Towards an Isabelle/HOL Formalisation of Core Erlang

Joseph R. Harrison
University of Kent
School of Computing
Canterbury, England
jrh53@kent.ac.uk

Abstract

As part of broader work to improve the safety of Erlang systems, we are attempting to detect (and prevent) messages which remain forever unreceived in process' mailboxes using a mix of static and runtime analysis. We have formalised the communicating portion of Core Erlang using Isabelle/HOL, an interactive theorem prover. We can use the Isabelle toolchain to prove properties of our model, automatically prepare documents, and generate verified executable code in a variety of functional programming languages.

We formally model a communicating fragment of Core Erlang in a language we call CoErl. After defining the evaluation of expressions, we model the process-local and concurrent semantics of the language using a labelled transition system. We also introduce the notion of mailbox traces which capture communication events during process execution. This is followed by some illustrative examples of the concurrent semantics.

Although our CoErl model is a solid foundation for a full formalisation of Core Erlang, it currently lacks higher-order and recursive behaviour. Isabelle/HOL has proved practical for formalising and verifying several properties of CoErl and its trace system, while ongoing and future work focuses on bringing the language to feature parity with Core Erlang and Erlang/OTP.

CCS Concepts • **Software and its engineering** → **Concurrent programming languages**; **Semantics**; *Software verification*; Automated static analysis; • **Theory of computation** → *Process calculi*;

Keywords Erlang, Core Erlang, Isabelle, labelled transition systems, asynchronous communication, proof assistant

ACM Reference format:
Joseph R. Harrison. 2017. Towards an Isabelle/HOL Formalisation of Core Erlang. In *Proceedings of 16th ACM SIGPLAN International Workshop on Erlang, Oxford, UK, September 8, 2017 (Erlang'17),* 9 pages.
https://doi.org/10.1145/3123569.3123576

1 Introduction

Erlang's approach to program construction is relatively easy going: typing is weak, and other aspects of the language mean that some properties of Erlang systems can only be checked dynamically. With the longer-term aim of guaranteeing trustworthiness of such systems, we are examining ways of combining static analysis at compile-time with dynamic runtime diagnostics and monitoring.

We have chosen to first consider orphan messages in Erlang systems: messages sent to a process which are never received. The memory leaks involved may not be immediately apparent to new or OTP-focused Erlang programmers. Orphan messages will accumulate in process mailboxes, consuming (and possibly exhausting) system memory.

Drawing inspiration from type-driven development principles, we are producing a lightweight ad-hoc communication contract system for Erlang/OTP. This will use a mixture of static and runtime analysis. With static analysis requirements in mind, we have chosen Core Erlang rather than Erlang itself as our basis.

So far we have produced a formal model of Erlang's asynchronous and out-of-order communication. We have using the Core Erlang specification [3] as a reference for our model, deferring to Erlang/OTP for BIF behaviour.

This formalisation has been created in Isabelle/HOL, an interactive theorem proving environment which supports a literate proof language, higher-order and recursive functions, code generation for a variety of functional programming languages, and LaTeX-based document preparation. A formalisation in a theorem prover such as Isabelle/HOL allows us not only to execute the semantics (via code generation) but also to prove that the semantics hold appropriate "hygiene" conditions.

After outlining the language, representations of processes, and the `receive` operation (section **??**), we define the semantics of individual processes (section 4).

Drawing inspiration from labelled transition systems, we define a set of labels over the process-local semantics (sec 5) before introducing the concept of strong(section 5.3) and weak (section 5.3) communication traces. Our "hygiene" conditions are then defined and proven: soundness and completeness with respect to the process-local semantics (section 5.4).

Ultimately, this transition system is used to model an Erlang node in an entirely formal context using the labels as *meeting points* between processes. This is followed by a few examples of communication in a node, using Isabelle's literate proof language to show each intermediate step (section 7).

The language is finally compared to Core Erlang to determine the next best steps in formalisation. We consider other calculi and formal models of Erlang alongside existing work on both static and runtime analysis of Erlang systems (section 10).

2 Background

Core Erlang serves as an official intermediate representation in Erlang/OTP [3]. Originally developed in collaboration with the HiPE project [21], the language seeks to be as regular as possible: it removes syntactic sugar and replaces the sometimes-confusing = operator with a more transparent `let` syntax.

We have chosen Core Erlang specifically for this reason: it is intended to be simpler for static analysis purposes [2]. As an added bonus, Core Erlang code can be converted back to Erlang (useful in source transformations), and line numbers are preserved (useful for error messages).

2.1 Isabelle/HOL

Isabelle is the generic proof assistant we have chosen to produce our model with. The system uses higher-order logic and Standard ML at its core. Isabelle/HOL is the jEdit-based environment which facilitates interactive use of Isabelle [23].

Isabelle proofs can be provided in two languages: apply style and Isar style. Apply style proofs use a concise syntax which makes the proof state difficult to inspect on paper and is somewhat prone to breakage, whereas the Isar style permits a more literate and explicit proof body [22].

Isabelle also famously features the sledgehammer tool: various external automatic theorem provers can be invoked at any time during a backwards proof via the `sledgehammer` keyword. If a proof is found, sledgehammer responds with the Isabelle commands necessary to prove the subgoal. These typically use the `metis` or `metis` provers.

The toolchain has other features: functions and inductive definitions can be exported to a variety of target languages including Haskell, Scala, and OCaml. By this method, it is possible to generate verified implementations. As an example, the seL4 microkernel has been fully verified in Isabelle [12]. Furthermore, Isabelle can generate LaTeX output from its theory files. As a demonstration, sections **??** through 6 inclusive have been generated from Isabelle theory files.

2.2 Introduction to Isabelle Syntax

Isabelle is a complex system with a variety of syntactic sugar, seemingly synonymous keywords, and an ML-like syntax. The Isabelle/Isar reference manual [24] provides comprehensive documentation of the system, while a introduction is provided here to serve as a reference for proofs in this document.

Functions and Definitions As HOL is a logic of total functions, definitions in Isabelle must have a proven termination order. Functions defined via the *fun* keyword will cause Isabelle to automatically attempt a termination proof. If this is not sufficient, the similar *function* keyword offers more flexibility at the cost of more complexity: the user must prove termination themselves.

Functions which use primitive recursion over algebraic datatypes are defined via *primrec*, with Isabelle again automatically dispatching the termination proof.

A more powerful feature is the ability to create *inductive* definitions complete with predicates and names. The semantics shown in this document are defined this way.

We can even create new notation for use in our proofs using mixfix syntax. Definitions appear in parentheses after the type signature and/or before the body of a function. For simple infix notation the *infixl* and *infixr* keywords are used, along with a numeric priority.

Datatypes Datatypes are defined using the *datatype* keyword with an ML-like syntax. These can also be parameterised.

Proofs A proof is started by stating the *theorem*, *lemma* or *schematic-goal* to be proven, optionally with a name. The proof can be completed in one either apply style or Isar style. Both languages allow the same tactics to be used, but the structure of the proof differs.

In apply style the proof is flat: subgoals are visible in the Isabelle/HOL proof window, but not the proof body. Methods are individually applied to modify or solve one or more subgoals. Methods are applied with the *apply* keyword and the proof is completed with *done*. The same method can be applied many times by appending *+*.

The preferred style is Isar: a language which embeds intermediate steps and facts in the proof body to create a literate proof. The keyword *proof* starts an Isar proof while the keyword *qed* finishes it.

In Isar, we "*have* facts *by* methods". The *also*, *moreover*, *hence*, *thus*, and *ultimately* keywords are used to chain facts and proofs.

The Isar tutorial [18] provides a simplified grammar, a sample proof skeleton, and an explanation of syntactic sugar.

Methods Isabelle proofs use a variety of methods to work towards the goal. Rules and elimination rules are applied via the keywords *rule* and *erule* respectively. For example, *rule exI* introduces the existential quantifier, while *erule exE* eliminates it.

The *auto* method combines classical reasoning with simplification to prove mostly-trivial subgoals, attempting to solve many. The *fastforce* method uses "many fancy proof tools to perform a rather exhaustive search" [18, p. 232]. Classical tableau solving is done via the *blast*, while first-order or mildly higher-order proofs can be found via the popular *metis* tactic.

3 The CoErl Language

CoErl is a fragment of Core Erlang for reasoning about Erlang's communication model. The language supports only sending, receiving, and sequences thereof. It does not express the higher-order or recursive functionality of Core Erlang.

The terminals of the language are integer and PID literals. The send, receive, and sequencing operations are represented as expressions.

type-synonym pid = nat

datatype terminal = Int int — An integer literal
 | Pid pid — A unique identifier for a process

datatype expr = Send pid terminal
 | Receive clause
 | Seq expr expr
 | Term terminal
and clause = Clause terminal expr
 | ClauseSeq clause clause

Receive expressions consist of a (non-empty) sequence of clauses which are processed in order during receive operations. *Send* and *Receive* are the communication primitives of the language. *Term* represents a *terminal* literal. Furthermore, this definition of *clause* forbids empty *Receive* statements.

We also define a function to iterate over our clauses in a manner similar to the *find* function found in other functional programming languages such as Haskell:

```
primrec find-clause :: (terminal ⇒ bool) ⇒ clause ⇒
   (terminal × expr) option
 where
   find-clause f (Clause t e) =
     (if f t then Some (t,e) else None) |
   find-clause f (ClauseSeq c1 c2) =
     (case find-clause f c1 of
      Some res ⇒ Some res
      |None ⇒ find-clause f c2)
```

3.1 Mailboxes

Mailboxes in CoErl are modelled minimally as a list of terms. We will ignore the BEAM's messaging optimisations which use multiple queues and pointers for the sake of simplicity.

```
type-synonym mailbox = terminal list
```

Then we model the clause selection process using our previously-defined *find-clause* function:

```
fun receive :: mailbox ⇒ clause ⇒
   (terminal × expr) option
 where
   receive [] c = None |
   receive (m#mb) c = (case find-clause (λp. p = m) c of
                      None ⇒ receive mb c
                      |res ⇒ res)
```

3.2 Processes

The final component of CoErl is processes. These are a simplification of Erlang's own, modelled simply as an expression and mailbox pair:

```
type-synonym process = expr × mailbox
```

4 Process-Local Semantics

We assume that *pop m xs* removes the first instance of *m* from mailbox *mb*:

```
definition pop :: 'a ⇒ 'a list ⇒ 'a list
 where pop ≡ List.remove1
```

Figure 1 shows the process-local small-step semantics of CoErl.

These model the reduction of expressions to terms using the mailbox as state: they do not model concurrent behaviour. As such, the SEND rule does not actually send a message to any other process. It simply reduces to the body of the sent message, as in Erlang.

To simplify the concurrent semantics in section 6, we have modelled RECEIVE operations atomically via the separate *select* operation (section 3.1). This prevents messages arriving in the process' mailbox *during* a receive operation, avoiding a race condition regarding clause selection.

In the BEAM, a different approach is taken. A receive operation essentially operates on a snapshot of the mailbox:

1. When a process attempts to retrieve a message from its mailbox (receive), it is marked as `receiving`. The mailbox can now be scanned to find matching messages.
2. Any arriving messages are held in a temporary, separate queue.
3. The process finishes scanning its mailbox, optionally selecting and removing a message. It is no longer marked as `receiving`.
4. The queued messages are appended to the process' mailbox.

The ARRIVE makes the process-local semantics non-syntax-directed: the rule applies at any time, regardless of expression or mailbox. This represents the non-deterministic behaviour of the concurrent language, but its use is highly restricted (section 6). The arrive rule necessarily introduces non-determinism to the semantics:

```
theorem process-local-nondeterm:
 ∃s s' s''. s → s' ∧ s → s'' ∧ s' ≠ s''
proof –
  have (Send ι t, []) → (Send ι t, [m]) by (simp add: Arrive)
  also have (Send ι t, []) → (Term t, []) by (rule Send)
  ultimately show ?thesis by blast
qed
```

Finally, sequencing requires two rules: SEQ reduces the left sub-expression, while SEQTERM discards the result of a fully-evaluated left sub-expression.

4.1 Reflexive Transitive Closure

The execution of a process can be described as the reflexive transitive closure of the process-local semantics. First, we define the following definition of *star* [19, p. 49]:

```
inductive
  star :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool
for r where
  refl: star r x x |
  step: r x y ⟹ star r y z ⟹ star r x z
```

Then we define the reflexive transitive closure of the relation from figure 1:

```
abbreviation process-local-steps :: process ⇒ process ⇒ bool
where x →* y ≡ star process-local x y
```

5 Labelled Transitions

Drawing inspiration from labelled transition systems, we define a set of observable actions for CoErl processes:

```
datatype action = SendA pid terminal — written ι ! t
  | ReceiveA terminal — written ? t
  | ArriveA terminal — written arr t
  | τ — the internal action
```

The actions describe specific communications behaviour:

SENDA A process is observed emitting (sending) a message. For example, sending the message *t* to process with pid *ι*: *ι* ! *t*.

RECEIVEA A process consumes a message from its own mailbox using the *pop* operation.

ARRIVEA A message is appended to a process' mailbox via some SENDA action elsewhere in the system. In the parlence of session types, arrival is the *dual* of sending, not receive.

τ The internal action. This is for any other local step the process takes; these are not related to communication.

$$\frac{}{(Send\ \iota\ t,\ mb) \rightarrow (Term\ t,\ mb)}\ \textsc{Send}$$

$$\frac{receive\ mb\ cs = Some\ (t,\ e) \land mb' = pop\ t\ mb}{(Receive\ cs,\ mb) \rightarrow (e,\ mb')}\ \textsc{Receive} \qquad \frac{mb' = mb\ @\ [t]}{(e,\ mb) \rightarrow (e,\ mb')}\ \textsc{Arrive}$$

$$\frac{(e_1,\ mb) \rightarrow ({e_1}',\ mb')}{(Seq\ e_1\ e_2,\ mb) \rightarrow (Seq\ {e_1}'\ e_2,\ mb')}\ \textsc{Seq} \qquad \frac{}{(Seq\ (Term\ t)\ e_2,\ mb) \rightarrow (e_2,\ mb)}\ \textsc{SeqTerm}$$

Figure 1. Process-local small-step semantics for CoErl

$$\frac{}{(Send\ \iota\ t,\ mb) \xrightarrow{\iota\ !\ t} (Term\ t,\ mb)}\ \textsc{Send}\iota \qquad \frac{receive\ mb\ cs = Some\ (t,\ e) \land mb' = pop\ t\ mb}{(Receive\ cs,\ mb) \xrightarrow{?\ t} (e,\ mb')}\ \textsc{Receive}\iota$$

$$\frac{mb' = mb\ @\ [t]}{(e,\ mb) \xrightarrow{arr\ t} (e,\ mb')}\ \textsc{Arrive}\iota \qquad \frac{(e_1,\ mb) \xrightarrow{\sigma} ({e_1}',\ mb')}{(Seq\ e_1\ e_2,\ mb) \xrightarrow{\sigma} (Seq\ {e_1}'\ e_2,\ mb')}\ \textsc{Seq}\iota$$

$$\frac{}{(Seq\ (Term\ t)\ e_2,\ mb) \xrightarrow{\tau} (e_2,\ mb)}\ \textsc{SeqTerm}\iota$$

Figure 2. Labelled transitions for CoErl processes

5.1 Process Transitions

Figure 2 shows the transition rules for a CoErl process. We annotate each rule from the process-local semantics in figure 1 with an action.

The Sendι, Receiveι, and Arriveι rules are analogous to their process-local counterparts. The SeqTermι is considered internal, while the Seqι rule inherits the sub-action σ from reducing the left-hand sub-expression.

Stepping through a process with these actions instead of directly stepping through the process-local semantics provides additional information:

1. Whether a process has performed a communications action, i.e. it is not internal (τ).
2. What communications action has occurred, the message contents, and the destinations of sent messages.

5.2 Strong Traces

The sequence of actions taken by a process is called a trace:

type-synonym trace = action list

A process can be traced by listing all actions it takes. To do this, we first add a list to our definition from section 4.1 to create a labelled reflexive transitive closure:

```
inductive label-star :: ('a ⇒ 'b ⇒ 'a ⇒ bool) ⇒
    'a ⇒ 'b list ⇒ 'a ⇒ bool
  for r where
    refl:
      label-star r x [] x |
    step:
      r x a y ⟹ label-star r y as z ⟹ label-star r x (a#as) z
```

This definition is then used to define the closure of the *transition* relation:

abbreviation transitions :: process ⇒ action list ⇒ process ⇒ bool
  where x −as→∗ y ≡ label-star transition x as y

such that process x transitions to y by the sequence of actions *as*.

$$\frac{}{p \overset{[]}{\rightsquigarrow} p}\ \textsc{ReflW} \qquad \frac{p \xrightarrow{\sigma} p' \land p' \overset{t}{\rightsquigarrow} p''}{p \overset{\sigma\ \cdot\ t}{\rightsquigarrow} p''}\ \textsc{StepW}$$

$$\frac{p \xrightarrow{\tau} p' \land p' \overset{t}{\rightsquigarrow} p''}{p \overset{t}{\rightsquigarrow} p''}\ \textsc{TauPW} \qquad \frac{p \overset{t}{\rightsquigarrow} p'}{p \overset{\tau\ \cdot\ t}{\rightsquigarrow} p'}\ \textsc{TauTW}$$

Figure 3. Weakly tracing CoErl processes

5.3 Weak Wraces

The *transitions* definition unnecessarily describes all internal actions taken by a process. From a communications perspective, these internal actions are unimportant.

To consider only communications in our transition system, we will weaken the *label-star* relation. The τ actions will be freely stepped over in both the trace and the process.

Figure 3 shows this weakened *transitions* relation. ReflW is identical to refl rule from *label-star* in section , as is StepW to step.

The TauPW and TauTW rules allow the process and trace respectively to freely step through the τ actions. As such, if $\tau \notin ts$, then t in $p \overset{t}{\rightsquigarrow} p''$ is the sequence of *communications actions* from process state p to p'.

As a sanity check, we ensure that any strong trace is also a weak trace using a short apply style proof:

theorem all-strong-traces-weaken:
  $p \xrightarrow{t} * p' \Longrightarrow p \overset{t}{\leadsto} p'$
 apply(induction rule: label-star.induct)
 using ReflW StepW **apply** auto
 done

5.4 Coverage of the Process-Local Semantics

As explained in section 1, we want our semantics to exhibit some useful "hygiene" properties.

Our transition system should be *sound* and *complete* with respect to the process-local semantics. In this context, the system is considered sound if only models behaviour of the process-local semantics, and the system is considered complete if all of the process-local semantics is modelled. These are the "hygiene" properties.

To show completeness it is sufficient to provide a witness action σ for every possible step in the process-local semantics. This is written in Isar style:

theorem transition-complete: $s \rightarrow s' \Longrightarrow \exists\,\sigma.\ (s \xrightarrow{\sigma} s')$
proof (induction rule: process-local.induct)
 case (Send ι t mb)
 hence (Send ι t, mb) $\xrightarrow{\iota\,!\,t}$ (Term t, mb)
  by (simp add: process-local.Send Sendι)
 thus ?case **by** (rule exI) — where exI: $P\,x \Longrightarrow \exists\,x.\ P\,x$
next
 case (Receive mb c t e mb$'$)
 hence (Receive c, mb) $\xrightarrow{?\,t}$ (e,mb$'$)
  using Receiveι **by** blast
 thus ?case **by** (rule exI)
next
 case (Arrive mb$'$ mb t e)
 hence (e, mb) $\xrightarrow{arr\ t}$ (e, mb$'$)
  by (simp add: process-local.Arrive Arriveι)
 thus ?case **by** (rule exI)
next
 case (Seq e_1 mb $e_1{}'$ mb$'$ e_2)
 then obtain σ' **where** (e_1, mb) $\xrightarrow{\sigma'}$ ($e_1{}'$, mb$'$)
  by blast
 hence (Seq e_1 e_2, mb) $\xrightarrow{\sigma'}$ (Seq $e_1{}'$ e_2, mb$'$)
  by (simp add: Seqι)
 thus ?case **by** (rule exI)
next
 case (SeqTerm t e_2 mb)
 hence (Seq (Term t) e_2, mb) $\xrightarrow{\tau}$ (e_2, mb)
  by (simp add: process-local.SeqTerm SeqTermι)
 thus ?case **by** (rule exI)
qed

The proof for the labelled reflexive transitive closure is quickly dispatched using apply style. Sledgehammer was used to prove some of the subgoals in this proof, as evidenced by the use of `metis`, the typical output produced by the tool (section 2.1):

theorem transitions-complete:
 $s \rightarrow* s' \Longrightarrow \exists\,t.\ (s \xrightarrow{t} * s')$
 apply(induction rule: star.induct)
  apply(rule exI[**where** x=[]])
  apply(rule label-star.refl)
 apply(metis label-star.step transition-complete)
 done

Soundness is proven just as quickly:

theorem proc-action-sound:
 $s \xrightarrow{\sigma} s' \Longrightarrow s \rightarrow s'$
 apply(induction rule: transition.induct)
  using process-local.intros **apply** simp+
 done

and also for the labelled reflexive transitive closure:

theorem proc-action-star-sound:
 $s \xrightarrow{t} * s' \Longrightarrow s \rightarrow* s'$
 apply(induction rule: label-star.induct)
  apply (metis proc-action-sound star.simps)+
 done

5.5 Trace Enumeration

All possible executions of a process can be considered by enumerating *all possible* traces of a process using a set comprehension:

definition enum-traces :: process ⇒ process ⇒ trace set
 where enum-traces p p$'$ = {t. $p \xrightarrow{t} * p'$}

A drawback is that this definition is not executable in Isabelle due to a limitation of set comprehensions: the actions in traces embed the non-enumerable int and nat types. The definition can be used in the context of formal reasoning, though.

6 Concurrent CoErl

A running Erlang node consists of multiple processes (with associated PIDs) running concurrently. In CoErl, we model nodes as a mapping from PIDs to processes:

type-synonym node = pid ⇀ process

Additionally, we define the || symbol to represent processes running concurrently:

fun par :: process × pid ⇒ node ⇒ node (**infixr** || 62)
 where $(p,\iota)\ ||\ \Pi = \Pi(\iota \mapsto p)$

We use $(p, \iota)\ ||\ \Pi$ to represent a node where process p with PID ι is running concurrently with some other processes Π.

The rules from our transition system in figure 2 are used to model execution on a CoErl node. Processes can make internal transitions freely via $\Pi\tau$) while communications operations are modelled via *meeting points*.

A sending process creates a meeting point with the destination process as follows:

ΠSend1 When a process sends a message to another process (i.e. $\iota' \neq \iota$), the two processes meet. Process ι transitions through its send while the destination process transitions through an arrival.

ΠSend2 The special case when a process sends a self addressed message. The process essentially creates a meeting point with itself. The sending of the message and the arrival of the message in the process' own mailbox is considered atomic.

$$\frac{p \xrightarrow{\iota' \,!\, t} p' \wedge q \xrightarrow{arr\ t} q'}{(p, \iota) \parallel (q, \iota') \parallel \Pi \xrightarrow[\iota]{\iota' \,!\, t} (p', \iota) \parallel (q', \iota') \parallel \Pi}\ \Pi\textsc{Send1} \qquad \frac{p \xrightarrow{\iota \,!\, t} p' \wedge p' \xrightarrow{arr\ t} p''}{(p, \iota) \parallel \Pi \xrightarrow[\iota]{\iota \,!\, t} (p'', \iota) \parallel \Pi}\ \Pi\textsc{Send2}$$

$$\frac{p \xrightarrow{\iota' \,!\, t} p' \wedge \iota' \neq \iota \wedge \iota' \notin dom\, \Pi}{(p, \iota) \parallel \Pi \xrightarrow[\iota]{\iota' \,!\, t} (p', \iota) \parallel \Pi}\ \Pi\textsc{Send3}$$

$$\frac{p \xrightarrow{?\, t} p'}{(p, \iota) \parallel \Pi \xrightarrow[\iota]{?\, t} (p', \iota) \parallel \Pi}\ \Pi\textsc{Receive} \qquad \frac{p \xrightarrow{\tau} p'}{(p, \iota) \parallel \Pi \xrightarrow[\iota]{\tau} (p', \iota) \parallel \Pi}\ \Pi\tau$$

Figure 4. a CoErl node using actions for messaging

ΠSend3 In Erlang a message can be sent to a PID which does not exist. The send operation provides no information to the sender[1]. Evaluation of the sending process continues and the message is discarded.

7 Examples

We can demonstrate the formal operation of a CoErl node in Isabelle. The **schematic-goal** command specifies our goal, which is then followed by an Isar proof. An illustrative diagram is provided with each example showing which process performs the transition. Note that the diagrams describe a single *atomic* action in the system.

- ΠSend1: Figure 5 shows the process ι sending the message t to a running process ι'. This composes the Sendι and Arriveι rules on processes ι and ι' respectively.
- ΠSend2: Figure 6 has the process ι with an empty mailbox sending the message t to itself. This uses the Sendι and Arriveι rules in sequence on process ι.
- ΠReceive: Figure 7 shows the process ι receiving the message t from its mailbox via clause selection.

8 Discussion

We have formalised a fragment of Core Erlang in Isabelle/HOL. Function definitions for mailbox operations and inductive definitions for process-local and concurrent semantics have been produced. The transition and trace systems have proven properties relating to soundness, completeness, and weakening.

A concurrent model of the language has also been built using this transition system, showing that it is capable of Erlang-like communication. Isabelle has quickly verified our theories: the sources were checked in an average of 13 seconds, using 600MB of memory on a laptop with an Intel i5-5200 CPU [2].

Some proofs in this document (e.g. those using the *metis* method) were automatically discovered via Isabelle's *sledgehammer* tool. The *try* keyword was used in the proof body to invoke a variety of external SMT solvers. The process can take several minutes and often fails, but a successful invocation typically yields a subgoal proof which executes in under a second.

[1]except when sending to registered (named) processes in the BEAM which do not exist, which will cause an error

[2]Isabelle's core theories (e.g. HOL, List, Monoid) are pre-built and cached, saving several minutes.

8.1 Nuances of Send

One liberty taken by our implementation of Core Erlang is that we have modelled `send (!)` *syntactically*, rather than as a BIF. In Erlang, send is a BIF named `erlang:send`. In Core Erlang, this is omitted: BIFs are left to the implementation. The omission however allows individual Erlang distributions to implement send behaviour in a domain-appropriate fashion.

For our model we defer to Erlang/OTP's implementation of `send`. We use the Send rule in figure 1 to reduce send expressions to body of the sent message, labelling this reduction via rule Sendι in figure 2. We ultimately use this labelled transition to implement the desired Erlang/OTP `send` behaviour in rules ΠSend1, ΠSend2, and ΠSend3 in figure 4.

8.2 Scope of the Model

CoErl currently only models the communication primitives of Erlang/OTP. To this end, the language does not *currently* feature function definitions, pattern matching, or single-assignment variables. These are a priority for future work as they are essential for bringing CoErl to feature parity with Core Erlang.

The language intentionally omits support for Core Erlang's annotation system. Partial support will be added for diagnostic purposes: we will be able to determine original source code locations using the generated line number annotations.

CoErl is intended for formal reasoning of Core Erlang specifically in the context of Erlang/OTP. It is not intended as a replacement or intermediate form Core Erlang.

One other notable omission is the lack of the `after` keyword in receive expressions: CoErl systems currently have no concept of time. The static analysis of current and future work however focuses on messages which will *never* be received. In general, an `after` clause leaves the possibility of reception.Another approach would be to adopt a timer model similar to those seen in multiparty session types [1].

8.3 Traces

Our trace system is capable of representing any program expressible in the CoErl language, but it is unlikely to be able to *efficiently* represent tail-recursive or higher-order programs. The system has some shortcomings which are better addressed by a CFSM-based system, which will be adopted in the future.

schematic-goal example-ΠSend1:

$((\text{Send } \iota' \text{ t,mb}),\iota) \parallel ((\text{e},[]),\iota') \parallel \Pi \xrightarrow[\iota]{\iota' ! t} ((\text{Term t,mb}),\iota) \parallel ((\text{e},[\text{t}]),\iota') \parallel \Pi$

proof –

have 1 to 2 for ι: (Send ι' t, mb) $\xrightarrow{\iota' ! t}$ (Term t, mb)
 by (simp add: Send Sendι)

also have 2 to 3 for ι': (e, []) $\xrightarrow{arr\ t}$ (e, [t])
 by (simp add: Arrive Arriveι)

ultimately show 1 to 3: ?thesis
 using ΠSend1 **by** simp

qed

(a) Isar proof

$$((Send\ \iota'\ t,\ mb),\ \iota)\ \parallel ((e,\ []),\ \iota')\ \parallel \Pi \quad (1)$$
$$\vdots$$
$$((Term\ t,\ mb),\ \iota)\ \parallel ((e,\ []),\ \iota')\ \parallel \Pi \quad (2)$$
$$\vdots$$
$$((Term\ t,\ mb),\ \iota)\ \parallel ((e,\ [t]),\ \iota')\ \parallel \Pi \quad (3)$$

(b) Diagram

Figure 5. Application of the ΠSend1 rule

schematic-goal example-ΠSend2:

$((\text{Send } \iota \text{ t},[]),\iota) \parallel \Pi \xrightarrow[\iota]{\iota ! t} ((\text{Term t},[\text{t}]),\iota) \parallel \Pi$

proof –

have 1 to 2 for ι: (Send ι t, []) $\xrightarrow{\iota ! t}$ (Term t, [])
 by (simp add: Send Sendι)

also have 2 to 3 for ι: (Term t, []) $\xrightarrow{arr\ t}$ (Term t, [t])
 by (simp add: Arrive Arriveι)

ultimately show 1 to 3: ?thesis
 using ΠSend2 **by** simp

qed

(a) Isar proof

$$((Send\ \iota\ t,\ []),\ \iota)\ \parallel \Pi \quad (1)$$
$$\vdots$$
$$((Term\ t,\ []),\ \iota)\ \parallel \Pi \quad (2)$$
$$\vdots$$
$$((Term\ t,\ [t]),\ \iota)\ \parallel \Pi \quad (3)$$

(b) Diagram

Figure 6. Application of the ΠSend2 rule

schematic-goal example-receive:

$((\text{Receive (Clause t e)},[\text{t}]),\iota) \parallel \Pi \xrightarrow[\iota]{?\ t} ((\text{e},[]),\iota) \parallel \Pi$

proof –

have 1 to 2 for ι: (Receive (Clause t e),[t]) $\xrightarrow{?\ t}$ (e,[])
 by (simp add: pop-def receive.cases Receive Receiveι)

thus 1 to 2: ?thesis
 using ΠReceive **by** simp

qed

(a) Isar proof

$$((Receive\ (Clause\ t\ e),\ [t]),\ \iota)\ \parallel \Pi \quad (1)$$
$$\vdots$$
$$((e,\ []),\ \iota)\ \parallel \Pi \quad (2)$$

(b) Diagram

Figure 7. Application of the ΠReceive rule

```
(Seq
  (Receive (CSeq
    (Clause 0 e0)
    (Clause 1 e1)))
  e2)
```

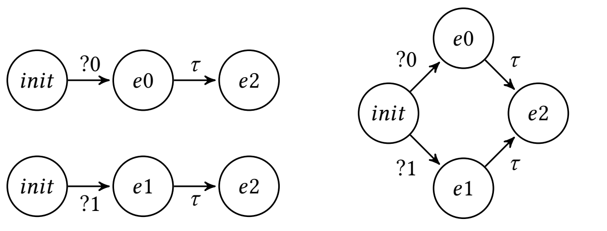
Figure 8. Receiving then sequencing in CoErl

Consider the CoErl expression in figure 8 which first receives 0 and evaluates $e0$, or receives 1 and evaluates $e1$, then via sequencing always evaluates $e2$.

Figure 9a shows a subset of the enumerated traces for this CoErl expression, as per the definition of $enum{-}traces$ in section 5.5. This produces two traces, one for each branch of the receive expression. Each time a receive statement with n clauses is encountered, the number of enumerated traces will multiply n-fold.

Figure 9b, however, shows the same expression represented as a labelled transition system. The receive operation introduces two new states and accompanying edges, but the eventual convergence to $e2$ is clear.

This is a shortcoming of traces: each one describes an execution of the process, while the LTS shows multiple execution paths simultaneously. The LTS can also clearly encode recursive behaviour by adding edges to create a cyclic graph: list-based traces cannot express this potentially infinite behaviour concretely.



(a) Enumerated traces **(b)** Labelled transition system

Figure 9. Trace and LTS representations of figure 8

9 Related Work

π-calculus is another method of modelling the operational semantics of concurrent systems. Many principles of π-calculus apply to Erlang: both use processes as the concurrent primitive, the processes are sequential, and data is transferred via messages [14].

Unfortunately, other aspects are incompatible: the messaging model of the calculus *synchronous* and *ordered*, unlike Erlang's *asynchronous* and *out-of-order* model. Additionally, channels in π-calculus can be delegated, unlike Erlang's mailboxes, which cannot. Despite these incompatibilities, there has been work to model Erlang using a π-calculus [20].

The CoErl model is also similar to the CSP model [11]: processes in both systems are again sequential, these processes also serve as the concurrency primitive, and like the CoErl node model, concurrent behaviour relies on a rendezvous or "meeting point".

9.1 Static Analysis

Regardless of the underlying calculi, there has been extensive work on static analysis of Erlang's communication model. One promising area of research uses *session types* to describe communication protocols for an entire Erlang system [15]. There is also an implementation of *multiparty session actors* [17] in Erlang, using the Scribble protocol language [25]. The system has motivating practical examples [8].

Session types provide a good formal model for communicating systems, but they typically require a *global* session type which describes *all* communication in the specified system. Creating these specifications may be possible for a new system, but it seems impractical to fully specify the communications of a legacy system.

Dialyzer's [13] analysis output has been used to detect race conditions in Erlang. By using Dialyzer's automatic (and optionally directed) analysis, race conditions can be detected in legacy systems [5]. The benefits are immediately obvious: a system can be specified and analysed *incrementally*.

These static analyses all share one weakness due to one of Erlang's strengths: code replacement. The code server in Erlang/OTP allows modules to be replaced at runtime [7], but these analyses are generally performed before or at compile-time. Reloading a module at runtime can invalidate these analyses.

A similar formalisation of Core Erlang called λActor is used for abstract interpretation of Erlang programs, using a vector addition system. The described Soter tool analyses programs in a matter of seconds [6].

As an alternative approach, McErlang serves as an alternative runtime for Erlang systems. It is a model checker offering easy access to the program state [10].

9.2 Runtime Monitoring

Runtime monitoring is an alternative approach to static analysis. Source code is analysed to generate *monitors* which observe and verify communications at runtime. The detectEr tool produces instrumented code for synchronously and asynchronously monitoring Erlang processes for communication violations. In exchange for closing the code replacement hole, this analysis suffers from runtime overhead [4, 9].

The Scribble protocol language has also been integrated with the `gen_supervisor` behaviour to generate optimised recovery strategies for use at runtime [16].

10 Conclusions and Future Work

It has been practical to model parts of Core Erlang in Isabelle. The definitions are executable, checked, and can be translated into a variety of functional programming languages. Our fragment of Core Erlang models the communication behaviour of an Erlang node, but we do not yet support pattern matching, recursion, or higher-order behaviour. To this end, current work is focusing on their implementation.

Future work will abandon trace enumeration in favour of the graphs seen in labelled transition systems. This will allow us to reason about branching and recursive behaviour more easily. From here, we will move onto the main goal of the broader work: detecting orphan messages. As process evaluation is non-deterministic, we will also prove convergence and congruence properties of the language as it grows.

Ultimately, we want to produce a *useful* development tool for Erlang with a simple *command line* interface. Isabelle will allow us to generate code for this tool and we can use it to develop our model.

Acknowledgments

This work is supported by the Engineering and Physical Sciences Research Council.

References

[1] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. 2014. Timed Multiparty Session Types. In *International Conference on Concurrency Theory (CONCUR) (Lecture Notes in Computer Science)*, Paolo Baldan and Daniele Gorla (Eds.). Springer, 419–434. http://kar.kent.ac.uk/43729/
[2] Richard Carlsson. 2001. An introduction to Core Erlang. In *In Proceedings of the PLI'01 Erlang Workshop*.
[3] Richard Carlsson, Björn Gustavsson, Erik Johansson, et al. 2000. Core Erlang 1.0.3 language specification. *Information Technology Department, Uppsala University, Tech. Rep* (2000). https://it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf
[4] Ian Cassar and Adrian Francalanza. 2015. On Synchronous and Asynchronous Monitor Instrumentation for Actor-based systems. *Electronic Proceedings in Theoretical Computer Science* 175 (feb 2015), 54–68. https://doi.org/10.4204/eptcs.175.4
[5] Maria Christakis and Konstantinos Sagonas. 2010. Static Detection of Race Conditions in Erlang. In *Practical Aspects of Declarative Languages*. Springer Berlin Heidelberg, 119–133. https://doi.org/10.1007/978-3-642-11503-5_11
[6] Emanuele D'Osualdo, Jonathan Kochems, and C. H. Luke Ong. 2013. Automatic Verification of Erlang-Style Concurrency. In *Static Analysis*. Springer Berlin Heidelberg, 454–476. https://doi.org/10.1007/978-3-642-38856-9_24
[7] Ericsson AB. 2017. *Erlang Reference Manual User's Guide* (8.3 ed.). Chapter 14. Compilation and Code Loading. http://erlang.org/doc/reference_manual/code_loading.html
[8] Simon Fowler. 2016. An Erlang Implementation of Multiparty Session Actors. *Electronic Proceedings in Theoretical Computer Science* 223 (aug 2016), 36–50. https://doi.org/10.4204/eptcs.223.3
[9] Adrian Francalanza and Aldrin Seychell. 2014. Synthesising correct concurrent runtime monitors. *Formal Methods in System Design* 46, 3 (nov 2014), 226–261. https://doi.org/10.1007/s10703-014-0217-9

[10] Lars-Åke Fredlund and Hans Svensson. 2007. McErlang: A Model Checker for a Distributed Functional Programming Language. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. ACM, New York, NY, USA, 125–136. https://doi.org/10.1145/1291151.1291171

[11] C. A. R. Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (aug 1978), 666–677. https://doi.org/10.1145/359576.359585

[12] Gerwin Klein, Michael Norrish, Thomas Sewell, et al. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*. ACM Press. https://doi.org/10.1145/1629575.1629596

[13] Tobias Lindahl, Konstantinos Sagonas, Daniel Luna, and Maria Christakis. 2004. Dialyzer. (2004). https://www.it.uu.se/research/group/hipe/dialyzer

[14] Robin Milner. 1999. *Communicating and Mobile Systems*. Cambridge University Press.

[15] Dimitris Mostrous and Vasco Vasconcelos. 2011. Session Typing for a Featherweight Erlang. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 95–109. https://doi.org/10.1007/978-3-642-21464-6_7

[16] Rumyana Neykova and Nobuko Yoshida. 2017. Let It Recover: Multiparty Protocol-Induced Recovery. In *26th International Conference on Compiler Construction*. ACM, 98–108. https://doi.org/10.1145/3033019.3033031

[17] Rumyana Neykova and Nobuko Yoshida. 2017. Multiparty Session Actors. *Logical Methods in Computer Science* 13, 1 (2017). https://doi.org/10.23638/LMCS-13(1:17)2017

[18] Tobias Nipkow. 2016. A Tutorial Introduction to Structured Isar Proofs. (2016). http://www.it.uu.se/grad/courses/gc0910/isabelle/isar-overview.pdf

[19] Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics*. Springer International Publishing.

[20] Thomas Noll and Chanchal Kumar Roy. 2005. Modeling Erlang in the pi-calculus. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang - ERLANG '05*. ACM Press. https://doi.org/10.1145/1088361.1088375

[21] Kostis Sagonas, Mikael Pettersson, Jesper Wilhelmsson, Tobias Lindahl, Richard Carlsson, and Per Gustafsson. 1998. The High-Performance Erlang Project. (1998). https://www.it.uu.se/research/group/hipe/

[22] Makarius Wenzel. 2007. Isabelle/Isar - a generic framework for human-readable proof documents.

[23] Makarius Wenzel et al. 2016. Isabelle. (2016). http://isabelle.in.tum.de/index.html

[24] Makarius Wenzel et al. 2016. The Isabelle/Isar Reference Manual. (12 2016). http://isabelle.in.tum.de/doc/isar-ref.pdf

[25] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2014. The Scribble Protocol Language. In *Trustworthy Global Computing*. Springer International Publishing, 22–41. https://doi.org/10.1007/978-3-319-05119-2_3