Published: 09 October 2025

Citation in BibTeX format

Erlang '25: 24th ACM SIGPLAN International Workshop on Erlang
October 12 - 18, 2025
Singapore, Singapore

Conference Sponsors:
SIGPLAN

RESEARCH-ARTICLE

Deriving an Erlang Interpreter from a Mechanised Formal Semantics of Core Erlang

GERGŐ LAJOS TURÁN, Eötvös Loránd University, Budapest, Hungary

ARSENII FOMIN, Eötvös Loránd University, Budapest, Hungary

PÉTER BERECZKY, Eötvös Loránd University, Budapest, Hungary

DÁNIEL HORPÁCSI, Eötvös Loránd University, Budapest, Hungary

SIMON THOMPSON, University of Kent, Canterbury, Kent, U.K.

Open Access Support provided by:

Eötvös Loránd University

University of Kent

Deriving an Erlang Interpreter from a Mechanised Formal Semantics of Core Erlang

Gergő Lajos Turán
Eötvös Loránd University
Budapest, Hungary
mtlevr@inf.elte.hu

Arsenii Fomin
Eötvös Loránd University
Budapest, Hungary
nbtsvl@inf.elte.hu

Péter Bereczky
Eötvös Loránd University
Budapest, Hungary
berpeti@inf.elte.hu

Dániel Horpácsi
Eötvös Loránd University
Budapest, Hungary
daniel-h@elte.hu

Simon Thompson
University of Kent
Canterbury, United Kingdom
s.j.thompson@kent.ac.uk

Abstract

We present an interpreter for Erlang in Haskell, derived from a formal semantics for Core Erlang mechanised in Coq. The interpreter function is derived from the Coq inductive definitions that make up the semantics by extracting Haskell code from Gallina functions provably equivalent to the inductive definitions, and optimising the result. The semantics is inherently non-deterministic, and it is made deterministic by introducing a scheduler component; we also present a computation graph that shows all the non-deterministic choices that arise during computation. The paper concludes with an evaluation of the work and preliminary performance data.

CCS Concepts: **• Theory of computation → Operational semantics**; **Functional constructs**; **Concurrency**; **• Software and its engineering → Interpreters**.

Keywords: Erlang, Core Erlang, Formal semantics, Formally based interpreter, Scheduling, Coq, Rocq, Haskell

ACM Reference Format:
Gergő Lajos Turán, Arsenii Fomin, Péter Bereczky, Dániel Horpácsi, and Simon Thompson. 2025. Deriving an Erlang Interpreter from a Mechanised Formal Semantics of Core Erlang. In *Proceedings of the 24th ACM SIGPLAN International Workshop on Erlang (Erlang '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3759161.3763046

1 Introduction

Programming language implementations are crucial pieces of computing infrastructure, and their correct behaviour underpins any argument about the trustworthiness of any particular program. Compilers and interpreters are subject to thorough testing in practice, but testing alone is *post hoc* and necessarily incomplete. An alternative is offered by verification of an implementation against a formal[1] semantics for the language, and a tractable mechanism to do this is to *derive* an implementation from such a semantics. This paper presents an interpreter for Erlang developed in that way,[2] and can be seen as a *reference interpreter* that gives a formal standard an executable form. This allows potential changes to the language—including Erlang Enhancement Proposals[3]—to be modelled formally and tried out before being accepted.

We build on a formal semantics for Core Erlang [3–6] mechanised in the Coq proof assistant,[4] deriving a Haskell interpreter from the semantics. The mechanised semantics takes the form of a series of inductive definitions of reduction relations over configurations that represent the state of a node as a set of concurrent processes and a communication 'ether'. The first stage of the extraction is to define a set of total functions in Coq that implement single reduction steps in the semantics, and to prove their equivalence to the relational semantics. The functions can then automatically be extracted into equivalent Haskell definitions using existing tooling within Coq [11], and then be optimised, replacing data type definitions by efficient equivalents, and making some operations (such as substitution) strict instead of lazy. To interpret the full Erlang language we translate it to Core Erlang using the standard Erlang compiler.

The semantics of Erlang is inherently non-deterministic, and it is made deterministic by introducing a parametric scheduler component that ensures that all processes are able to progress during evaluation, with the specific choice being provided by an instance of a scheduler typeclass. Complementing this we also design and prototype a computation graph that shows all the choices that arise during the course of a computation.

[1]We use the term "formal" for the mathematical semantics, and "mechanised" for the embodiment of that semantics in Coq.
[2]The interpreter is available on GitHub [21].
[3]https://www.erlang.org/eep
[4]Coq was renamed to Rocq in 2025; however, this project still uses version 8.20 of the proof assistant, which is called Coq.

What is the assurance argument for an interpreter extracted in this way? The computation rules that it uses are guaranteed to be correct by the construction process that has been used, and so overall correctness depends only on the correctness of the code extraction process, the optimisations made, and the translation function from Erlang to Core Erlang, each of which is considerably simpler than writing a complete interpreter. Directly executing the rules of the formal semantics comes at a performance cost, but this is a reasonable trade-off for the correctness guarantees it gives.

The rest of the paper is structured as follows. Section 2 summarises the semantics of Core Erlang on which we build our interpreter. Our formally based interpreter is presented in Section 3, and the prototype computation graph regarding the non-deterministic decisions made during program evaluation is discussed in Section 4. Section 5 presents an evaluation of the work. Section 6 summarises related work, and Section 7 concludes and identifies future directions.

2 Mechanised Formal Semantics for Erlang

"Formal semantics models the computational meaning of programs", utilising mathematically precise and rigorous descriptions [33]. There is a body of work that describes a series of semantics for Erlang, formalised in the Coq proof assistant (in big-step [3, 5] and in frame stack definition styles [4, 6]). In this work, we build on the frame stack-style definition because it also addresses concurrency.

2.1 Overview of the Formal Operational Semantics

Table 1. Layers of the semantics

Layer name	Notation	Description
Inter-process	$\xrightarrow{\iota:a}$	System-level reductions
Process-local	$\xrightarrow{a}$	Process-level reductions
Sequential	$\longrightarrow$	Computational reductions

The frame stack semantics consists of three layers. Reductions denoted with $\longrightarrow$ are computational steps done by the sequential semantics. Reductions denoted with $p \xrightarrow{a} p'$ are the process-local steps, which involve one single process and coordinate with the inter-process semantics with semantic actions (a). The inter-process semantics (denoted with $N \xrightarrow{\iota:a} N'$) describes how communication is carried out among processes inside a node. Here, we provide a brief overview of each layer, and refer to [4, 6] for further details.

Abstract Syntax of Core Erlang. A Core Erlang expression (Figure 1) is either a value or a non-value. The following expressions represent values: integers (denoted i), atoms (a), variables (x), function identifiers (f/n), process identifiers (PIDs, ι), function closures ($clos$), empty and nonempty lists (square brackets), tuples (braces), and maps (tilde-braces).

$$
\begin{aligned}
p \in \mathit{Pattern} &::= i \mid a \mid x \mid \texttt{[]} \mid \texttt{[}p_1\texttt{|}p_2\texttt{]} \mid \{p_1,\dots,p_n\} \\
&\mid \sim\{p_1^k \Rightarrow p_1^v,\dots,p_n^k \Rightarrow p_n^v\}\sim \\
ps \in \mathit{PatternList} &::= \texttt{<}p_1,\dots,p_n\texttt{>} \\
cli \in \mathit{ClosItem} &::= f/n = \texttt{fun}(x_1,\dots,x_n) \rightarrow e \\
ext \in \mathit{ClosItemList} &::= cli_1,\dots,cli_n \\
cl \in \mathit{Clause} &::= ps\ \texttt{when}\ e^g \rightarrow e^b \\
v \in \mathit{Val} &::= i \mid a \mid x \mid f/n \mid \iota \mid clos(ext,[x_1,\dots,x_n],e) \\
&\mid \texttt{[}v_1\texttt{|}v_2\texttt{]} \mid \texttt{[]} \mid \{v_1,\dots,v_n\} \mid \sim\{v_1^k \Rightarrow v_1^v,\dots,v_n^k \Rightarrow v_n^v\}\sim \\
nv \in \mathit{NonVal} &::= \texttt{fun}(x_1,\dots,x_n) \rightarrow e \mid \texttt{<}e_1,\dots,e_n\texttt{>} \mid \texttt{[}e_1\texttt{|}e_2\texttt{]} \\
&\mid \{e_1,\dots,e_n\} \mid \sim\{e_1^k \Rightarrow e_1^v,\dots,e_n^k \Rightarrow e_n^v\}\sim \\
&\mid \texttt{call}\ e^m\texttt{:}e^f(e_1,\dots,e_n) \mid \texttt{primop}\ a(e_1,\dots,e_n) \\
&\mid \texttt{apply}\ e(e_1,\dots,e_n) \mid \texttt{case}\ e_1\ \texttt{of}\ cl_1;\dots;cl_n\ \texttt{end} \\
&\mid \texttt{let}\ \texttt{<}x_1,\dots,x_n\texttt{>} = e_1\ \texttt{in}\ e_2 \mid \texttt{do}\ e_1\ e_2 \mid \texttt{letrec}\ ext\ \texttt{in}\ e \\
&\mid \texttt{try}\ e_1\ \texttt{of}\ \texttt{<}x_1,\dots,x_n\texttt{>} \rightarrow e_2\ \texttt{catch}\ \texttt{<}x_{n+1},\dots,x_{n+m}\texttt{>} \rightarrow e_3 \\
e \in \mathit{Exp} &::= nv \mid v
\end{aligned}
$$

Figure 1. Mutually defined syntax of Core Erlang

Core Erlang has Erlang-like expressions, such as `fun`, application, pattern matching with `case` and exception handling with `try`. Moreover, Core Erlang also features `let` and `letrec` binders, sequencing (`do`) expressions, and there are primitive operations (`primop`) expressing low-level behaviour, such as mailbox operations.

Free variables (and function identifiers) can be substituted. Substitutions are denoted by $e[x_1 \mapsto v_1,\dots,x_n \mapsto v_n]$, meaning that variables $x_1,\dots,x_n$ are syntactically replaced by values $v_1,\dots,v_n$ in the expression e.

In Figure 2 we present a running example, which consists of two variants of mapping a function over the list elements in Core Erlang: `'map'/2` is the textbook definition of this operation, while `'pmap'/3` delegates some computation (specified by the index `I`) to a child process. Note that the `receive` expression shown in the figure is in fact implemented as a combination of primitive operations as of OTP version 23.0 [19], which is also reflected in the formal semantics [6] we use in this work.

Sequential Frame Stack Semantics. The sequential semantics of Core Erlang defines how a process executes computational steps, expressing how a single expression evaluates, independent of the process state. The evaluation result is either a sequence of values (*value sequence*) or an exception (represented with a triple of an exception class atom, and two values about the exception's reason and details). Value sequences are used to express simultaneous variable bindings in `let`, `try`, `case` expressions, and only value list expressions ($\texttt{<}e_1,\dots,e_n\texttt{>}$) evaluate to non-singleton value sequences.

```
'map'/2 = fun(F, L) -> case L of
   <[]> when 'true' -> []
   <[H|T]> when 'true' ->
     [apply F(H) | apply 'map'/2(F, T)]
  end

'pmap'/3 = fun(F, I, L) ->
    case call 'lists':'split'(I, L) of
     <{L1, L2}> when 'true' ->
      let <S> = call 'erlang':'self'() in
       do call 'erlang':'spawn'(fun() ->
            call 'erlang':'!'(S, apply 'map'/2(F,L1)))
          let <M2> = apply 'map'/2(F,L2) in
          receive <M1> when 'true' ->
            call 'erlang':'++'(M1, M2)
          after 'infinity' -> []
    end
```

Figure 2. Sequential and concurrent list transformation

We also introduce redexes, which are either expressions, exceptions, value sequences, or the box (□) that is used in the semantics of empty parameter lists.

$$exc \in Exception ::= \{a, v^r, v^d\}^X$$
$$r \in Redex ::= e \mid exc \mid <v_1, \ldots, v_n> \mid \Box$$

The frame stack semantics [35] is reduction-style [13]: it represents the reduction context as a stack (K) of elementary frames (F); we use the infix notation $F :: K$ to put a frame onto the stack. These frames are essentially non-values with one of their subexpressions replaced by □, which denotes the position of the subexpression being currently evaluated.

We do not quote the entire semantics definition of Core Erlang [4] in this paper, but we show a representative[5] example recalling the reduction rules for `try` redexes (Figure 3). First, STry is used to evaluate the head expression of `try` to a value sequence or an exception. If the result is a value sequence, PTry is used to evaluate the first clause of `try` after substituting the results to the given variables. If the head expression evaluated to an exception, ExcTry is used to evaluate the `catch` clause, after substituting the exception to the specified variables. Finally, ExcProp expresses exception propagation. For the complete definition, see [4] and [21].

Process-local Semantics. The process-local semantics extends the sequential configuration into a process state. A live process is represented by a quintuple (K, r, q, L, b) where K and r denote a sequential configuration (a frame stack and a redex), q is the message queue (mailbox), L denotes the set of linked process identifiers, and b is the status of the `'trap_exit'` process flag. Dead processes (denoted by T) map linked PIDs to values to be sent to the linked processes.

The process-local steps are annotated by concurrent actions to emit side effects (which are used to coordinate with the inter-process semantics); these actions are the following:

- $send(\iota^s, \iota^d, s)$ the signal s is sent from the process identified by ι^s to the process with PID ι^d.
- $arr(\iota^s, \iota^d, s)$ the signal s arrived at the process identified by ι^d, sent from the process with PID ι^s.
- $self(\iota)$ is used to retrieve a process's own identifier from the inter-process semantics.
- τ denotes silent reduction steps, such as steps of the sequential semantics and message receipts.
- $spawn(\iota, v^f, v_1, \ldots, v_k, b)$ spawns a process that evaluates v^f applied to the arguments $v_1, \ldots, v_k$; the flag b says whether the process is to be linked to its parent.

We present a few rules from the semantics on Figure 4. Seq lifts the sequential semantics to the process level. Exit describes sending an exit signal by calling `exit/2`. ExitDrop, ExitTerm, and ExitConv handle the various cases of *exit* signal arrival. Dead shows a dead process emitting an *exit* signal (chosen non-deterministically) to one of its links. For the rest of the process-local rules, see [6].[6]

The process-local semantics is nondeterministic; however, in any state a live process can only reduce in two different ways: either a computational step is made (e.g., Seq, Exit), or a signal arrives at the process (e.g., ExitDrop, ExitTerm, ExitConv). Dead process evaluation is non-deterministic too, due to the lack of ordering of the linked PIDs.

Inter-process Semantics. An Erlang node consists of a number of processes that run concurrently, and communicate by signals. Inter-process semantics describes this communication, formalising *messages*, *exit* and *(un-)link* signals.

Faithful to the language documentation, signal passing is not atomic: sent but undelivered signals reside in an ether (in order of arrival, to comply with the "signal ordering guarantee"); therefore, an Erlang node is formalised as a pair: $(\mathcal{E}, \Pi)$, where $\mathcal{E}$ denotes the ether and Π denotes a process pool. We use the following operations for pools and ethers:

- $\iota : p \parallel \Pi$ adds process p identified by ι to the pool Π;
- $\mathcal{E}[(\iota^s, \iota^d) \mapsto^+ s]$ adds the signal s to the ether $\mathcal{E}$ with source ι^s and destination ι^d;
- $\mathcal{E} \setminus_{1st} (\iota^s, \iota^d)$ removes the first signal from the ether which targets ι^d originating from ι^s;
- $PIDs(\Pi)$ (and $PIDs(\mathcal{E})$) denote all PIDs that syntactically occur in a pool Π (or ether $\mathcal{E}$).

Inter-process reductions are annotated by $\iota : a$, which means that the process identified by ι makes a process-local reduction step based on action a. The entire semantics consists of only 4 rules, which we present in Figure 5. Essentially, all 4 rules propagate a reduction step to the process-local semantics, while making adjustments to the node's state.

[5]We quote rules that show the principles behind frame stack semantics, as well as highlight how the substitution operation appears in the semantics.

[6]The formalisation does not include `receive` timeouts in full generality, only `0` and `'infinity'` are modelled.

$$\langle K, \texttt{try}\ e_1\ \texttt{of}\ \texttt{<}x_1,\dots,x_n\texttt{>} \to e_2\ \texttt{catch}\ \texttt{<}x_{n+1},\dots,x_{n+k}\texttt{>} \to e_3\rangle \longrightarrow$$
$$\langle \texttt{try}\ \square\ \texttt{of}\ \texttt{<}x_1,\dots,x_n\texttt{>} \to e_2\ \texttt{catch}\ \texttt{<}x_{n+1},\dots,x_{n+k}\texttt{>} \to e_3 :: K, e_1\rangle \qquad (\textsc{STry})$$

$$\langle \texttt{try}\ \square\ \texttt{of}\ \texttt{<}x_1,\dots,x_n\texttt{>} \to e_2\ \texttt{catch}\ \texttt{<}x_{n+1},\dots,x_{n+k}\texttt{>} \to e_3 :: K, \texttt{<}v_1,\dots,v_n\texttt{>}\rangle \longrightarrow \langle K, e_2[x_1 \mapsto v_1,\dots,x_n \mapsto v_n]\rangle \qquad (\textsc{PTry})$$

$$\langle \texttt{try}\ \square\ \texttt{of}\ \texttt{<}x_1,\dots,x_n\texttt{>} \to e_2\ \texttt{catch}\ \texttt{<}x_{n+1},\dots,x_{n+3}\texttt{>} \to e_3 :: K, \{c, v^r, v^d\}^X\rangle \longrightarrow \langle K, e_3[x_{n+1} \mapsto c, x_{n+2} \mapsto v^r, x_{n+3} \mapsto v^d]\rangle \qquad (\textsc{ExcTry})$$

$$\langle F :: K, \{c, v^r, v^d\}^X\rangle \longrightarrow \langle K, \{c, v^r, v^d\}^X\rangle \qquad (\text{if } F \neq \texttt{try}\ \square\ \texttt{of}\ \ \dots) \qquad (\textsc{ExcProp})$$

Figure 3. Sequential semantics of exception handling

$$\frac{\langle K, r\rangle \longrightarrow \langle K', r'\rangle}{(K, r, q, L, b) \xrightarrow{\tau} (K', r', q, L, b)} \qquad (\textsc{Seq})$$

$$(\mathit{call}(\texttt{'erlang'}, \texttt{'exit'})(\iota^d, \square) :: K, \texttt{<}v\texttt{>}, q, L, b)$$
$$\xrightarrow{\mathit{send}(\iota^s,\iota^d,\mathit{exit}(v,\mathit{false}))} (K, \texttt{<'true'>}, q, L, b) \qquad (\textsc{Exit})$$

$$\frac{\iota^s \neq \iota^d \land ((b = \mathit{false} \land v = \texttt{'normal'}) \lor (\iota^s \notin L \land b^e = \mathit{true}))}{(K, r, q, L, b) \xrightarrow{\mathit{arr}(\iota^s,\iota^d,\mathit{exit}(v,b^e))} (K, r, q, L, b)} \qquad (\textsc{ExitDrop})$$

$$\frac{\begin{gathered}(v = \texttt{'kill'} \land b^e = \mathit{false} \land v' = \texttt{'killed'}) \lor \\ (b = \mathit{false} \land v = \texttt{'normal'} = v' \land \iota^s = \iota^d) \lor \\ (b = \mathit{false} \land v \neq \texttt{'normal'} \land v' = v \land \\ (b^e = \mathit{true} \to \iota^s \in L) \land (b^e = \mathit{false} \to v \neq \texttt{'kill'}))\end{gathered}}{(K, r, q, L, b) \xrightarrow{\mathit{arr}(\iota^s,\iota^d,\mathit{exit}(v,b^e))} (\lambda\iota \Rightarrow (\iota, v')) \texttt{<\$>}\ L} \qquad (\textsc{ExitTerm})$$

$$\frac{b = \mathit{true} \land ((b^e = \mathit{false} \land v \neq \texttt{'kill'}) \lor (b^e = \mathit{true} \land \iota^s \in L))}{(K, r, q, L, b) \xrightarrow{\mathit{arr}(\iota^s,\iota^d,\mathit{exit}(v,b^e))} (K, r, \mathit{push}(q, \{\texttt{'EXIT'}, \iota^s, v\}), L, b)} \qquad (\textsc{ExitConv})$$

$$\frac{T[\iota^d] = \mathit{Some}(v)}{T \xrightarrow{\mathit{send}(\iota^s,\iota^d,\mathit{exit}(v,\mathit{true}))} T \setminus \{\iota^d\}} \qquad (\textsc{Dead})$$

We used two auxiliary notations: $\lambda\iota \Rightarrow (\iota, v')$ <\$> L transforms the set of links L into a finite map by associating the PIDs in L with the value v'. $T[\iota]$ denotes the reason value associated with ι in the dead process T.

Figure 4. Selected rules of the process-local semantics

NSend adds the signal sent by process with PID ι^s to the ether $\mathcal{E}$. NArrive removes a signal from the ether, and forwards it to process with PID ι^d. NLocal makes an internal computation step (or evaluates a `self` call) with process identified by ι. Finally, NSpawn spawns a new process with an unused PID ι_2. This step is a side effect of evaluating a `spawn` (or `spawn_link`, denoted by the flag b) call in the process ι_1. The new process's initial configuration consists of an empty frame stack, mailbox, set of links, the `trap_exit` flag is set to false, and the expression to evaluate is the result of applying the closure v_f to the arguments $v_1, \dots, v_k$. Both the closure and the arguments are communicated from the process-local semantics: they appear inside the *spawn* action.

We note that this layer of the semantics is also non-deterministic as it captures all possible interleavings of processes. Technically, any process can be selected from a process pool (e.g., by a scheduler) for making a reduction step at any node state. Program execution is essentially a sequence of such reduction steps.

This summary hides several details of the semantics: for the full formalisation, see [6] and the implementation [21].

$$\frac{p \xrightarrow{\mathit{send}(\iota^s,\iota^d,s)} p'}{(\mathcal{E}, \iota^s : p \parallel \Pi) \xrightarrow{\iota^s:\mathit{send}(\iota^s,\iota^d,s)} (\mathcal{E}[(\iota^s, \iota^d) \mapsto^+ s], \iota^s : p' \parallel \Pi)} \qquad (\textsc{NSend})$$

$$\frac{p \xrightarrow{\mathit{arr}(\iota^s,\iota^d,s)} p' \qquad \mathcal{E} \setminus_{1st} (\iota^s, \iota^d) = \mathit{Some}(s, \mathcal{E}')}{(\mathcal{E}, \iota^d : p \parallel \Pi) \xrightarrow{\iota^d:\mathit{arr}(\iota^s,\iota^d,s)} (\mathcal{E}', \iota^d : p' \parallel \Pi)} \qquad (\textsc{NArrive})$$

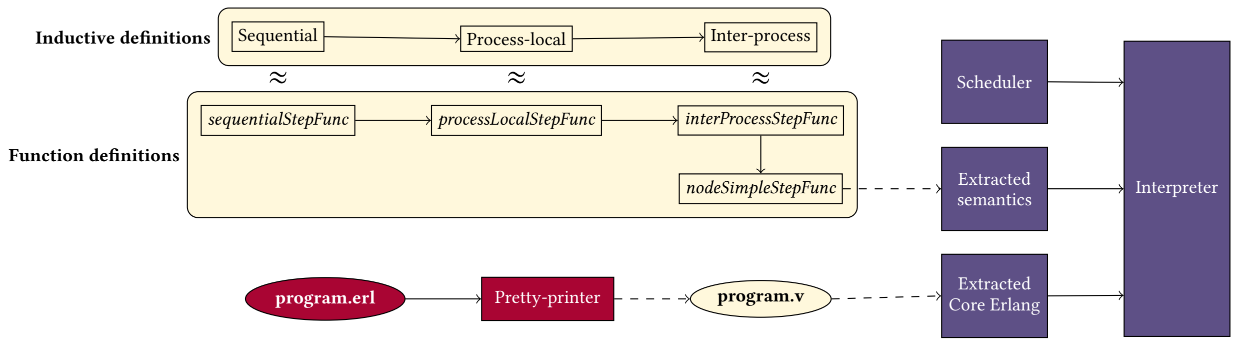
$$\frac{p \xrightarrow{\alpha} p' \qquad \alpha \in \{\mathit{self}(\iota), \tau\}}{(\mathcal{E}, \iota : p \parallel \Pi) \xrightarrow{\iota:\alpha} (\mathcal{E}, \iota : p' \parallel \Pi)} \qquad (\textsc{NLocal})$$

$$\frac{\begin{gathered}p \xrightarrow{\mathit{spawn}(\iota_2,v^f,v^l,b)} p' \qquad L = \mathit{if}\, b\ \mathit{then}\ \{\iota_1\}\ \mathit{else}\ \emptyset \\ \iota_2 \notin O \cup \mathit{PIDs}(\mathcal{E}) \cup \mathit{PIDs}(\iota_1 : p \parallel \Pi) \qquad r = \texttt{apply}\ v^f(v_1, \dots, v_k)\end{gathered}}{\begin{gathered}(\mathcal{E}, \iota_1 : p \parallel \Pi) \xrightarrow{\iota_1:\mathit{spawn}(\iota_2,v^f,\texttt{[}v_1,\dots,v_k\texttt{]},b)} \\ (\mathcal{E}, \iota_2 : (\varepsilon, r, \emptyset, L, \mathit{false}) \parallel \iota_1 : p' \parallel \Pi)\end{gathered}} \qquad (\textsc{NSpawn})$$

Figure 5. Semantics of communication between processes

2.2 Mechanising the Formal Semantics

The formal semantics of Core Erlang are mechanised in Coq [21]. The syntax and semantics are implemented with proof simplicity in mind: the syntax is expressed as mutually inductive types with a nameless variable representation [10, 39], enabling general definitions and proofs for capture-avoiding substitution, which is used when evaluating binders, pattern matching, and function application.



Note: Colors indicate implementation languages: beige for Coq, red for Erlang, and blue for Haskell.

Figure 6. System components and their relationships

The semantics was expressed as inductively defined relations. Inductive relations in Coq are not inherently executable: to automatically evaluate a program with the semantics, either a proof tactic (known to be inefficient [2]) or an equivalent function-based semantics is required [2]; we employ the latter approach in the following section.

3 The Derived Interpreter

Turning inductive relations defined in Coq into an executable interpreter takes several steps. The relations describe single evaluation steps between nodes, while an interpreter needs to deterministically choose from these steps, repeatedly.

The interpreter itself cannot be constructed entirely in Coq. Gallina, Coq's specification language only allows total, terminating functions. This is because the main use of the language is proving theorems, and allowing non-termination would lead to logical inconsistencies. Since many Erlang programs rely on infinite recursion, it is impossible to make an interpreter for the language entirely in Coq. Therefore, in this work, we used the following approach:

1. We construct functions in Coq that are equivalent to the three layers of the semantics (Section 3.1);
2. We extract these functions into Haskell using the extraction mechanism in Coq (Section 3.2) and optimise the code for performance and pragmatic reasons;
3. We define a process scheduler in Haskell (Section 3.3);
4. To execute concrete Erlang programs, we build a component to translate Erlang code into a Core Erlang abstract syntax tree in Haskell via Coq (Section 3.4).

Figure 6 displays the components of the system, which will be discussed each in more detail in the following subsections.

3.1 Adapting the Semantics

First, we need to transform the three, inductively defined semantics layers (Table 1) into computable functions. Defining these functions also serves as a validation step for the semantics: due to their totality, unspecified behaviour in the semantics has to be explicitly handled.

```
sequentialStepFunc : FrameStack -> Redex
    -> option (FrameStack * Redex)
processLocalStepFunc : Process -> Action
    -> option Process
interProcessStepFunc : Node -> Action -> PID
    -> option Node
```

Figure 7. Signature of the step functions

All layers define reduction steps between configurations. In the sequential semantics, the configuration contains a frame stack and a redex, the process-local layer reduces processes, while the inter-process semantics handles nodes. These configurations are reflected in the function-based definitions too (Figure 7). The functions implementing the process-local and inter-process rules have an action (and a PID) argument, which determines the reduction step to take, and since the inductively defined semantics is not total, we use `option` in the return types.

We defined these functions in Coq, with one minor conceptual change in the original definition: closedness[7] preconditions from the inductive semantics were omitted. However, we proved that our functions preserve closedness, and based on this property, we also showed the equivalence between the function-based and the inductive semantics for closed constructs (such as concrete Erlang programs).

Sequential Step Function. The sequential step function implements the sequential layer of the semantics. To minimize the number of pattern matches, the function matches on the redex first, and only matches on the frame stack if needed. If the semantic rules have other preconditions, they are checked after the pattern matching. An excerpt is given in Figure 8, showing the four rules for exception handling (indicated by STry, PTry, ExcTry, and ExcProp in comments).

[7] A value, expression, or process is closed if it does not contain free variables.

```
Definition sequentialStepFunc
    (K : FrameStack) (r : Redex) :=
match r with
...
| ETry e1 vl1 e2 vl2 e3 => (* STry *)
    Some ((FTry vl1 e2 vl2 e3)::K, RExp e1)
| RValSeq vs =>
    match K with
    ...
    | (FTry vl1 e2 vl2 e3)::K' => (* PTry *)
        match vl1 =? length vs with
        | true =>
          Some (K', RExp e2.[list_subst vs idsubst])
        | _ => None
        end
| RExc (cl, rn, dt) =>
    match K with
    | (FTry vl1 e2 3 e3)::K' => (* ExcTry *)
      Some (K', RExp e3.[list_subst
        [exclass_to_value cl; rn; dt] idsubst])
    | F::K' => (* ExcProp *)
      match isPropagatable F with
      | true => Some (K', RExc (cl, rn, dt))
      | _ => None
      end
    | _ => None
    end
end
```

Figure 8. Exception handling rules (sequential step function)

Process-local Step Function. The function-based process-local semantics is the most complex of the three functions. While the sequential semantics is deterministic, the process-local semantics is not. For all processes, two kinds of reductions can be performed (based on the concurrent action):

- A computation step (i.e., a non-arrival step), which is deterministic for live processes (e.g., SEQ, EXIT). It is possible that no non-arrival action could be performed on a living process, if it is waiting for a message to arrive. On the other hand, the only action dead processes can perform involves the DEAD rule, which is nondeterministic, due to the unspecified order in which exit signals should be sent to the linked processes.
- A signal can arrive, if the destination process of the signal is alive (e.g., EXITDROP, EXITTERM, EXITCONV).

The step that a process should take is determined by the concurrent action, which is propagated from the inter-process semantics. The scheduling of these actions will be discussed in the following paragraphs. Furthermore, to define exit signal arrival in the function, a complex nested case separation was needed to express the preconditions of EXITDROP, EXITTERM, and EXITCONV.

Worth mentioning that due to the totality of the function-based semantics, a handful of edge cases were discovered that were not covered by the three exit rules. However, in all these cases, the destination of the signal matched its source, and the link flag of the exit signal was set. Processes cannot be linked to themselves, therefore, they cannot send signals to themselves via links, and thus these configurations cannot occur in an Erlang system. For this reason, the function-based definition returns the value `None` of Coq's `option` type in these cases.

Inter-process Step Function. As mentioned before, the inter-process semantics (Figure 5) selects a process from the process pool, and reduces it with a process-local step, while potentially modifying the ether or the process pool of the node. However, the semantics does not specify the order in which processes should evaluate, nor the order of signal arrival. In the case of NSPAWN, the PID of the spawned process is also required to be unused.

We designed the equivalent, function-based variant of this semantics to match the behaviour of the inductive relation; so the function also takes an action and a PID as parameters (`interProcessStepFunc` in Figure 7). However, for an interpreter, using this function directly is cumbersome because the action parameter needs to be constructed for every step.

To have a better interface with the inter-process step function, we wrapped it into another function that only takes the PID (or PIDs), and the action itself is determined from these parameters and the state of the node (see Figure 9). In particular, if this function is applied with a single PID, the process identified by it makes a non-arrival reduction step (either NSEND, NLOCAL, or NSPAWN). When spawning new processes, a fresh PID is assigned to them that does not exist in the process pool or the ether. If the function gets called with a source-destination PID pair, a signal from the ether arrives to the process identified by the destination PID from the source PID (as in NARRIVE).

```
nodeSimpleStep: Node -> PID + (PID * PID)
    -> option (Node * Action)
```

Figure 9. Declaration of the node's simple step function

With this simplified function for the inter-process semantics, only a scheduler is needed to execute it for a concrete program, and the scheduler only needs to generate these PID or PID pair values, which we discuss in Section 3.3.

3.2 Extracting the Code

The Coq proof assistant has a built-in mechanism for extracting function definitions into Haskell, OCaml or Scheme [11].[8] We chose extraction to Haskell not only because it offers a rich ecosystem of libraries and profiling tools, but also because lazy evaluation is generally advantageous for us.

[8]Even though the extraction mechanism is trustworthy, semantics-preservation is not formally guaranteed.

```
Definition pool_lookup (pid : PID) (prs : ProcessPool)
  : option Process := lookup pid prs.
```

```
pool_lookup :: PID -> ProcessPool
    -> Prelude.Maybe Process
pool_lookup pid prs =
  lookup (gmap_lookup eq_dec1 nat_countable) pid prs
```

```
pool_lookup :: PID -> ProcessPool
    -> Prelude.Maybe Process
pool_lookup = Data.HashMap.lookup
```

Figure 10. Wrapper function for lookups in the process pool

Although Haskell extraction is unverified and it is not as mature as it is to OCaml, all the features we use are simple enough to be handled correctly. On the other hand, we may consider verified extraction into OCaml [14] in the future.

During extraction, if a definition depends on other definitions, those will also get extracted automatically. Data type definitions can be swapped for preexisting definitions in the target language. Similarly, function definitions can be replaced with constants and functions can also be inlined.

Using definitions extracted from Coq without utilising the abovementioned extraction customisation features is unlikely to yield the desired result. The generated code will have redundant definitions for existing types and functions in the target language that have been implemented in a more efficient way. The Coq standard library provides some extraction options for basic types such as booleans, natural numbers, integers, and strings. While types do get replaced, the Haskell imports for these need to be included in a post-processing step alongside missing `Eq` and `Show` instance derivations.

Extracting Maps and Sets. The semantics functions we designed in Coq utilised `gmaps` and `gsets` from the `std++` library [41]. These are map and set representations, implemented using extensional binary tries. While these data types behave correctly after extraction, they are not as efficient as Haskell's map and set alternatives. We fine-tuned the extraction of `gmaps` into `HashMaps`, and `gsets` into `HashSets`.

However, these data types are implemented quite differently in Coq and Haskell. The `gmap` implementation uses implicit arguments, which are turned into explicit arguments during the extraction. Haskell's `HashMap` implementation uses type classes instead. This makes the interfaces of the operations incompatible. To mitigate this issue, we use wrappers around `std++` functions; they can still be evaluated in Coq, but during extraction they get replaced and inlined with their Haskell counterparts. These replacements are not proved correct; instead we use property-based testing in QuickCheck [9] for the map and set functions in the different implementations to verify that they give equal results.

As an example, Figure 10 presents `pool_lookup`, a wrapper function for looking up processes inside the process pool. The top of the figure is the Coq implementation, where `lookup` is the lookup function of `std++` maps. The middle code segment is the extracted Haskell version without any replacements during extraction. Since implicit arguments are turned explicit during extraction, `lookup` has an additional argument, which contains information about the countability of natural numbers and the decidability of equality between them. Haskell's `HashMap.lookup` does not need these, since it is implemented using type classes. Only `pid` and `prs` is needed. The bottom of the figure presents the function after the constant replacement during extraction: The type signature of `pool_lookup` matches that of `HashMap.lookup`. Note that functions replaced with constants are eta-reduced. For even more efficiency, definitions can also be inlined.

One caveat with this approach is the type definitions. Since `gmaps` and `gsets` are used in some declarations, the type definitions themselves need to be replaced with aliases to Haskell's `HashMaps` and `HashSets`. Unlike with functions, Coq's inductive types cannot be freely replaced with constants in the extraction mechanism. Therefore, these changes had to be done during the post-processing step.

Strict Evaluation of Substitutions. One of the key properties of Haskell is lazy evaluation. Evaluating expressions only when they are needed comes with many advantages, such as performance improvements or infinite data structures. There are however some disadvantages to lazy evaluation. Unevaluated expressions are kept in memory as thunks, and these thunks can easily cause issues with space leaks[9].

Substitutions in the sequential layer of the semantics cause memory management issues with lazy evaluation. Substitutions traverse the entire structure of the program, and evaluating them using lazy evaluation caused exponential thunk build-up in the memory, consuming as much as a gigabyte in 10 seconds. To overcome this, our interpreter can evaluate substitutions strictly using Haskell's `DeepSeq` library: the function `deepseq` forces the full evaluation of its first argument before its second. Unfortunately, this is a major step backwards in terms of runtime, as is shown in Section 5; however, without this change, the interpreter would be incapable of operating on a standard personal computer.

Figure 11 presents how the `subst` function was made strict using `DeepSeq`. Since all other substitution functions are called by `subst` or a descendant of `subst`, only this function needs to be made strict.

Simplified Substitutions. Another performance limitation originated from the general definition of capture-avoiding parallel substitution. To ensure capture-avoidance, the substitution relies on renaming of free variables in the substituted values. However, if the substituted values are closed (i.e., they do not contain variables), this renaming is an unnecessary traversal of the value's syntax tree. In

[9]Unlike other forms of memory leaks, space leaks can get resolved after the evaluation of thunks, given enough memory.

```
subst :: Substitution -> Exp -> Exp
subst xi base =
  case base of { VVal v -> VVal (substVal xi v);
                 EExp e -> EExp (substNonVal xi e)}
```

```
subst :: Substitution -> Exp -> Exp
subst = (\xi base ->
  case base of {
   VVal v -> let v' = (substVal xi v)
             in v' `deepseq` VVal v';
   EExp e -> let e' = (substNonVal xi e)
             in e' `deepseq` EExp e'})
```

Figure 11. Strict evaluation of substitutions

the evaluation of concrete programs, due to the strictness of (Core) Erlang, substituted expressions are always fully evaluated, which could not be done if there are free variables. Therefore, we specialised substitutions for closed values by eliminating the unnecessary renaming operations.

3.3 Deterministic Scheduling

As explained already, Erlang program behaviour is nondeterministic: the runtime system schedules processes' computation steps, and external factors determine whether and when signals get sent or delivered. This nondeterminism appears in the formal semantics as either multiple rules being applicable to the current node configuration, or particular rules being applicable with different arguments.

For instance, when multiple processes can make local computation steps, they race for the NLocal rule to be applied with their PIDs; similarly, if a dead process is sending exit signals to its links (where the notification order is not defined), the pending messages are racing for NSend to emit them into the ether. Processes may also be eligible to progress with multiple node semantics rules: they may be waiting for messages to be delivered with NArrive, at the same time of awaiting local computations to be executed by NLocal.

Building the interpreter on top of the formal semantics, we designed it to run programs under an arbitrary but deterministic scheduling strategy; this ensures that running the program multiple times using the same scheduling will give consistent results. To this end, choices were abstracted from the semantic functions into a so-called scheduler component, which only has to specify the PID (identifying the process that takes computation steps) or a pair of PIDs (identifying the processes between which communication action will be carried out) to take action with.

Interpretation Cycles. Execution happens in cycles. Each cycle activates a process, allows it to take local steps, then the node delivers signals awaiting arrival. More precisely,

1. The active process is allowed to take at most k non-arrival steps (local computation, signal sending or process spawning, determined by the state of the process), and if at any point a non-arrival step is not possible (e.g., because the process is waiting for a message to arrive), the interpreter moves to the next point.
2. If the active process terminates, all its links get notified immediately (signals are sent).
3. All signals floating in the ether are delivered to their destination (making arrival actions).[10]

With this strategy, we uniquely define a sort of action for every node state, eliminating the nondeterminism stemming from the multiple semantic rules applicable in the operational semantics. Note that in this setting, the node state includes the semantic configuration (describing the entire state of the Erlang node) along with the identifier of the active process.

However, even if the rule to be applied is determined by the above strategy, the parameterisation may vary: in (2) the notification order of the dead process's links is not defined and therefore nondeterministic, as well as in (3) the order of signal arrivals (across the entire system) is not defined and an arbitrary order can be applied as long as it respects the signal ordering guarantee.

Scheduling Signal Sending in Dead Processes. When processing dead processes, we transform the links into a list from their original map representation, and the exit signals are sent to the links based on their order in this list. While this is not ideal, as nondeterminism for dead processes cannot be resolved in a custom way, since the interpreter sends out all the exit signals from a dead process and then in a subsequent step all these signals are delivered, the order of them becomes irrelevant.

Scheduling Signal Arrival. During signal arrival, nondeterminism is caused by two or more processes having sent signals to the same process, since the order in which these signals arrive at the destination process is not specified. Note, however, that the order of multiple signals from the same process must be the same during arrival, so in this scenario the order is defined. Now observe that in our solution, only the latter scenario can happen, the signals in the ether after the k non-arrival steps fall into two categories:

- Signals with destination PIDs not present in the process pool. These signals will never arrive, as processes will not spawn with these destination PIDs.
- Signals with source PIDs that match the current process in focus. Note that this category is not necessarily distinct from the previous one.

When the interpreter tries to deliver all signals from the ether, all signals that can be delivered will have the same source PID. The same can also be said for the exit signals from dead processes. Since no arrivals are happening between the same destination and different sources, nondeterministic behaviour is simply ruled out.

[10]Signals that cannot be delivered remain in the ether.

Scheduling Processes for Non-arrival Steps. Since the interpreter runs on a single thread, there is always exactly one active process in our simulated Erlang system. To choose the next process in focus, a *Scheduler* typeclass was developed in Haskell, making schedules parametric. A scheduler needs to be able to provide PIDs for non-arrival steps or pairs of PIDs for arrival steps; it also needs to adjust its inner state based on the action taken (e.g. process spawning). Note that the separation of the scheduler facilitates formal reasoning about its properties (such as starvation-freedom) in the theorem prover, and integration after extraction.

By default, our interpreter uses basic round-robin scheduling [24] to choose the next active process. The scheduler tracks PIDs of processes in a list and uses an index to point to the currently evaluating PID. This index gets decreased by 1 at process termination, increased by 1 at spawning a new process, and increased by 1 when moving along to the next process (after the last element of the list, the index gets reset to 0). When a new process is spawned, the PID of the new process gets added to the start of the list. When a process terminates, after all the exit signals get sent out to the linked processes, the PID of said process gets removed from the list. An empty scheduler list means that all processes have terminated and thus the computation is finished.

3.4 Integration

For our formally based interpreter to execute Erlang programs we had to implement a gluing component that translates Erlang program text into Core Erlang abstract syntax tree (AST) expressed in Coq. Similarly to previous work [2], we did not implement a parser in Coq for Core Erlang, but rather exploited the standard compiler's features: namely, we translated Erlang into Core Erlang, and implemented a pretty-printer (based on the abstract syntax of Core Erlang in Erlang). In contrast to previous work [2], this pretty-printing is not only translating Core Erlang AST in Erlang into an AST in Coq: the Coq formalisation uses a nameless variable representation [10], so erasure of (variable) names from the syntax trees was also necessary.

To start the evaluation of the translated Erlang program, we extract the program and the step functions to Haskell. We then initialize a node with a single process that evaluates the `main/1` function of the extracted program; the semantics can then be executed on this node by the scheduler.

4 The Computation Graph

The interpreter presented in the previous section explores only a single execution path per program run, employing a flexible but deterministic scheduling policy. As a result, running a program consisting of deterministic processes will show deterministic behaviour. However, sometimes what we need is not to execute a particular scheduling but to have an overview of all the possible interleavings.

The observable behaviour of a concurrent program is determined by the order in which key concurrency and communication operations occur, such as process creation, signal sending and arrival. Our semantics explicitly models these points of nondeterminism, allowing us to formally capture and analyse the range of possible execution paths; this has already been demonstrated in a case study [6, Figure 13].

To better support this kind of program comprehension, we made an alternate interpreter that, rather than following a single deterministic scheduling, enumerates every possible execution path. Building on this formal semantics-based state exploration technique, we present a tool that visualises the execution paths of concurrent Erlang programs concisely, allowing developers to analyse and understand program behaviour under different schedulings.

4.1 Design

The various execution paths are captured in a *computation graph*, whose nodes correspond to system states (configurations in the formal semantics) and whose edges represent the application of semantic rules to transition system states.

In general the number of potential node states is infinite, and the number of reduction steps, even in terminating systems, may be vast. Therefore, the computation graph we present does not directly mirror each individual interpreter step; rather, to ensure clarity and scalability, we apply a number of principles in the construction of these graphs.

We compress τ- (and *self*) reductions. The vast majority of steps occurring within the sequential layer of the semantics are τ-reductions. Representing each such step explicitly would lead to an overwhelming number of nodes without contributing to the comprehension of concurrent behaviour. Where appropriate, these reductions are *grouped together*. Since such reductions are deterministic and confluent ([6, Theorem 4]) we do not risk missing any graph forks.

The computation graph is not necessarily a tree: nodes may be *shared* between different execution paths. That is, if multiple paths lead to the same semantic configuration, we represent this convergence explicitly by reusing the corresponding node, rather than duplicating it. This allows the graph to reflect where execution paths *diverge and merge*, which is essential for understanding non-deterministic concurrent computation.

Building the Graph. The graph builder component uses the following process, based on the rules discussed above.
(1) Constructing the initial configuration: A graph corresponding to a particular program is built iteratively: initially it contains only a root node depicting the starting configuration, which is selected for further processing.
(2) Executing local computation steps: Next, every process in its configuration's process pool needs to be "advanced" further in their evaluation through τ and *self* actions. To prevent infinite evaluation of non-communicating processes,

at most k steps can be performed at each stage per process in the pool. If no further τ or *self* steps can be made for particular process, then it either a) has finished its execution; b) is blocked; c) has run out of the above limit of steps; or d) can take a step with an action other than τ or *self*.
(3) Merging with existing states: In any of the cases, once all the possible processes have been "advanced", a new node is formed, depicting a configuration state before performing a critical Erlang-node-level reduction, which is also a potential point of non-deterministic path divergence. However, before deriving the outgoing edges of the new node, it is checked against all existing nodes: if two nodes contain the same semantic configurations then they are merged and the building function for the current graph path stops.
(4) Forking new execution paths: Otherwise, the graph builder examines which path forks are possible from the newly constructed node. The options correspond to the possible reduction steps for the processes in the pool. Each of these steps, represented by the tuple of a targeted PID and an action taken, marks a separate outgoing edge from the graph node. Based on the computed map of such PID-action values, the next path continuations can be constructed. If the map is empty, then construction below this node is terminated.
(5) Executing the forks recursively: Next, for every option in the map, the path's continuations are built in a recursive manner. First, the chosen reduction step needs to be applied to the semantic configuration. Afterwards, the instructions starting from Step (2) are repeated until the graph path is concluded or the depth-bound provided, m, is reached.

Conceptual Optimisations. A potential optimisation concerns Step (2). When a new node is formed, all the processes in the process pool are advanced up to their "limit" within the context of τ-related actions (if a process actually exceeds the k step limit, it is assumed to be infinitely running and non-communicating). Therefore, after performing system-level configuration reduction from Step (5), the only processes that would be able to "advance" with Step (2) further are the ones transformed by the action taken. Thus, only those processed affected by the transformation performed at Step (5) need to be examined each time.

Another improvement concerns Step (4). The possible system-level reduction steps do not have to be recomputed each time a new node is formed for the processes that have not been advanced in Step (2). In other words, recomputing the PID-action map in Step (4) should happen only for the processes targeted by the system-level reduction performed at the start of the graph subpath. The rest of the values can be reused from the previous iteration.

4.2 Implementation

The implementation of the computation graph constructor is made up of three components: an abstract graph builder, a data translator to JSON-format, and a graph displayer.

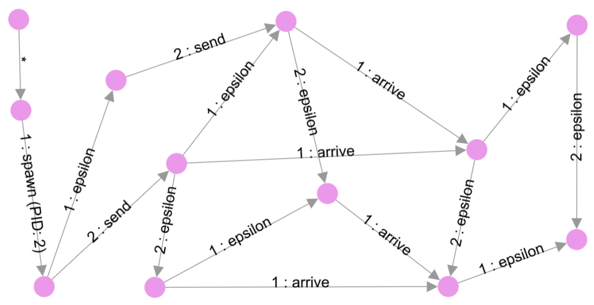


Figure 12. The computation graph of *pmap* example (WIP).

The Graph Builder. The abstract graph builder is implemented in Haskell and reuses the extracted functions from Section 3.2. It follows the workflow described in Section 4.1, with the following architectural approach.

The *graph node structure* includes the following fields: a node depth limit, an optional PID-action tuple representing the preceding system level reduction step, a map of PID-action tuples representing "pending" system-level reductions computed at earlier steps,[11] a list of PIDs that exceeded the step limit for τ reductions at the previous step, and a semantic node configuration along with its hash.

The *hash node* field is present to identify the nodes and facilitate their "symbolic" merging. This decision also allows the use of tree-like data structures to store the graph itself.

Despite the approach being general, there are still scenarios where the computation graph does not depict *all* possible execution paths. This issue stems from the implementation not supporting non-deterministic notification of links; instead, the links are always sent signals in a particular order. In addition, many system-level reductions as well as τ-steps are recomputed several times while being on different execution paths; this can be mitigated using memoization.

Data Translation and Graph Display. The data translator transforms and outputs the internal graph data structure to the format processable by the graph display component to the file, which can be later passed to the display module.

The display component is the one virtually responsible for showing the computation graph. The graph visualisation problem in general is not new, and there are multiple software solutions addressing it: for instance, Graphviz [18] and its DOT language have been in use since the 1990's. We chose instead to adopt a solution that supports dynamic display and inspection of information, including configurations: *Cytoscape.js* [15] allows importing graph data from JSON format, making it a suitable option for representing the graph, and it provides a smooth, intuitive node rearrangement, which is essential for displaying large, dense graphs.

[11] Excluding message arrivals. Those are derived from the state of the ether.

Figure 12 displays a capture of the computation graph constructed from the `pmap` example (Figure 2). We note that `receive` expressions in Core Erlang consist of several primitive operations, which are denoted by "epsilon". The different paths on the graph show all possible orders in which primitive operations of `receive` and message sending (and arrival) from process 2 can happen. For example, in the topmost path process 1 tries to receive a message that has not been sent, and gets stuck on the infinity timeout until that message arrives from process 2 (by "2 : send" and "1 : arrive"). Then process 1 receives the message ("1 : epsilon") and concatenates the two list segments. The final step ("2 : epsilon") denotes the termination of process 2.

5 Evaluation

The semantics itself covers a significant subset of Core Erlang. Since both the interpreter and the graph drawer build on functions equivalent to the semantics, they cover the same subset of the language. Notable features not covered include floating-point numbers, binaries, and the module system.

The ErLLVM Benchmarking Suite was used [42] for testing the interpreter. The pretty-printer (Section 3.4) gives warnings for features not implemented in the semantics; if it produces no warnings, then the interpreter can run the program. The semantics defined in [6, 21] uses 2-parameter variants of `spawn` and `spawn_link` due to the lack of a formalised module system: applications of these functions in our tests had to be rewritten into this format manually.

The machine used for evaluation has an 11th Gen Intel Core i5-1135G7 processor, with a clock speed of 2.4 GHz and 16 GB of RAM. Since the interpreter runs on a single thread, the number of CPU cores is irrelevant. Haskell modules were compiled with GHC 9.6.7, using the `-O2` flag.

Tests were conducted for three example programs using different optimisations. The first is an evaluation of `pmap` (Figure 2); a long list of zeros is transformed by adding one to each element. The second and third are modified programs from the ErLLVM Suite. `length` constructs a 20 thousand element long list and calculates its length 500 times. `life` is an implementation of John Conway's Game of Life, on a 10×10 board for 10,000 iterations. `pmap` use 2 processes, `length` only 1, and `life` spawns 100, one per cell.

The four tests for each program were:

1. Running the extracted program untransformed.
2. Evaluating substitutions strictly.
3. Replacing the extracted `gmap`s and `gset`s with Haskell `HashMap`s and `HashSet`s.
4. Replacing substitutions with a simplified counterpart.

Each of these transformations subsumes the preceding ones. Table 2 shows the data collected during the evaluation. The execution time and memory usage figures for substitutions, map and set operations, and scheduler operations are broken down by percentages. A runtime estimate is given for each test, along with a maximum memory usage figure in 2 minutes of runtime. Note that in the highlighted rows, substitutions are evaluated lazily, resulting in a space leak. The maximum memory consumed in 2 minutes therefore does not reflect the true maximum of memory usage.

These tests revealed that substitution is by far the most significant bottleneck; this should be no surprise, since substitutions traverse the entire structure of the program every time a binder (e.g., `let`, `try`, `case`) is evaluated. As discussed in Section 3.2, this traversal cannot be avoided with lazy evaluation due to the accumulation of unevaluated thunks. Not counting evaluations with space leaks, substitutions accounted for almost all the execution time and memory usage in the tests. This reflects earlier work on interpreter performance, as discussed in Section 6.

6 Related Work

The first interpreters were built in the 1950s, and they continue to be developed now, as evidenced by books like *Crafting Interpreters* [34]. Reynolds' work on definitional interpreters [37] takes a more rigorous approach where higher-order functions in the interpreter are systematically replaced by closures. Language workbenches, e.g. Spoofax [23], offer a different approach by providing toolsets for building tools, including interpreters, for domain-specific languages.

The approach taken in this paper is to *extract* a program from an artefact that has been mechanised in a theorem prover. Many theorem provers support this, including Isabelle and Coq [11], from which it is possible to derive code in OCaml (the default), Haskell and Scheme.

Mechanised meta-theoretic proofs about language properties can be the basis for extracting interpreters from those proofs. Early work [7] extracts code from normalization proofs: formalisations of Tait's normalization proof for the simply typed λ-calculus are developed in the proof assistants Minlog, Coq and Isabelle/HOL, and normalization-by-evaluation algorithms are extracted from each of these. More recently, Wadler *et al.* [25] extracted an evaluator from a constructive proof of preservation and progress in Agda. Using intrinsically-typed definitional interpreters [44], it is possible to build "progressful" interpreters whose types certify their own soundness and progress [36]. Complementary work shows how a formally verified compiler for a subset of Java (rather than an interpreter) can be extracted from a proof assistant [8]. The CakeML [27] and CompCert [32] projects show that formalised language processing scales to practical, real-world languages, namely ML and C.

Finally, the K framework [38] can automatically generate interpreters directly from a language's small-step semantics rules. Moreover, the derived interpreters achieve decent performance by compiling the rewrite rules' pattern matching into optimized decision trees, which are subsequently translated into low-level code through K's LLVM-based backend.

Table 2. Summary of data gathered during evaluation tests

		Execution time (%)				Memory usage (%)				Max mem.	runtime
		subst	map/set	sched	other	subst	map/set	sched	other	in 2 min.	(est.)
pmap (100000 list)	raw Coq extraction	25.2	42.3	1.3	30.2	30.0	26.1	1.2	42.7	1.5 GB *	3 s
	strict substitutions	96.8	1.2	0.0	2.0	48.7	14.3	0.8	36.2	48 MB *	44 s
	maps/sets replaced	98.3	0.1	0.0	1.6	55.2	6.3	0.9	37.6	49 MB *	44 s
	improved substitution	98.1	0.2	0.0	1.7	38.8	8.4	1.3	51.5	46 MB *	44 s
length (20000 list, 500 iters)	raw Coq extraction	89.3	5.5	0.2	5.0	91.3	2.9	0.1	5.7	>16 GB **	*(killed)*
	strict substitutions	98.4	0.9	0.0	0.7	95.8	1.4	0.1	2.7	12 MB	87 min
	maps/sets replaced	99.2	0.1	0.0	0.7	96.5	0.5	0.1	2.9	12 MB	87 min
	improved substitution	98.8	0.1	0.0	1.1	91.9	1.1	0.2	6.8	11 MB	56 min
life (10x10 board, 10000 iters)	raw Coq extraction	33.6	40.8	5.8	19.8	38.5	49.5	5.0	7.0	984 MB	1.4 h
	strict substitutions	92.1	6.1	0.5	1.3	90.5	8.1	0.7	0.7	84 MB	16.2 h
	maps/sets replaced	97.3	1.2	0.5	1.0	97.5	1.0	0.7	0.8	83 MB	15.8 h
	improved substitution	96.2	1.1	0.9	1.8	85.2	4.3	4.8	5.7	64 MB	6.6 h

Note: Executions highlighted with red have space leak issues
* The program terminated before the 2 minute mark
** The machine used for evaluation ran out of memory before the 2 minute mark

Interpreter Performance. Wadsworth's PhD thesis [45] builds an interpreter for the λ-calculus based on expression substitution (as we do). Interpreter performance was improved by compiling to combinators [43] and supercombinators [12], and culminating in the spineless, tagless, G-machine [22] that underlies the current implementation of Haskell. These improved the efficiency of evaluating (lazy) functional programs by moving away from directly rewriting expressions, thus no longer reflecting a direct interpretation of a high-level semantics.

Formal Semantics of Erlang. The semantics of Core Erlang [4, 6] used for this project is not the first formalisation of this language. The most well-known formalisation of Erlang (including sequential, concurrent, and distributed semantics too) was created by Fredlund [16, 40], which served as a basis for the McErlang [17] model checker. There is extensive research conducted by Lanese et al. [26, 28–31] about reversible semantics of Core Erlang and casual debugging. Furthermore, there are also machine-checked formalisations of subsets of Core Erlang in Isabelle [1, 20]. Our interpreter is based on the semantics made by Bereczky et al.[6] because it defines a recent version of the language (e.g., it implements a change about `receive` expressions from Erlang/OTP 23 [19]), and is fully mechanised, including a set of theorems.

7 Conclusion

We have presented an interpreter for Erlang derived from a mechanised formal semantics for Core Erlang in Coq. We have described the ecosystem needed to build an optimised version of the tool that covers the Erlang language, using a scheduler to present a deterministic semantics. We have also built a rendering of the computation graph that shows the non-deterministic choice points that arise during computation of concurrent programs in Erlang.

This work has delivered a working, high-assurance reference interpreter for Erlang with less than a person-year of work. The tool is correct by construction as it is derived from the semantics, relying only on the correctness of the extraction tooling, the code optimisations and the standard translation from Erlang to Core Erlang.

Future Work. We have presented a minimal viable interpreter for Erlang, based on the semantics mechanised in Coq. Beyond the discussion in the paper, We will investigate further ways of improving the interpreter performance while preserving the trustworthiness of the code.

We aim to extend the coverage of the interpreter to features of Erlang not yet covered, including floating-point numbers, bitstrings, further signals (e.g., *monitor*, *demonitor*) and process dictionaries. Incorporating these requires changes to the formal semantics, with consequent revisions of proofs about the semantics. However, these changes are expected not to affect all three semantics layers and proofs. For example, adding binaries and floats needs additions on the sequential level, while formalising further signals and process dictionaries has to be done on the process-local level.

We will also fully develop the graph construction to a functional state, and address its behavioural issues discussed previously. Beyond this we will explore ways of making the (presentation of the) computation graph interactive.

Acknowledgments

We are very grateful to the Erlang Ecosystem Foundation for their financial support of the work reported in this paper.

References

[1] Ibrahim Abdelrahman Mohamedi. 2025. *A Formal Semantics of Core Erlang*. Master's thesis. Uppsala University, Department of Information Technology. Retrieved August 22nd, 2025 from https://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Auu%3Adiva-548800

[2] Péter Bereczky, Dániel Horpácsi, Judit Kőszegi, Soma Szeier, and Simon Thompson. 2021. Validating Formal Semantics by Property-Based Cross-Testing. In *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages* (Canterbury, United Kingdom) *(IFL '20)*. Association for Computing Machinery, New York, NY, USA, 150–161. doi:10.1145/3462172.3462200

[3] Péter Bereczky, Dániel Horpácsi, and Simon Thompson. 2020. A proof assistant based formalisation of a subset of sequential Core Erlang. In *Trends in Functional Programming*, Aleksander Byrski and John Hughes (Eds.). Springer, Cham, Germany, 139–158. doi:10.1007/978-3-030-57761-2_7

[4] Péter Bereczky, Dániel Horpácsi, and Simon Thompson. 2024. A frame stack semantics for sequential Core Erlang. In *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages* (Braga, Portugal) *(IFL '23)*. ACM, New York, NY, USA, Article 5, 13 pages. doi:10.1145/3652561.3652566

[5] Péter Bereczky, Dániel Horpácsi, and Simon J. Thompson. 2020. Machine-checked natural semantics for Core Erlang: exceptions and side effects. In *Proceedings of Erlang 2020*. ACM, New York, NY, USA, 1–13. doi:10.1145/3406085.3409008

[6] Péter Bereczky, Dániel Horpácsi, and Simon Thompson. 2024. Program equivalence in the Erlang actor model. *Computers* 13, 11 (2024), 32 pages. doi:10.3390/computers13110276

[7] Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. 2006. Program Extraction from Normalization Proofs. *Studia Logica: An International Journal for Symbolic Logic* 82, 1 (2006), 25–49. doi:10.1007/s11225-006-6604-5

[8] Stefan Berghofer and Martin Strecker. 2004. Extracting a formally verified, fully executable compiler from a proof assistant. *Electronic Notes in Theoretical Computer Science* 82, 2 (2004), 377–394. doi:10.1016/S1571-0661(05)82598-8 COCV'03, Compiler Optimization Meets Compiler Verification.

[9] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279. doi:10.1145/357766.351266

[10] N. G. de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. doi:10.1016/1385-7258(72)90034-0

[11] Rocq documentation. 2025. *Program extraction*. Inria. Retrieved June 6th, 2025 from https://rocq-prover.org/doc/v8.20/refman/addendum/extraction.html

[12] Jon Fairbairn and Stuart Wray. 1987. TIM: A simple lazy abstract machine to implement supercombinators. In *Functional Programming Languages and Computer Architecture*, Gilles Kahn (Ed.). Springer, Heidelberg, 34–45. doi:10.1007/3-540-18317-5_3

[13] Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, Martin Wirsing (Ed.). North-Holland, Amsterdam, Netherlands, 193–222.

[14] Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. 2024. Verified Extraction from Coq to OCaml. *Proc. ACM Program. Lang.* 8, PLDI, Article 149 (June 2024), 24 pages. doi:10.1145/3656379

[15] Max Franz, Christian T. Lopes, Gerardo Huck, Yue Dong, Onur Sumer, and Gary D. Bader. 2015. Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics* 32, 2 (09 2015), 309–311. doi:10.1093/bioinformatics/btv557

[16] Lars-Åke Fredlund. 2001. *A framework for reasoning about Erlang code*. Ph. D. Dissertation. Mikroelektronik och informationsteknik.

[17] Lars-Åke Fredlund and Clara Benac Earle. 2006. Model checking Erlang programs: the functional approach. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang* (Portland, Oregon, USA) *(ERLANG '06)*. ACM, New York, NY, USA, 11–19. doi:10.1145/1159789.1159793

[18] Emden R. Gansner and Stephen C. North. 2000. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.* 30, 11 (Sept. 2000), 1203–1233. doi:10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N

[19] Björn Gustavsson. 2020. *EEP 52: Allow key and size expressions in map and binary matching*. Ericsson Computer Science Laboratory. Retrieved August 22nd, 2025 from https://www.erlang.org/eeps/eep-0052

[20] Joseph R. Harrison. 2017. Towards an Isabelle/HOL formalisation of Core Erlang. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang* (Oxford, UK) *(Erlang 2017)*. ACM, New York, NY, USA, 55–63. doi:10.1145/3123569.3123576

[21] High-Assurance Refactoring Project. 2025. *Core Erlang formalization*. Eötvös Loránd University. Retrieved August 22nd, 2025 from https://github.com/harp-project/Core-Erlang-Formalization/releases/tag/v1.1.0

[22] Simon Peyton Jones and Jon Salkild. 1989. The spineless tagless G-machine. In *Functional Programming Languages and Computer Architecture*. ACM, New York, NY, USA, 184–201. doi:10.1145/99370.99385

[23] Lennart C.L. Kats and Eelco Visser. 2010. The spoofax language workbench: rules for declarative specification of languages and IDEs. *SIGPLAN Not.* 45, 10 (Oct. 2010), 444–463. doi:10.1145/1932682.1869497

[24] Leonard Kleinrock. 1964. Analysis of a time-shared processor. *Naval research logistics quarterly* 11, 1 (1964), 59–73.

[25] Wen Kokke, Jeremy G. Siek, and Philip Wadler. 2020. Programming language foundations in Agda. *Science of Computer Programming* 194 (2020), 102440. doi:10.1016/j.scico.2020.102440

[26] Aurélie Kong Win Chang, Jérôme Feret, and Gregor Gössler. 2023. A semantics of Core Erlang with handling of signals. In *Proceedings of the 22nd ACM SIGPLAN International Workshop on Erlang* (Seattle, WA, USA) *(Erlang 2023)*. ACM, New York, NY, USA, 31–38. doi:10.1145/3609022.3609417

[27] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 179–191. doi:10.1145/2535838.2535841

[28] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. 2018. CauDEr: a causal-consistent reversible debugger for Erlang. In *International Symposium on Functional and Logic Programming*, John P. Gallagher and Martin Sulzmann (Eds.). Springer, Springer, Cham, Germany, 247–263. doi:10.1007/978-3-319-90686-7_16

[29] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. 2018. A theory of reversibility for Erlang. *Journal of Logical and Algebraic Methods in Programming* 100 (2018), 71–97. doi:10.1016/j.jlamp.2018.06.004

[30] Ivan Lanese, Adrián Palacios, and Germán Vidal. 2021. Causal-consistent replay reversible semantics for message passing concurrent programs. *Fundam. Inf.* 178, 3 (jan 2021), 229–266. doi:10.3233/FI-2021-2005

[31] Ivan Lanese, Davide Sangiorgi, and Gianluigi Zavattaro. 2019. *Playing with bisimulation in Erlang*. Springer, Cham, Germany, 71–91. doi:10.1007/978-3-030-21485-2_6

[32] Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. doi:10.1007/s10817-009-9155-4

[33] Peter D. Mosses. 2006. Formal Semantics of Programming Languages: — An Overview —. *Electronic Notes in Theoretical Computer Science* 148, 1 (2006), 41–73. doi:10.1016/j.entcs.2005.12.012

[34] Robert Nystrom. 2021. *Crafting interpreters*. Genever Benning. https://craftinginterpreters.com

[35] Andrew M. Pitts. 2002. Operational semantics and program equivalence. In *Applied Semantics*, Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva (Eds.). Springer, Berlin, Heidelberg, 378–412. doi:10.1007/3-540-45699-6_8

[36] Xiaojia Rao, Stefan Radziuk, Conrad Watt, and Philippa Gardner. 2025. Progressful Interpreters for Efficient WebAssembly Mechanisation. *Proc. ACM Program. Lang.* 9, POPL, Article 22 (Jan. 2025), 29 pages. doi:10.1145/3704858

[37] John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation* 11 (1998), 363–397. doi:10.1023/A:1010027404223

[38] Grigore Roșu and Traian Florin Șerbănuță. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. doi:10.1016/j.jlap.2010.03.012

[39] Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: reasoning with de Bruijn terms and parallel substitutions. In *Interactive Theorem Proving*, Christian Urban and Xingyuan Zhang (Eds.). Springer, Cham, Germany, 359–374. doi:10.1007/978-3-319-22102-1_24

[40] Hans Svensson and Lars-Åke Fredlund. 2007. A more accurate semantics for distributed Erlang. In *Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop* (Freiburg, Germany) *(ERLANG '07)*. ACM, New York, NY, USA, 43–54. doi:10.1145/1292520.1292528

[41] The std++ developers and contributors. 2024. *Rocq-std++: An extended "Standard Library" for Rocq*. Iris project. Retrieved August 22nd, 2025 from https://gitlab.mpi-sws.org/iris/stdpp/

[42] Yiannis Tsiouris, Christos Stavrakakis, and Kostis Sagonas. 2012. *ErLLVM Benchmark Suite*. National Technical University of Athens. Retrieved June 11th, 2025 from https://github.com/cstavr/erllvm-bench/tree/shootout2

[43] David A. Turner. 1979. A new implementation technique for applicative languages. *Software — Practice and Experience* 9 (1979), 31–49. doi:10.1002/SPE.4380090105

[44] Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter Mosses. 2022. Intrinsically-typed definitional interpreters à la carte. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 192 (Oct. 2022), 30 pages. doi:10.1145/3563355

[45] Christopher Wadsworth. 1971. *The semantics and pragmatics of the lambda calculus*. Ph. D. Dissertation. Department of Mathematics, University of Oxford.

Received 2025-06-13; accepted 2025-08-15