

# Cloud Haskell Semantics (DRAFT)

Well-Typed LLP

November 7, 2012

## 1 Introduction

Cloud Haskell brings Erlang-style concurrency to Haskell in the form of a library.<sup>1</sup> The original publication [Epstein et al., 2011] is a good introduction to Cloud Haskell, and is accompanied by an ever-growing number of resources, collected on the Haskell Wiki.<sup>2</sup> The current document augments the original publication about Cloud Haskell by giving a precise semantics to the Cloud Haskell primitives. It is meant as a reference, not an introduction.

The original Cloud Haskell paper stipulates that messages are “asynchronous, reliable, and buffered”, but does not describe how this can be achieved. Understanding “reliable” to mean “reliable ordered” (or “TCP-like”), the reliability of message delivery comes from the reliability of the underlying network protocol—up to a point.

The problem is that the underlying network protocol is connection-oriented, but Cloud Haskell is not. Intuitively, when  $P$  sends a message to  $Q$ , we open a reliable-ordered connection from  $P$  to  $Q$ . Reliability of message delivery now follows from reliability of the network protocol, until  $P$  somehow gets disconnected from  $Q$ . If  $P$  now sends another message to  $Q$ , the implementation cannot simply reconnect: after all, some messages that were sent on the first connection might not have been delivered. This means that  $P$  might send  $m_1, m_2, m_3$  to  $Q$ , but  $Q$  will receive  $m_1, m_3$ .<sup>3</sup>

One (non-)solution is for  $P$  to buffer all messages it sends to  $Q$ , and remove messages from this buffer only when  $Q$  acknowledges that it received them. On a reconnect  $P$  must ask which message  $Q$  last received, and retransmit the rest. This means that when  $P$  gets disconnected from  $Q$ , it must infinitely buffer all messages sent to  $Q$  (until a connection is reestablished). However, infinite buffering is too strong a requirement; moreover, this is unsatisfactory because it means implementing a reliable protocol on top of the underlying reliable network protocol. We would like a different solution.

Instead, Cloud Haskell does *not* attempt to reconnect automatically, but provides a `reconnect` primitive which gives programmers the option of reconnecting manually. This is an explicit acknowledgement from the programmer that message loss might occur, and forces them to consider how such loss might be dealt with.

The semantics we present is based on [Svensson et al., 2010], which we will refer to as the “Unified” semantics. However, we will present the semantics in a more “Haskell” style following the semantics for STM [Harris et al., 2008]. It differs from the Unified semantics in that

1. We introduce an explicit notion of *reconnecting* (with potential message loss)
2. We simplify the semantics: we “flatten” sets of nodes of processes as sets of processes (but assume a mapping from process identifiers to node identifiers), do not have per-process mailboxes (but only the system queue or “ether”) and do not have an explicit concept of node controllers

Our semantics differs from the STM semantics in that we pretend that the Cloud Haskell `Process` monad is the top-level monad, and do not consider the `IO` monad at all. Current imprecisions with respect to the “real” Cloud Haskell are

1. We ignore the issue of serializability, other than to say that the semantics will get stuck when trying to send a non-serializable payload; consequently, we do not formalize `static`
2. We do not formalize all Cloud Haskell primitives (merging of ports, “advanced messaging”, and others)

---

<sup>1</sup><http://hackage.haskell.org/package/distributed-process>

<sup>2</sup><http://www.haskell.org/haskellwiki/CloudHaskell>

<sup>3</sup>Indeed, message passing in Erlang is ordered but unreliable for the same reason [Svensson and Fredlund, 2007].

$\frac{\llbracket M \rrbracket = V \quad M \neq V}{M \rightarrow V} \text{EVAL} \quad \frac{M \rightarrow N}{\langle \mathbb{P}[M]_{pid}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathbb{P}[N]_{pid}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle} \text{ADMIN}$								
<table style="width: 100%; border: none;"> <tr> <td style="padding: 5px;"><code>return</code> <math>N \gg= M \rightarrow MN</math></td> <td style="padding: 5px; text-align: center;">BIND</td> <td style="padding: 5px;"><code>catch</code> <math>(\text{return } M) N \rightarrow \text{return } M</math></td> <td style="padding: 5px; text-align: right;">CATCH<sub>1</sub></td> </tr> <tr> <td style="padding: 5px;"><code>throw</code> <math>N \gg= M \rightarrow \text{throw } N</math></td> <td style="padding: 5px; text-align: center;">THROW</td> <td style="padding: 5px;"><code>catch</code> <math>(\text{throw } M) N \rightarrow NM</math></td> <td style="padding: 5px; text-align: right;">CATCH<sub>2</sub></td> </tr> </table>	<code>return</code> $N \gg= M \rightarrow MN$	BIND	<code>catch</code> $(\text{return } M) N \rightarrow \text{return } M$	CATCH <sub>1</sub>	<code>throw</code> $N \gg= M \rightarrow \text{throw } N$	THROW	<code>catch</code> $(\text{throw } M) N \rightarrow NM$	CATCH <sub>2</sub>
<code>return</code> $N \gg= M \rightarrow MN$	BIND	<code>catch</code> $(\text{return } M) N \rightarrow \text{return } M$	CATCH <sub>1</sub>					
<code>throw</code> $N \gg= M \rightarrow \text{throw } N$	THROW	<code>catch</code> $(\text{throw } M) N \rightarrow NM$	CATCH <sub>2</sub>					

**Figure 1:** Administrative Transitions

3. Some of the concepts that we do formalize are lower-level concepts; for instance, the primitive `spawn` that we formalize is asynchronous (following the Unified semantics); a synchronous construct can be derived.

## 2 Preliminaries

Cloud Haskell Processes run on *nodes*. Processes communicate by sending messages to each other (directly or using typed channels). Processes can also send messages to nodes (for instance, a request to spawn or monitor a process).

We assume disjoint countable sets `NodeId`, `ProcessId`, and `ChannelId`, changed over by *nid*, *pid* and *cid* respectively, and representing process identifiers, node identifiers, and (typed) channel identifiers. We assume the existence of total functions

$$\begin{aligned} \text{node} &: (\text{ProcessId} \uplus \text{ChannelId}) \rightarrow \text{NodeId} \\ \text{process} &: \text{ChannelId} \rightarrow \text{ProcessId} \end{aligned}$$

and define

$$\text{Identifier} = \text{NodeId} \uplus \text{ProcessId} \uplus \text{ChannelId}$$

and let *id* range over `Identifier`.

We represent a process as a pair  $M_{pid}$  of a term *M* and a process ID *pid*. We will denote a set of processes as

$$M_{pid} \parallel \dots \parallel N_{pid'}$$

A *system*  $\langle \mathcal{P}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle$  is a tuple containing a set  $\mathcal{P}$  of processes, a *system queue*  $\mathcal{Q}$ , a *blacklist*  $\mathcal{B}$ , and a set of monitors  $\mathcal{M}$ . The set of monitors  $\mathcal{M}$  is a set of tuples  $(id_{to}, pid_{fr}, nid, ref)$  which records that node *nid* knows that process  $pid_{fr}$  is monitoring  $id_{to}$  (*ref* is the monitor reference). The system queue is a set of triples  $(id_{to}, id_{fr}, message)$  of messages that have been sent but not yet processed. The blacklist records disconnections and is represented as a of pairs  $(id_{to}, id_{fr})$ .

## 3 Semantics

We follow the STM semantics as closely as possible. The language is the same except for the primitives considered, and we use the same concept of evaluation contexts:

value	$V$	::=	$id \mid \lambda x. M \mid \text{return } M \mid M \gg= N \mid \text{throw } M \mid \text{catch } M N \mid \text{expect} \mid \text{send } pid M \mid \text{spawn } nid M \mid \text{monitor } id$
term	$M, N$	::=	$x \mid V \mid MN \mid \dots$
Evaluation	$\mathbb{E}$	::=	$\square \mid \mathbb{E} \gg= M \mid \text{catch } \mathbb{E} M$
context	$\mathbb{P}_{pid}$	::=	$\mathbb{E} \square_{pid} \parallel \mathcal{P}$

Indeed, the “administrative” transitions are identical (Figure 1).

### 3.1 Disconnect and Reconnect

Figure 2 gives the semantics for disconnecting and reconnecting. Rule `DISCONNECT` models random network disconnect between nodes  $nid_1$  and  $nid_2$ . We assume that entire *nodes* get disconnected from each other, not individual processes. *Reconnecting* however is on a per connection basis (`RECON-EX`). Connections to and from controllers can be implicitly reconnected (`RECON-IM1` and `RECON-IM2`).

$$\begin{array}{c}
\frac{nid_1 \neq nid_2}{\langle \mathcal{P}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathcal{P}; \mathcal{Q}; \mathcal{B} \cup (\overline{nid_1} \times \overline{nid_2}); \mathcal{M} \rangle} \text{DISCONNECT} \\
\frac{}{\langle \mathbb{P}[\text{reconnect } id_{to}]_{pid_{fr}}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathbb{P}[\text{return } ()]_{pid}; \mathcal{Q}; \mathcal{B} \setminus (id_{to}, pid_{fr}); \mathcal{M} \rangle} \text{RECON-EX} \\
\frac{}{\langle \mathcal{P}; \mathcal{Q}; \mathcal{B}, (nid_{to}, id_{fr}); \mathcal{M} \rangle \rightarrow \langle \mathcal{P}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle} \text{RECON-IM}_1 \quad \frac{}{\langle \mathcal{P}; \mathcal{Q}; \mathcal{B}, (id_{to}, nid_{fr}); \mathcal{M} \rangle \rightarrow \langle \mathcal{P}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle} \text{RECON-IM}_2
\end{array}$$

where

$$\overline{nid} \subset \text{Identifier} = \{nid\} \cup \{pid \mid \text{node}(pid) = nid\} \cup \{cid \mid \text{node}(cid) = nid\}$$

**Figure 2:** Disconnect and Reconnect

$$\begin{array}{c}
\frac{}{\langle \mathbb{P}[\text{send } pid_{to} M]_{pid_{fr}}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathbb{P}[\text{return } ()]_{pid_{fr}}; \mathcal{Q} \triangleright_{\mathcal{B}} (pid_{to}, pid_{fr}, M); \mathcal{B}; \mathcal{M} \rangle} \text{SEND} \\
\frac{id_{fr} \notin \text{senders}(\mathcal{Q})}{\langle \mathbb{P}[\text{expect}]_{pid_{to}}; \mathcal{Q}, (pid_{to}, id_{fr}, M), \mathcal{Q}'; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathbb{P}[\text{return } M]_{pid_{to}}; \mathcal{Q}, \mathcal{Q}'; \mathcal{B}; \mathcal{M} \rangle} \text{EXPECT} \\
\frac{cid \text{ fresh} \quad \text{process}(cid) = pid}{\langle \mathbb{P}[\text{newChan}]_{pid}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathbb{P}[\text{return } (cid^s, cid^r)]_{pid}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle} \text{NEWCHAN} \\
\frac{}{\langle \mathbb{P}[\text{sendChan } cid_{to}^s M]_{pid_{fr}}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathbb{P}[\text{return } ()]_{pid_{fr}}; \mathcal{Q} \triangleright_{\mathcal{B}} (cid_{to}, pid_{fr}, M); \mathcal{B}; \mathcal{M} \rangle} \text{SENDCHAN} \\
\frac{pid_{fr} \notin \text{senders}(\mathcal{Q})}{\langle \mathbb{P}[\text{receiveChan } cid_{to}^r]_{pid}; \mathcal{Q}, (cid_{to}, pid_{fr}, M), \mathcal{Q}'; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathbb{P}[\text{return } M]_{pid}; \mathcal{Q}, \mathcal{Q}'; \mathcal{B}; \mathcal{M} \rangle} \text{RECEIVECHAN} \\
\frac{ref \text{ fresh}}{\langle \mathbb{P}[\text{spawn } nid M]_{pid}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathbb{P}[\text{return } ref]_{pid}; \mathcal{Q} \triangleright_{\mathcal{B}} (nid, pid, \text{spawn } ref M); \mathcal{B}; \mathcal{M} \rangle} \text{SPAWN-ASYNC} \\
\frac{pid \notin \text{senders}(\mathcal{Q}) \quad pid' \text{ fresh} \quad \text{node}(pid') = nid}{\langle \mathcal{P}; \mathcal{Q}, (nid, pid, \text{spawn } ref M), \mathcal{Q}'; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathcal{P} \parallel M_{pid'}; \mathcal{Q}, \mathcal{Q}' \triangleright_{\mathcal{B}} (pid, nid, \text{spawned } ref pid'); \mathcal{B}; \mathcal{M} \rangle} \text{SPAWN-EXEC}
\end{array}$$

where

$$\begin{array}{ll}
\mathcal{Q} \triangleright_{\mathcal{B}} (id_{to}, id_{fr}, M) = \mathcal{Q}, (id_{to}, id_{fr}, M) & \text{if } (id_{to}, id_{fr}) \notin \mathcal{B} \\
\mathcal{Q} \triangleright_{\mathcal{B}} (id_{to}, id_{fr}, M) = \mathcal{Q} & \text{otherwise}
\end{array}$$

( $\triangleright_{\mathcal{B}}$ ) is only defined for serializable payloads.

**Figure 3:** Basic Primitives

### 3.2 Communication

The semantics for the basic primitives are listed in Figure 3. Once a connection has been blacklisted, no further messages can be sent across that connection (until an explicit or implicit reconnect).

Message passing is ordered but only for a given sender and receiver; no ordering guarantees exist between messages sent by different processes. For that reason rules EXPECT, RECEIVECHAN and SPAWN-EXEC choose the *first* message of a *randomly chosen* sender.

Since the semantics is not type-driven, we represent typed channels simply as an identifier with an annotation whether it is the send-end ( $cid^s$ ) or the receive end ( $cid^r$ ) of the channel (rules NEWCHAN, SENDCHAN and RECEIVECHAN).

Spawning finally is asynchronous. When process  $P$  spawns process  $Q$  on node  $nid$  (rule SPAWN-ASYNC) a message is sent to node  $nid$  (which may, of course, never arrive). When the remote node receives the message and actually spawns  $Q$  (rule SPAWN-EXEC) it sends a message back to  $P$  with  $Q$ 's process ID. In Cloud Haskell this primitive is called `spawnAsync`, and the Cloud Haskell `spawn` primitive is defined in terms of `spawnAsync` (we do not consider the synchronous version in this document).

$\frac{nid_{fr} = \text{node}(pid_{fr}) \quad nid_{to} = \text{node}(id_{to}) \quad \text{ref fresh}}{\langle \mathbb{P}[\text{monitor } id_{to}]_{pid_{fr}}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathbb{P}[\text{return } ref]_{pid_{fr}}; \mathcal{Q} \triangleright_{\mathcal{B}} (nid_{to}, pid_{fr}, \text{monitor } ref \ id_{to}); \mathcal{B}; \mathcal{M}, (id_{to}, pid_{fr}, nid_{fr}, ref) \rangle}$	MON-LOC
$\frac{pid_{fr} \notin \text{senders}(\mathcal{Q})}{\langle \mathcal{P}; \mathcal{Q}, (nid_{to}, pid_{fr}, \text{monitor } ref \ id_{to}), \mathcal{Q}'; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathcal{P}; \mathcal{Q}, \mathcal{Q}'; \mathcal{B}; \mathcal{M}, (id_{to}, pid_{fr}, nid_{to}, ref) \rangle}$	MON-REM
$\frac{nid_{fr} = \text{node}(pid_{fr}) \quad (id_{to}, pid_{fr}) \in \mathcal{B}}{\langle \mathcal{P}; \mathcal{Q}; \mathcal{B}; \mathcal{M}, (id_{to}, pid_{fr}, nid_{fr}, ref) \rangle \rightarrow \langle \mathcal{P}; \mathcal{Q}, (pid_{fr}, id_{to}, \text{discon } ref); \mathcal{B}; \mathcal{M} \rangle}$	MON-DIS
$\frac{\text{node}(pid) = nid}{\langle (\text{return } ())_{pid} \parallel \mathcal{P}; \mathcal{Q}; \mathcal{B}; \mathcal{M}, (pid, pid', nid, ref) \rangle \rightarrow \langle (\text{return } ())_{pid} \parallel \mathcal{P}; \mathcal{Q} \triangleright_{\mathcal{B}} (pid', nid, \text{died } ref); \mathcal{B}; \mathcal{M} \rangle}$	MON-RET
$\frac{\text{node}(pid) = nid}{\langle (\text{throw } M)_{pid} \parallel \mathcal{P}; \mathcal{Q}; \mathcal{B}; \mathcal{M}, (pid, pid', nid, ref) \rangle \rightarrow \langle (\text{throw } M)_{pid} \parallel \mathcal{P}; \mathcal{Q} \triangleright_{\mathcal{B}} (pid', nid, \text{exc } ref \ M); \mathcal{B}; \mathcal{M} \rangle}$	MON-THROW
$\frac{\text{node}(pid) = nid \quad pid \notin \mathcal{P}}{\langle \mathcal{P}; \mathcal{Q}; \mathcal{B}; \mathcal{M}, (pid, pid', nid, ref) \rangle \rightarrow \langle \mathcal{P}; \mathcal{Q} \triangleright_{\mathcal{B}} (pid', nid, \text{unknown } ref); \mathcal{B}; \mathcal{M} \rangle}$	MON-UNKNOWN

**Figure 4:** Monitoring

$\frac{\text{node}(pid) = nid \quad \nexists pid', ref \cdot (pid', pid, nid, ref) \in \mathcal{M}}{\langle (\text{return } ())_{pid} \parallel \mathcal{P}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathcal{P}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle}$	TERMRET
$\frac{\text{node}(pid) = nid \quad \nexists pid', ref \cdot (pid', pid, nid, ref) \in \mathcal{M}}{\langle (\text{throw } M)_{pid} \parallel \mathcal{P}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle \rightarrow \langle \mathcal{P}; \mathcal{Q}; \mathcal{B}; \mathcal{M} \rangle}$	TERMTROW

**Figure 5:** Process Termination

### 3.3 Monitoring

When process  $P$  on node  $nid_P$  wants to monitor process  $Q$  on node  $nid_Q$  both nodes must be notified. The local node it notified immediately, and a message is sent to to the remote node (rules MON-LOC and MON-REM). As with all messages, the message to the remote node might be lost. When  $P$  is disconnected from  $Q$  it is the responsibility of  $P$ 's local node  $nid_P$  to notify  $P$  (MON-DIS); when  $Q$  terminates (MON-RET) or crashes (MON-THROW) it is the responsibility of the remote node  $nid_Q$  to notify  $P$ .

### 3.4 Process Termination

The rules for normal and abnormal process termination are defined in Figure 5. When a process crashes it dies silently, unless monitors are setup.

## 4 Open Issues

### 4.1 Ordering of Monitor/Link Notifications

The semantics as described above, like the original Unified semantics, does not guarantee that messages send from process  $P$  to process  $Q$  (SEND) are ordered with respect to the link or monitor notification sent when process  $P$  terminates normally or abnormally (MON-RET, MON-THROW). This means that if process  $P$  does

```
receiveWait [
  match $ \(Reply reply) -> ...
  , match $ \(ProcessMonitorNotification ..) -> ...
]
```

and process  $Q$  does

```
send pidA (Reply reply)
// terminate or indeed throw an exception
```

then the semantics does not guarantee that the reply from process  $Q$  will arrive at process  $P$  before the monitor notification; hence, this results in an (artificial) race condition in process  $P$ .

One possible solution is to regard such a link notification as a message from process  $Q$  to process  $P$ , which should be ordered along with the other messages.

## 4.2 Ordering and Typed Channels

The situation is more tricky still than sketched above because we of typed channels. The semantics does not provide ordering guarantees between messages sent directly to the process and messages sent on typed channels. That means that even if we consider a link or monitor notification as a message sent to the process, to be ordered with other messages sent to that process, it is still unordered with respect to messages sent on typed channels. This means that we have similar race conditions<sup>4</sup>

A possible solution is to insist that *all* messages from process  $P$  to process  $Q$  are ordered, no matter how they are sent. From an implementation point of view, this would entail the use of a single ordered network connection for all messages from  $P$  to  $Q$ , rather than using an ordered connection per typed channel plus one for direct messages.

## References

- [Epstein et al., 2011] Epstein, J., Black, A. P., and Peyton-Jones, S. (2011). Towards Haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA. ACM.
- [Harris et al., 2008] Harris, T., Marlow, S., Jones, S. P., and Herlihy, M. (2008). Composable memory transactions. *Commun. ACM*, 51(8):91–100.
- [Svensson and Fredlund, 2007] Svensson, H. and Fredlund, L.-A. (2007). A more accurate semantics for distributed erlang. In *Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, ERLANG '07, pages 43–54, New York, NY, USA. ACM.
- [Svensson et al., 2010] Svensson, H., Fredlund, L.-Å., and Benac Earle, C. (2010). A unified semantics for future Erlang. In *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*, Erlang '10, pages 23–32, New York, NY, USA. ACM.

---

<sup>4</sup>Even ignoring the fact that we currently don't even provide a way to wait for a message on a typed channel *or* a direct message.