM ()
receiveChan :: Serializable a ⇒ ReceivePort a → ProcessM a
mergePortsBiased :: Serializable a ⇒ [ReceivePort a]
                 → ProcessM (ReceivePort a)
mergePortsRR:: Serializable a ⇒ [ReceivePort a]
             → ProcessM (ReceivePort a)
```

Advanced messaging

```
instance Monad MatchM
receiveWait    :: [MatchM q ()] →ProcessM q
receiveTimeout :: Int → [MatchM q ()]
               → ProcessM (Maybe q)
match :: Serializable a ⇒ (a → ProcessM q) →MatchM q ()
matchIf :: Serializable a ⇒ (a → Bool)
        → (a → ProcessM q) →MatchM q ()
matchUnknown:: ProcessM q →MatchM q ()
```

Process management

```
spawn :: NodeId → Closure (ProcessM ()) 
      → ProcessM ProcessId
call :: Serializable a ⇒ NodeId →
        Closure (ProcessM a) → ProcessM a
terminate :: ProcessM a
getSelfPid :: ProcessM ProcessId
getSelfNode :: ProcessM NodeId
```

Process monitoring

```
linkProcess :: ProcessId → ProcessM ()
monitorProcess :: ProcessId → ProcessId
                → MonitorAction → ProcessM ()
```

Initialization

```
type RemoteTable = [(String,Dynamic)]
runRemote      :: Maybe FilePath → [RemoteTable]
               → (String → ProcessM ()) → IO ()

type PeerInfo = Map String [NodeId]
getPeers       :: ProcessM PeerInfo
findPeerByRole :: PeerInfo → String → [NodeId]
```

Syntactic sugar

```
mkClosure :: Name → Q Exp
remotable :: [Name] →Q [Dec]
```

Logging

```
say :: String → ProcessM ()
```

Type classes

```
class (Binary a, Typeable a) ⇒ Serializable a
```

```
class Typeable a where typeOf :: a → TypeRep
class Binary t where {put :: t → PutM (); get :: Get t}
encode :: Binary a ⇒ a → ByteString
  -- Defined in terms of put
decode :: Binary a ⇒ ByteString → a
  -- Defined in terms of get
```

Figure 2. The interface functions of Cloud Haskell.

return a ProcessId, which can be used with send. Since spawn will itself depend on messaging, it, too, will be in the ProcessM monad. So, its type will be something like:

```
-- wrong
spawn :: NodeId → ProcessM () → ProcessM ProcessId
```

In combination with the ping and pong functions, spawn could be used like this:

```
-- wrong
do pingProc ← spawn someNode ping
   pongProc ←spawn otherNode pong
   send pingProc (Pong pongProc)
```

This code is intended to start two new processes, located on someNode and otherNode, with each process expecting to receive messages of a particular type. To begin the exchange, we send an initial Pong message to the ping process.

In Cloud Haskell, the actual type of spawn is

```
spawn :: NodeId → Closure (ProcessM ())
```

```
    → ProcessM ProcessId
```

The difference between this and our initial guess is that the second argument to `spawn` is `Closure (ProcessM ())` rather than just `ProcessM ()`. Serializing a function means serializing two things: a representation of its code, and a representation of its environment — more precisely, the bindings of its free names. A `Closure` is exactly this, but the details of closures turn out to be surprisingly tricky. We discuss this at length in Section 5.

2.5 Fault Tolerance

Fault tolerance in Cloud Haskell is based on ideas from Erlang. The premise of Erlang-style fault tolerance is that, when something goes deeply wrong inside a process, the best course of action is for the process to terminate; attempting to glue the pieces back together with string and sealing wax will not lead to robustness. Instead, another process should take over. To make this possible, one process can monitor another process; if the monitored process terminates, the monitoring process will be notified. Ascertaining the origin of the failure and recovering from it are left to the application or a higher-level framework.

A process can request to be notified in the event that another process terminates for any reason, or only in the case of a fault involving the terminating process. Examples of such faults are uncaught exceptions, and the node on which the process is running becoming inaccessible. Notifications can be delivered in two ways: as asynchronous exceptions, which can be caught with Haskell's usual exception handling mechanisms; or by message, which can be received like any other message. The choice is made by the monitoring process when monitoring is established.

The functions for setting-up process monitoring are:

```
monitorProcess :: ProcessId → ProcessId
                  → MonitorAction → ProcessM ()
linkProcess    :: ProcessId → ProcessM ()
```

monitorProcess a b ma establishes unidirectional process monitoring. That is, process a will be notified if process b terminates. The third argument determines whether the monitoring process a will be notified by exception or by message.

linkProcess corresponds to Erlang's link. It establishes bidirectional process monitoring between the current process and a given process; if either of those processes terminate abnormally, the other will receive an asynchronous exception. linkProcess is defined in terms of monitorProcess.

3. Matching Messages

In Section 2.2, we introduced the expect function, which lets us receive messages of a particular type. But what if our process wants to be able to accept messages of multiple types? Ideally, we'd like to be able to approximate the Erlang receive syntax:

```
1  math() →
2      receive
3          {add, Pid, Num1, Num2} →
4              Pid ! Num1 +Num2;
5          {divide, Pid, Num1, Num2} when Num2 ≠0 →
6              Pid ! Num1 /Num2;
7          {divide, Pid, _, _} →
8              Pid ! div_by_zero
9      end,
10     math().
```

This code will accept and respond to several different messages. It does this by pattern matching on the values of the messages, which are (by convention) all tuples. Each pattern has a corresponding message-handling action. Line 3 attempts to match a tuple containing the atom `add`, a process ID, and two numbers; when it finds a message of that form, it will invoke the corresponding handler on line 4, which sends back the sum of the two numbers. In the next pattern, on line 5, the code responds to a message with a `divide` atom, but only if the divisor is not zero; this is controlled by the `when` syntax. Finally, the third pattern (line 7) will send back a `div_by_zero` atom in response to all messages not matched by the previous patterns, which is to say, those cases when the divisor is zero. Patterns are tested against each message in the order that they appear, so that the last pattern will be reached only if the first two fail to match. If no pattern matches, the unmatched message is left in the message queue, and the whole receive statement is repeated for the next message. This goes on until a match is found or the queue is exhausted. How can we provide similar functionality in Haskell?

3.1 Receiving and Matching

Haskell doesn't have an atom data type, but an idiomatic way of representing the different messages is to use type constructors. Imagine that a process should be able to perform mathematical operations remotely, and should be able to respond to two requests: `Add pid a b`, and `Divide pid num den`. The response should be to send back either a value `Answer Double`, or a `DivByZero` error message. It is certainly possible to create a message type that represents the sum of all of these variants, suitable for use with `expect`:

```
data MathOp = Add ProcessId Double Double
            | Divide ProcessId Double Double
            | Answer Double
            | DivByZero
```

There are several problems with using a sum type like MathOp. Remember that expect can select messages only by type. Thus, a process receiving MathOps messages would need to be able to respond to all variants, even those that don't make sense: presumably, only client processes should receive Answer and DivByZero while only the calculating process should receive Add and Divide. Moreover, putting all messages into a single sum type breaks modularity by exposing details of the calculating process to the client; when we add a new mathematical operation, every client will need to update its code, even if it doesn't use that operation. Worse, a single process that offers more than one service could not keep them separate; clients of either would be forced to see the interface of both.

To avoid this kind of false dependency between what ought to be independent modules, it is better to break the single MathOp type into several types, each of which can be handled separately. We can imagine three types of messages:

```
data Add = Add ProcessId Double Double
data Divide = Divide ProcessId Double Double
data DivByZero = DivByZero
```

In addition to the above types, the answer can be sent to the client simply as a message of type Double — no wrapper required. But now we have a different quandary: although expect can receive any type of message, it can receive only one type of message at a time, and will block until a message of that type is put into the message

queue. So there's no way to handle the above message types if we don't know the order in which the client is going to send Adds and Divides. What we need is an alternative to expect that provides the notion of *choice* — something like the multi-way choice of Erlang's receive syntax.

How might we embed multi-way choice into Haskell? First, let's consider how to specify a pair consisting of a message type and its corresponding action. We introduce match, which accepts a message handler function as a parameter.

```
match :: Serializable a ⇒ (a → ProcessM q) → MatchM q ()
```

When match tests an incoming message, it compares the type of the message against the type a, the parameter of the message handler. Any message of that type will be considered "matched": it will be removed from the message queue, and given as an argument to the message handler function.

match is in the MatchM monad, which is responsible for providing match with the current state of the message queue, and then providing match's caller with the result of its test. match will typically be used with receiveWait, which mimics Erlang's receive syntax by evaluating a list of MatchMs in order. The value returned by the selected message action is also the return value of receiveWait.

```
receiveWait :: [MatchM q ()] →ProcessM q
```

How can we mimic Erlang's when clause, which allows message acceptance to be qualified by a predicate? We do this with matchIf, whose first parameter is a predicate that is allowed to examine the incoming message without removing it from the queue. Because this predicate is pure, it cannot alter the state of the message queue.

```
matchIf :: Serializable a ⇒ (a → Bool) →
                (a → ProcessM q) → MatchM q ()
```

Now let's use these tools to implement the math function in Haskell. Notice that Erlang's patterns (lines 3, 5 and 7 of the code on page 4) have been replaced by lambda functions.

```
-- omitted: Serializable instances for Add, Divide, and
     DivByZero types
math :: ProcessM ()
math =
  receiveWait
    [ match   (λ(Add pid num1 num2) →
                send pid (num1 +num2)),
      matchIf (λ(Divide _ _ num2) →num2 ≠0)
              (λ(Divide pid num1 num2) →
                send pid (num1 /num2)),
      match   (λ(Divide pid _ _) →
                send pid DivByZero) ]
        ≫ math
```

The combination of receiveWait and match closely corresponds to Erlang's receive syntax. The MatchMs are tested in order against each message in the message queue. When a matching message is found, the corresponding lambda function is invoked.

Notice that matching by message type is not quite the same as matching by message value. For example, if a particular match accepts messages of a certain type, then all variants of that type must be handled. In the above example, this is okay, because the Add type has only a single variant, built with the Add constructor. If instead it were a sum type with a second constructor, then the match call that deals only with the Add constructor would raise

a pattern match exception if a message with the other constructor were received.

Clearly, receiveWait is more flexible than expect. In fact, expect is implemented in terms of receiveWait. Its definition is:

```
expect :: Serializable a ⇒ ProcessM a
expect = receiveWait [match return]
```

3.2 Matching without Blocking

Whether we use receiveWait or expect, we run the risk that the function will block until a certain type of message arrives. What if we want to check the incoming message queue, but not wait indefinitely? Erlang lets us do this with the receive ... after syntax:

```
Pid ! {query, Stuff},
receive
  {response, Answer} →
    show_answer(Answer)
after
  50000 →
    show_error("Timeout!")
end
```

This code will wait at most 50 seconds to receive a response to its query; after 50 seconds, it will stop waiting and display an error message. The corresponding function in Cloud Haskell is receiveTimeout, which is very similar to receiveWait. Like receiveWait, it takes a list of matches, but it also takes a timeout value. If the timeout is exceeded, the function returns Nothing.

```
receiveTimeout :: Int → [MatchM q ()] →
                          ProcessM (Maybe q)
```

Thus we can translate the Erlang example into Haskell:

```
do send pid (Query stuff)
   ret ← receiveTimeout 50000
     [ match(λ(Response answer) →
             return answer) ]
   case ret of
     Nothing → showError "Timeout!"
     Just ans → showAnswer ans
```

As with Erlang's `receive ... after` syntax, `receiveTimeout` can be called with a timeout value of zero, which has the effect of checking for a matching message and returning immediately if no match is found.

4. Messages through channels

In the previous sections, we've shown how a message can be sent to a process. As you can see from the type of `send`, any serializable data structure can be sent as a message to any process. Whether or not a particular message will be accepted (i.e., dequeued and acted upon) by the recipient process isn't determined until runtime. But what about Haskell's strong typing? Wouldn't it be nice to have some static guarantees that messages are sent to receivers who know how to deal with them?

To offer this assurance, we provide distributed *typed channels* as an alternative to sending messages directly to a process. Each channel consists of two ends, which we call the *send port* and the *receive port*. Messages are inserted via the send port, and extracted in FIFO order from the receive port. Unlike process identifiers, channels are created with a specific type: the send port will accept

messages only of that type, and consequently the receive port will proffer messages only of that type.

The central functions of the channel interface are:

```
newChan :: Serializable a
        ⇒ ProcessM (SendPort a, ReceivePort a)
sendChan :: Serializable a ⇒ SendPort a → a → ProcessM ()
receiveChan :: Serializable a ⇒ ReceivePort a → ProcessM a
```

A critical point is that although a SendPort can be serialized and copied to other nodes, allowing the channel to accept data from multiple sources, a ReceivePort is not serializable, and thus cannot be moved from the node on which it was created. This restriction is enforced by making SendPort an instance of Serializable , but not ReceivePort. This is a deliberate design decision; as a consequence, receive ports in Cloud Haskell are more like Erlang process identifiers or TCP ports than they are like receive rights in the Accent [13] and Mach [5, §4.2.3] operating system kernels. This decision reflects our intended execution environment, in which we expect channels to be used to communicate across a network. The ability, offered by Mach, to move a receive port from one place to another can be very useful for moving responsibility for a service from one provider to another, but only if this can be done without the risk of losing messages that are in transit. Although easy in an operating system like Mach, it is much less so in a distributed environment, especially when one of the prime reasons for moving a service to a new server is that the old server has crashed.

We can now reformulate our ping example to use typed channels. The ping process is given two ports: a receive port on which to receive pongs, and a send port on which to emit pings. Each message now contains the send port on which its recipient should

respond; thus the Ping message contains the send port of a channel of pongs, and vice-versa.

```
ping2 :: SendPort Ping → ReceivePort Pong → ProcessM ()
ping2 pingout pongin =
    do (Pong partner) ← receiveChan pongin
       sendChan partner (Ping pongin)
       ping2 pingout pongin
```

4.1 Combining ports

There is an analogy between the expect function and its channel-based counterpart, receiveChan: both receive a message of a particular type. The channel-based counterpart of receiveWait is the mergePorts family of functions, which let us receive a message from one of several channels.

```
mergePortsBiased :: Serializable a ⇒ [ReceivePort a] →
      ProcessM (ReceivePort a)
mergePortsRR:: Serializable a ⇒ [ReceivePort a] →
      ProcessM (ReceivePort a)
```

Given a list of ReceivePorts of the same message type, these functions will return a new ReceivePort that, when read with receiveChan, will provide a message taken from one of the input ReceivePorts. You can visualize that the mergePorts functions take several "feeder" ports and squeeze them together into a "merged" port. Even after producing the merged ReceivePort, the original ports may continue to be used independently. Messages read from the merged port are extracted from the queue of the feeder port,

and so it is impossible to receive the same message twice.

mergePortsBiased and mergePortsRR differ in the order that the input ports are queried, which is significant in the case that more than one port has a message waiting. Each subsequent read from the port created by mergePortsBiased will query the feeder ports in the order that they were provided to mergePortsBiased — so the port is "biased" towards the first feeder port. So, if the first feeder port always has a message waiting, it could starve the other ports.

If this biased behavior is undesirable, use mergePortsRR instead. The port created by mergePortsRR will rotate the order in which the feeder ports are queried with each subsequent read. The effect is a fairer "round-robin" multiplexing, which will guarantee that, given enough reads on the merged port, every feeder port will eventually have a chance to contribute a message from its queue.

Cloud Haskell also provides a family of combinePorts functions that can be used to combine ports of different types, but we do not discuss them in this paper.

5. Closures

As we hinted in Section 2.4, the question of how to transmit function closures from one node to another is a fundamental one. It must be addressed by any distributed implementation of a statically-typed, higher-order programming language. To see how pervasive this question is, consider sendFunc, which creates an anonymous function and sends it on a channel:

```
-- wrong
sendFunc :: SendPort (Int→Int) → Int → ProcessM ()
sendFunc p x = sendChan p (λy →x +y +1)
```

Notice that the function sent on the channel, (λy $\rightarrow$ x+y+1), is a closure that captures its free variables, in this case x. In general, to serialize a function value requires that one must also serialize its free variables. However, the types of these free variables are unrelated to the type of the function value, so it is entirely unclear *how* to serialize them.

To understand this problem more deeply, let's digress for a moment and consider the problem of specifying equality on lists.

```
instance Eq a ⇒ Eq [a] where
  (x:xs) == (y:ys) = x == y && xs == ys
```

This says that, provided that Eq a holds, we can make Eq [a] hold. Eq a means that the equality operator == is defined on values of type a. Analogously, Eq [a] means that the equality operator is defined on values of type [a]: indeed, the second line above defines it. This works because equality is a structural property of the types: we know that we can define equality on a list exactly when we have an equality on the elements of that list.

Now lets get back to the problem at hand: specifying that a function is serializable. We would like to write something like:

```
instance Serializable (types of the free variables of
    an a→b) ⇒ Serializable (a→b) where ...
```

but that can't be expressed, and for good reason: serializability of a function is not a structural property of the function, because Haskell's view of a function is purely extensional. In other words, all we can do with a function is apply it; we can't introspect on its internal structure

It is not acceptable to say that functions are simply not serial-

izable, because any implementation of spawn needs to be able to specify what function to run on the remote node. For example, consider the task of creating a new remote worker process:

```
-- wrong
newWorker:: ProcessM ()
newWorker = do (s,r) ← newChan
               spawn node (do ans ← ...
                              sendChan s ans)
               ...
```

The second argument of spawn, the code that we want to run on the remote node, is a value closed over its free variables, s, which is bound to the sendPort on which the remote process should send back the answer. Thus, serializing this argument requires serializing a function with a free variable. We can't avoid this problem, because the essence of distributed computation is providing a function like spawn that starts a remote computation; in a higher-order language like Haskell, that must inevitably involve serializing a closure of some kind.

5.1 Prior Solutions

One solution to this problem is to "bake in" serialization of function values — and indeed of all values — as a primitive operation implemented directly by the runtime system. That is, the runtime system allows one to serialize *any value at all*, and transports it to the other end of the wire. Now, function closures can be serialized by serializing their free variables, and adding a representation of their code. This approach is used by every other higher-order distributed language that we know of, including Erlang (see Section 10). However, making serializability built-in has multiple disadvantages:

- It relies on a single built-in notion of serializability. In contrast, the Serializable type class introduced in Section 2.3 gives the programmer control over how values are serialized. For example, a data structure might have redundant information cached in the nodes, which should be reconstructed at the far end rather than being serialized. This is exactly what type classes are for!
- It is crucial that some types are *not* serializable. For example, we do not want the receive port of a channels to be serializable, so that senders know where to send their messages to. Similarly making TVars non-serializable guarantees that STM transactions do not span processes.
- Serializing a value and sending it over the network has an important effect on the cost model; it should not be invisible.

In the object-oriented world, Java RMI [11] also builds-in a lot of serialization machinery. Java RMI requires the programmer to specify which objects are serializable (by declaring that they implement the interface Serializable), but the programer is not required to actually write the methods that serialize the fields of the object: that is taken care of by the language implementation itself. However, Java RMI also uses introspection to provide the programmer with fine control over which fields are serialized (the transient flag prevents serialization), and exactly how the data are encoded (by providing the private methods writeObject and readObject); it also allows the programmer to take control of the whole serialization process by implementing the Externalizable interface, which means implementing the writeExternal and readExternal methods by hand. As a consequence, Java RMI avoids the three disadvantages listed above, while still automating serialization for simple

data objects.

For deserialization, however, the Java approach depends crucially on the *runtime* ability to take an arbitrary type representation and cough up a deserializer for that type. Haskell lacks this ability so this option is not open to us. Yet, as we argue above, building-in serializability of every value is too blunt an instrument. We do need *some* built-in support, but we seek something more modest. Proposing such a mechanism for a language without reflection is one of the main contributions of this paper.

5.2 Static values

We begin with a simple observation: *some* functions can be readily transmitted to the other end of the wire, namely, functions that have no free variables. For the present we make the simplifying assumption that every node is running the same code. (We return to the question of code that varies between nodes in Section 7.1.) Under this assumption, a closure without free variables can be readily serialized as a single symbolic code address (aka linker label).

To distinguish values that can be readily serialized from those that cannot, we introduce a new type constructor (Static τ) to classify such values. The type constructor Static is a new built-in primitive, enjoying a built-in serialization instance:

```
instance Serializable (Static a)
```

It is helpful to remember this intuition: *the defining property of a value of type* (Static τ) *is that it can be serialized*, and moreover, that it can be serialized without knowledge of how to serialize τ. Operationally, the implementation serializes a Static value by first

evaluating it, and then serializing the code label for the result of the

$$\begin{array}{rcl} \Gamma & ::= & \overline{x :_\delta \sigma} \\ \delta & ::= & \mathsf{S} \mid \mathsf{D} \end{array}$$

$$\Gamma\downarrow \;=\; \{x :_\mathsf{s} \sigma \mid x :_\mathsf{s} \sigma \in \Gamma\}$$

$$\frac{\Gamma\downarrow \;\vdash e : \tau}{\Gamma \vdash \mathsf{static}\ e : \mathsf{Static}\ \tau} \qquad \text{(Static intro)}$$

$$\frac{\Gamma \vdash e : \mathsf{Static}\ \tau}{\Gamma \vdash \mathsf{unstatic}\ e : \tau} \qquad \text{(Static elim)}$$

Figure 3. Typing rules for Static

evaluation. Perhaps surprisingly, this works not only for functions, but also for data values of a type like Static Tree. For these we simply statically allocate the Tree value (at compile time), and pass the label of its root.

Along with the new type constructor, we introduce new terms: (static e) to introduce the type, and (unstatic e) to eliminate it. The typing judgements for these terms are given in Figure 3. The type environment Γ is a set of variable bindings, each of the form $x :_\delta \sigma$; the subscript δ is a static-ness flag, which takes the values S (static) or D (dynamic). The idea is that top-level variables are flagged as S by giving them bindings of the form $f :_\mathsf{S} \sigma$; all other variables have dynamic bindings $x :_\mathsf{D} \sigma$. (It is straightforward to formalise this idea in the typing judgements for top-level bindings and for terms; we omit the details.) The postfix operation $\downarrow$ filters a type

environment to leave only the S bindings. The rule (Static intro) states that a term static e is well typed iff all of e's free variables are flagged S, for only then will we be able to show that $e : \tau$ in the filtered environment $\Gamma\downarrow$. In short:

- A *variable* is S-bound iff it is bound at the top level.
- A *term* (static e) has type (Static τ) iff e has type τ, and all of e's free-variables are S-bound.

Although simple, these rules have some interesting consequences:

- A variable with an S binding may have a non-Static type. Consider the top-level binding for the identity function:

  ```
  id :: a → a
  id x = x
  ```

  Because the function id is top-level, its binding in Γ will have $\delta = \mathsf{S}$, in other words id has the non-Static type id :$_\mathsf{S}$ a→a. However, (Static id) has type (Static (a → a)).

- A variable with a D binding may have a Static type. For example

  ```
  f :: Static a → (Static a, Int)
  f x = (x, 3)
  ```

  Here, x is lambda-bound and so has a D binding, but x certainly has a Static type. So fully-dynamic functions can readily compute over values of Static type.

- The free variables of a term (static e) need not have Static

types. For example, this term is well-typed:

```
static (length ∘ filter id) :: Static ([Bool] → Int)
```

Looking at the argument to `static`, we see that all of its free variables (`length`, `(∘)`, `filter`, and `id`) are bound at top-level and hence have `S` bindings. However, all these functions have their usual types.

5.3 From static values to closures

In our earlier examples `sendFunc` and `newWorker` (introduced on page 6), we wanted to transmit closures that certainly did have free variables. How do static terms help us? They help by making *closure conversion* possible. A closure is just a pair containing a code pointer and an environment. With the aid of `Static` terms we can now try to represent a closure directly in Haskell:

```
data Closure a where   -- Wrong
  MkClosure :: Static (env → a) → env → Closure a
```

`MkClosure` takes two arguments: a `Static` function and an environment of some (unknown) type `env` that is appropriate as input to the function. The fact that the type of the environment is unknown (strictly, it is existentially quantified) means that two closures with the same type may nonetheless capture environments with differing types. For example, consider the list of closures `cs`:

```
cs :: [Closure Int]
cs = [MkClosure (static negate) 3,
      MkClosure (static ord)    'x']
```

Both closures in `cs` have the same type, `Closure Int`, but the first

captures an Int as its environment, while the second captures a Char. (The function ord has type Char→Int.)

This closure type is still not serializable, because env is not serializable. This is apparently easy to solve, by asking that the environment be serializable:

```
data Closure a where   -- Still wrong
  MkClosure :: Serializable env ⇒
               Static (env → a) → env → Closure a
  deriving (Typeable)
```

Now serialization is easy:

```
instance Binary (Closure a) where
  put (MkClosure f env) = put f ≫ put env
```

But how can we *de*-serialize a closure? The difficulty is that, at the receiving end, we do not know the type captured inside the closure, so we do not know which deserializer to use. Maybe we have to send a representation of the environment type, and do a run-time type-class lookup at the receiving end? Maybe we could send some representation of the deserialization function itself? But that seems to require a solution to the problem of serializing closures, so an infinite regress beckons.

Happily, the solution is simple and, with the benefit of hindsight, obvious: perform both serialization and deserialization at *closure-construction time*, not at *closure-serialization time*. In other words, we get rid of the existential quantification, and simply require that the environment be a ByteString.

```
data Closure a where   -- Right
  MkClosure :: Static (ByteString → a) →
               ByteString → Closure a
```

Although this may sound draconian, it is perfectly general, because any environment that is serializable is equipped with encode and decode functions that will convert it to and from a ByteString. In effect, this makes the correct deserializer part of the static function in the closure. Simple, but effective.

It is easy to un-closure-convert: we just apply the function to the environment.

```
unClosure :: Closure a → a
unClosure (MkClosure f x) = unstatic f x
```

The deserialization of the environment takes place in unClosure. For a function-valued closure it makes sense to apply unClosure once, and apply the resulting function many times, so that the deserialization is performed just once.

5.4 Closures in practice

To see closures in action, here is our earlier sendFunc example, expressed using closures:

```
sendFunc :: SendPort (Closure (Int → Int))
                     → Int → ProcessM ()
sendFunc p x = sendChan p clo
  where clo = MkClosure (static sfun) (encode x)

sfun :: ByteString → Int → Int
sfun bs y = let x = decode bs
            in x + y + 1
```

The type of the items that can be sent on port p is now (Closure (Int → Int)), instead of just (Int → Int). Instead of just sending

a lambda-expression on p, we send clo, a closure containing the pre-serialized environment encode x, and the static function sfun. The latter deserializes its argument bs to get the real environment x that it expects.

Now let's look at the newWorker example. Using closures we would rewrite it like this:

```
newWorker:: ProcessM ()
newWorker = do (s,r) ← newChan
               spawn node clo
               ...
  where clo = MkClosure (static child) (encode s)

child :: ByteString → ProcessM ()
child bs = let s = decode bs
           in do ans ← ...
                 sendChan s ans
```

The type of spawn is given in Figure 2; it takes a closure as its second argument.

5.5 Summary

In this section we introduced a rather simple set of language primitives: a new type constructor Static, with built-in serialization; a new term form (static e); and a new primitive function unstatic::Static a → a.

Building on these primitives we can manually construct closures and control exactly how and when they are serialized. Performing manual closure conversion is tiresome for the programmer, and one might wonder about adding some syntactic sugar. We have not yet explored this option in depth, preferring to work out the

foundations first. However in the next section we describe some simple Template Haskell support.

6. Faking it

We have not yet implemented `static` in GHC, but we have implemented some simple workarounds that allow us (and you, gentle reader) to experiment with closures without changing GHC. We describe these workarounds in this section.

6.1 Example

Here is the code for `sendFunc` using the workarounds; we will use this as a running example.

```
sendFunc :: SendPort (Closure (Int → Int))
                              → Int → ProcessM ()
sendFunc p x = sendChan p ($(mkClosure 'add1) x)

add1 :: Int → Int → Int
add1 x y = x + y + 1

$(remotable ['add1])
```

The programmer still has to do manual closure conversion, by defining a top-level function (`add1` in this case) whose first argument is the environment (an `Int`). However, the code is otherwise significantly more straightforward than in Section 5.4.

The Template Haskell splice `$(mkClosure 'add1)` is run at compile time. Its argument `'add1` is Template Haskell notation for the (quoted) name of the `add1` function. `mkClosure` (with a small `m`) operates on the *names* of functions.

```
mkClosure :: Name →Q Exp
```

The splice expands to a call to `add1__closure`, so the net result is just as if we had written

```
sendFunc ch x = sendChan ch (add1__closure x)
```

What is `add1__closure`? It is a new top-level function created by the Template Haskell splice `$(remotable ['add1])`.

```
remotable :: [Name] →Q [Dec]
```

This splice expands to the following definitions

```
add1__closure :: Int → Closure Int
add1__closure x = MkClosure (MkS "M.add1")(encode x)

add1__dec :: ByteString → Int → Int
add1__dec bs = add1 (decode bs)

__remoteTable :: [(String, Dynamic)]
__remoteTable = [("add1", toDyn add1__dec)]
```

Thus, a call to `mkClosure` gives us a *closure generator*, which will automatically serialize arguments of the correct type and return a closure suitable for use with `spawn`. Next we see how these definitions work.

6.2 How it works

We fake the `Static` type by a string, which will serve as the label of the function to call in the remote process.

```
newtype Static a = MkS String
```

We maintain a table in the ProcessM monad that maps these strings to the appropriate implementation function composed with the environment deserializer, which in our example is `add1__dec`. This table is initialised by the call to runRemote that initialises the ProcessM monad; the table may be consulted from within the monad:

```
runRemote :: Maybe FilePath → [(String, Dynamic)]
                            → ProcessM () → IO ()
lookupStatic :: Typeable a ⇒ String → ProcessM a
```

The lookupStatic function looks up the function in the table, and performs a run-time typecheck to ensure that the value returned has the type expected by the caller. Our fake implementation of statics is therefore still type-safe; it's just that the checks happen at runtime. If either the lookup or typecheck fail, the entire process crashes, consistent with Erlang's philosophy of crash-and-recover.

Tiresomely, the programmer has the following obligations:

- In each module, write one call $(remotable [...]), passing the names of all of the functions provided as arguments to mkClosure.
- In the call to runRemote, pass a list a list of all the __remoteTable definitions, imported from each module that has a call to remotable.

Finally, the closure un-wrapping process becomes monadic, which is a little less convenient for the programmer:

```
unClosure :: Typeable a ⇒ Closure a → ProcessM a
unClosure (MkClosure (MkS s) env)
      = do f ← lookupStatic s
```

```
    return (f env)
```

7. Implementation

Cloud Haskell has been tested with recent versions of the Glasgow Haskell Compiler (GHC). The code is evolving rapidly; a current snapshot is available at http://www.github.com/jepst/CloudHaskell. Some features of the implementation are:

- Processes use Concurrent Haskell's lightweight threads. The low incremental cost of running threads is important, because a single node may need to support hundreds of processes, and processes may start and end frequently. Lightweight threads are also used to service network connections; each incoming network connection is handled by forking a new thread.
- The mergePorts family of function was tricky to implement. It would have been simple, and wrong, to fork separate threads that called receiveChan on each of several ReceivePorts, but this would cause an unfortunate race condition. Imagine what would happen if messages were received concurrently on two ports and were extracted concurrently by two threads. Only one of the messages can be returned: what is to be done with the other? Because there is no way to put a message back into the channel in the original order, we would need to either reorder the channel (breaking the FIFO property) or discard the message (breaking reliability).

  Haskell's Software Transactional Memory (STM) helps us solve this problem elegantly. STM provides a mechanism to compose receive transactions on individual message queues in

a way that ensures that only one commits.

- Template Haskell was used to write the functions mkClosure and remotable, which automatically generate code necessary to invoke remote functions, as described in Section 6.

7.1 Dynamic code update

Erlang has a nice feature that allows program modules to be updated over the wire. So, when a new version of code is released, it can be transmitted to every host in the network, where it will replace the old version of the code, without having to restart the application. We decided not to go in this direction with our framework, partly because code update is a problem that can be separated from the other aspects of building a distributed computing framework, and partly because solving it is hard. The hardness is especially prohibitive in Haskell's case, which compiles programs to machine code and lets the operating system load them, whereas Erlang's bytecode interpreter retains more control over the loading and execution of programs.

Because we have not implemented dynamic code update, Cloud Haskell code needs to be distributed to remote hosts "out of band". In our development environment this was usually done with scp and similar tools. Furthermore, this imposes on the programmer the responsibility to ensure that all hosts are running the same version of the compiled executable. Because TypeReps are nominal, and because we don't make any framework-level provision for rectifying incompatible message types, sending messages between executables that use the same name to refer to message types with different structure would most probably crash the deserializing process. A simple solution is to include with a TypeRep a fin-

gerprint of the complete type definition and everything it depends on. GHC already computes such fingerprints as part of its recompilation checking, so it should not be hard to incorporate fingerprints in a TypeRep.

8. A complete Cloud Haskell Application

As a practical example, we present a complete, albeit trivial, example of a distributed application. The application provides a remote counter, which can be incremented and queried by a client process; it is based on an Erlang program, presented by Armstrong and colleagues [1]. We'll discuss every step necessary to actually get results from this example, including the configuration and deployment processes.

Deploying a distributed application involves creating several nodes and making sure that those nodes can communicate. Each running instance of the program creates a new node. While one could in principle multiplex several nodes of a distributed application on a single physical computer, the benefits and challenges of distributed computing are more apparent when the application is running on several computers connected by a network. There are innumerable possible network configurations, and we'll address only the relatively simple case of a local TCP network with no firewall. We'll assume that each computer on which we deploy the application has a unique name. This example only needs two computers; we assume that they are named acs−01 and acs−02.

Cloud Haskell supports several configuration options, only a few of which are necessary here. These options can be set in a configuration file, which is loaded by the program, or on the command

line. Here, we'll describe how to create a suitable configuration file.

The example application distinguishes two kinds of nodes: the *worker* node will initially wait passively for instruction, whereas the *master* node will find a worker node, spawn a thread on it, and use it to compute. Even though all nodes have to run the same program, their initial action is determined by this attribute, which we call their *role*. The role of a node is simply a string, which is part of the per-node configuration. This application looks for a configuration file named config in the current directory.

We arbitrarily select the computer acs−01 to host the master node, and the computer acs−02 to host the worker node. The program, given opposite, should be compiled for both machines. Each machine also needs its own configuration file, named config. Here is the configuration for acs−01:

```
cfgRole MASTER
cfgHostName acs−01
cfgKnownHosts acs−01 acs−02
```

And here is the configuration file for acs−02:

```
cfgRole WORKER
cfgHostName acs−02
cfgKnownHosts acs−01 acs−02
```

The cfgRole option sets the role of the node. The cfgHostName option should match the name by which the host is accessible on the network. The cfgKnownHosts option sets a list of hosts where nodes might be running. The getPeers function uses the value of cfgKnownHosts to contact any running nodes on each of the named machines. getPeers will then return a PeerInfo, which can be examined with findPeerByRole to see what other nodes are available

on the network.

Of the three options given in the above configuration files, only cfgRole is strictly necessary. If cfgHostName isn't specified, the framework will ask the operating system for it. Also, cfgKnownHosts is optional, because getPeers will try to discover peers on the local network via UDP broadcast. Nevertheless, to ensure that the example applications runs on as many networks as possible, it is safer to explicitly set these values.

Once the configuration file and binary executable have been distributed to both of the hosts, we can run the program. Because the master node expects that the worker node will already be available, we start the program on the worker node first.

Here's the code for the example application:

```
1  module Main where
2  -- omitted: module imports
3  data CounterMessage = CounterQuery ProcessId
4                      | CounterShutdown
5                      | CounterIncrement
6                      deriving (Typeable)
7  -- omitted: Serializable instance of CounterMessage
8
9  counterLoop :: Int → ProcessM ()
10 counterLoop val
11   = do val' ← receiveWait [match counterCommand]
12        counterLoop val'
13   where
14     counterCommand (CounterQuery pid)
15       = do send pid val
16            return val
17     counterCommand CounterIncrement = return (val +1)
18     counterCommand CounterShutdown = terminate
19
```

```
20 $( remotable ['counterLoop] )
21
22 increment :: ProcessId → ProcessM ()
23 increment cpid = send cpid CounterIncrement
24
25 shutdown :: ProcessId → ProcessM ()
26 shutdown cpid = send cpid CounterShutdown
27
28 query :: ProcessId → ProcessM Int
29 query counterpid =
30   do mypid ← getSelfPid
31      send counterpid (CounterQuery mypid)
32      expect
33
34 go "MASTER" =
35   do aNode ←liftM (head . flip
36         findPeerByRole "WORKER") getPeers
37      cpid ← spawn aNode ($(mkClosure 'counterLoop) 0)
38      increment cpid
39      increment cpid
40      newVal ←query cpid
41      say (show newVal)  -- prints out 2
42      shutdown cpid
43
44 go "WORKER" =
45   receiveWait []
46
47 main = runRemote (Just "config")
48          [Main.__remoteTable] go
```

When the program starts, it first calls Cloud Haskell's function

runRemote (line 47), which is responsible for reading the configuration file, starting system services, and finally starting the main application-level process, which in this case is named go. Notice that runRemote is given a list of function lookup tables, one for each module; these tables are generated by remotable. The lookup tables are merged and stored in the ProcessM monad, where they are accessible to unClosure.

The go function is defined in two parts: one, on line 34, defines the behaviour of the node designated as master, while the other, on line 44, indicates that worker nodes should idly wait for the master to start a process: in this application, all work is initiated by the master. On line 35, the master calls getPeers to discover other nodes on the network. It picks the first worker node and calls spawn on line 37 to start the counterLoop function on that node. The closure in spawn's second argument is created and the closure's argument is automatically serialized. Using the ID of the new process, the main program then sends two increment commands (on lines 38 and 39) to the counter and then queries the new value (line 40), which it outputs, before asking the process to shut itself down.

counterLoop maintains the state of the counter and responds to messages pertaining to that counter. It understands three messages: increment, which will increment the value of the counter; query, which will send the current value of the counter to the sender; and shutdown, which will end the counter process. The type of these messages is declared on line 3; convenience functions, defined on lines 22, 25, and 28, are used to send the messages. The counterLoop function demonstrates a technique for storing process-local state: the function tail-recursively calls itself after processing each message, each time giving itself a new value for the counter. Handling a query message doesn't change the counter, so it tail-

recurses with the same value, whereas the increment message is handled by tail-recursing with the successor of the current value.

Notice that it is not necessary to do any explicit thread synchronization in this program. All messages sent to the counter process are handled synchronously, in FIFO order, so it is impossible to create a race condition on the counter.

9. Performance

One goal of a distributed system is to be able coordinate compute- and data-intensive algorithms among many nodes without incurring a performance overhead. Here we discuss the performance of our implementation of the k-means algorithm. The algorithm is used to identify natural clusters within sets of data points; its input is a set of data points and an integer k, and its output is an assignment of each point to one of k clusters.

The k-means algorithm can be described as a generalization of MapReduce [4]. Each node is assigned a role of either *mapper* or *reducer*; the data points are divided evenly between the mappers. Each mapper then calculates the distance between each of its data points and the (initially random) centroid of each cluster, and on that basis assigns each point to the nearest cluster. The new assignments are collected by the reducer nodes and used to generate new cluster centroids. The new centroids are sent back to the mappers and used to repeat the algorithm, until either the centroids stop changing or the maximum number of iterations is reached.

k-means is computationally demanding, data intensive, and easily partitioned, and so provides a useful test case for Cloud Haskell. We compared the performance of a Cloud Haskell implementation of k-means with the Apache Mahout implementation, which

is based on the Hadoop framework. We deployed both versions on an Amazon EC2 cluster, giving them one million randomly generated 100-dimensional data points as input, and extracted $k = 10$ clusters. The number of iterations was fixed at five. We used one reducer node, while the number of mapper nodes was varied between one and 80. Each node was executed on an Amazon instance of type `m1.small`, which is a single core virtual machine with 1.7 GB of memory. The results of the tests are summarized in Figure 4.

The results show that Cloud Haskell has performance roughly comparable with that of the Hadoop framework, and in some cases superior. Although the Hadoop implementation performs better with fewer nodes, the Cloud Haskell version overtakes it as more nodes are added, and retains this lead. The greatest bottleneck in Cloud Haskell's performance is acquiring and loading the data; we hope to address this issue by improving file handling.

10. Related work

As should be clear, our main inspiration comes from Erlang [1], whose tremendous success for distributed programming led us to emulate its best features. A second inspiration is the Ciel execution engine and Skywriting language of Murray *et al* [9, 10].

In scientific computing, the most scalable solution for distributed parallel computation is the Message Passing Interface (MPI). As its name suggests, MPI is similar to our framework in

Performance of k-means on EC2 cluster

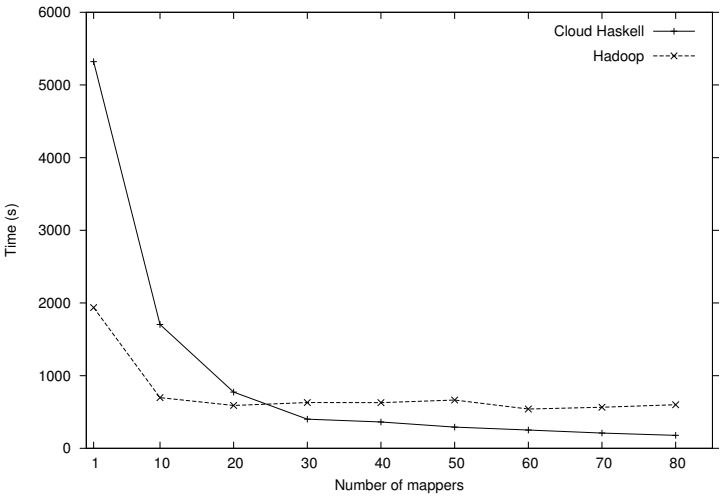


Figure 4. The run-time of the k-means algorithm, implemented under Cloud Haskell and Hadoop. The input data was one million 100-dimensional data points.

its preference for communication by messages. Unlike our framework, MPI is language-independent; interfaces are available for Fortran, C, and C++, and an array of different implementations have been optimized for a variety of supercomputer architectures. In comparison, Cloud Haskell, as a DSL embedded in Haskell, is not easily available from other languages. This is of course a weakness, but it is also a great strength. Haskell's type system separates shared-memory concurrency, distributed-memory concurrency and

pure computation, giving programmers powerful reasoning tools that are not available when using a subroutine library called from a language with a weaker type system.

There have been lots of mechanisms for executing functions on remote systems, staring with Birrell and Nelson's RPC [3]. Specific to Java is the Remote Method Invocation (RMI) mechanism [11]. CORBA provides the Interface Description Language (IDL) [19] to define implementation-language-independent remote functions. Web services often use the SOAP standard to implement remote procedure call. However, none of these mechanisms deal with the problem of transmitting first-class functions that capture their environment. Although it may be argued that any object-oriented language that supports mobile objects does as a consequence also support mobile functions, there have been few full implementations of mobile objects. A notable exception is Emerald [8], which dealt with the issue of environment capture by copying the relevant parts of the environment into an object when it was created. All objects, including stack frames, were mobile; serialization was handled by the run-time system and depended on introspection.

Implementing a framework for distributed computing in Haskell has been done before. One example is Glasgow Distributed Haskell [12], which differs form our work in that it maintains the semantics of shared-memory concurrency in a distributed environment, rather than insisting on message-based communication between nodes. We argue that this is undesirable because it gives the programmer no model for reasoning about the costs of the computation.

Another example is Eden [**?**], an extension to Haskell that provides distributed processes that encapsulate functions. The construction, use, and serialization of these processes are built into the

implementation, which will copy all of the bindings needed for the evaluation of a function's free variables. Eden also supports a rich library of higher-order functions, as well as a debugging tools such a trace viewer.

Our operations on channels are somewhat reminiscent of Concurrent ML's events [14]. However, Concurrent ML does not support distribution because that would be incompatible with its model of synchronization. paraML, an experimental variant of Concurrent ML, did have a distributed implementation; its serialization mechanism was entirely built-in [2].

The Acute language does a very solid job of dealing with type-safe marshalling, based on passing type representations at runtime [16]. Serialization is mostly built-in, with some programmer control offered for "rebinding", when values should be rebound rather than serialized. HashCaml, currently in alpha-release, is a variant of OCaml that supports type-safe distributed programming, including serialization of function values. Like Acute it uses explicit type passing, the details of how the free variables of function closures are marshalled are hazy. Rossberg's language Alice likewise supports type-safe distributed programming [15]. The Clean language supports type-safe saving of arbitrary values (including function closures) into files [18]; a recent paper proposes a mechanism (not yet implemented) for solving class constraints at runtime [17, §4.2].

A fundamental feature of all of these systems is that serialization of funciton closures is built-in. In contrast, our work puts the programmer in complete control of serialization, via the usual type-class mechanism. Exposing closure conversion is a significant burden — albeit one that can be relieved with some syntactic sugar. The payoff is that the programmer can reason about the cost of sending a message. We do not argue that our approach is always

better, but rather that it offers a new and unexplored design point.

11. Conclusions and Future Work

Cloud Haskell, as presented in this paper, provides a good starting point for building a distributed application. However, as yet it is no more than this: we have a lot of work left to do to make "Haskell in the Cloud" a reality.

Our ongoing work is on two levels. At the lower level, we plan to implement `Static` and the corresponding type inference rules in GHC. This will remove the need for the workarounds described in Section 6. At the higher level, we have designed a framework that builds on the interface described here. In it, the main unit of abstraction changes from the process to the *task*: an idempotent, restartable block of code that produces a well-defined result. The task layer of the framework, like the process layer presented in this paper, is accessible as a domain-specific language (DSL) embedded in Haskell as a monad.

Whereas the DSL described here lets programmers start processes, exchange messages, and detect failure, the task-based framework takes care of allocating tasks to physical resources, resolving data dependencies between tasks, and automatically recovering from failure. To make this possible, the task layer represents the computation as a directed acyclic graph, in which tasks are the vertices and the data dependencies between them are the edges, and are exposed to the programmer as *promises*. We hope that the task layer will provide functionality similar to well-known distributed frameworks such as MapReduce [4] and Dryad [7], although our immediate inspiration comes from the Skywriting project [9, 10].

Acknowledgments

We particularly thank John Launchbury who helped us see that Static τ for an arbitrary type τ would be an improvement over a static *function* type $\sigma \xrightarrow{\#} \tau$. John Hughes helped us to understand some details of Erlang, and we enjoyed discussing with him how to express Erlang's computational model in Haskell. Koen Claessen insightfully pointed out the importance of not making receive ports serializable. We thank Thomas van Noort, John Reppy, Peter Sewell, and the anonymous referees for helpful feedback. Andrew Black thanks Microsoft Research, Ltd, for providing a congenial home during his sabbatical. Jeff Epstein thanks Alan Mycroft for his support and helpful feedback.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. Concurrent programming in Erlang, 1993.

[2] P. Bailey. *Process-oriented language design for distributed address spaces*. PhD thesis, Australian National University, Jan. 1997.

[3] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *TOCS*, 2(1):39–59, 1984.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008. ISSN 0001-0782.

[5] *The GNU Mach Reference Manual*. Free Software Foundation, Inc., for system version 1.3.99. edition, November 2008. http://www.gnu.org/software/hurd/gnumach-doc/.

[6] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007. ISSN 0163-5980.

[8] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *TOCS*, 6(1):109–133, 1988.

[9] D. G. Murray and S. Hand. Scripting the cloud with Skywriting. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[10] D. G. Murray and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *NSDI '11: Proceedings of the eighth symposium on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2011. USENIX.

[11] E. Pitt and K. McNiff. *Java.rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0201700433.

[12] R. Pointon, P. Trinder, and H.-W. Loidl. The design and implementation of Glasgow Distributed Haskell. In M. Mohnen and P. Koopman, editors, *Implementation of Functional Languages*, volume 2011 of *Lecture Notes in Computer Science*, pages 53–70. Springer Berlin / Heidelberg, 2001.

[13] R. F. Rashid and G. G. Robertson. Accent: A communication oriented network operating systems kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 64–75. Association for Computing Machinery, October 1981.

[14] J. Reppy. *Concurrent programming in ML*. Cambridge University Press, 1999.

[15] A. Rossberg. *Typed Open Programming — A higher-order, typed approach to dynamic modularity and distribution*. PhD thesis, Universitat des Saarlandes, Jan. 2007.

[16] P. Sewell, J. Leifer, K. Wansbrough, F. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: high level programming language design for distributed computation. *Journal of Functional Programming*, 17(4–5), 2007.

[17] T. van Noort, P. Achten, and R. Plasmeijer. Ad-hoc polymorphism and dynamic typing in a statically typed functional laguage. In *Proc 6th ACM Workshop on Generic Programming, Baltimore*. ACM, Sept. 2010.

[18] M. Vervoot and R. Plasmeijer. Lazy dynamic input/output in the lazy functional language clean. In *Proc 14th Workshop on the Implementation of Functional Languages (IFL'02)*, pages 101–117. Spinger LNCS 2670, Sept. 2002.

[19] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, Feb. 1997. ISSN 0163-6804. doi: 10.1109/35.565655.