Automatic Detection of Core Erlang Message Passing Errors

Joseph Harrison
University of Kent
Canterbury, United Kingdom
jrh53@kent.ac.uk

Abstract

Erlang's powerful communication model allows us to build high-level concurrent systems. These can, however, harbour subtle communication errors less severe than global deadlock or crashes: messages never received can degrade performance and consume swaths of memory. We believe that some of these errors can be quickly detected with static analysis. We have built a prototype tool which operates at the Core Erlang level to assist identification of some of these errors.

We present a fragment of Erlang's type system as a subtyping relation, following up with type inference functions for a portion of Core Erlang's patterns, guards, and message syntax. The implementation of the prototype is detailed, noting specific behaviours of the Erlang compiler and nuances of Core Erlang's syntax along the way, some of which complicate our analysis.

Although our tool is at a very early stage of development, we show examples of the errors we can identify, despite using a considerable over-approximation in our type inference system. After comparing our tool to other work in the Erlang community and beyond, we reflect on the current state of the prototype, before considering further applications of our concept of message compatibility.

CCS Concepts • **Software and its engineering → Concurrent programming languages**; **Patterns**; *Compilers*; • **Theory of computation → Program analysis**; *Program verification*; *Concurrency*;

Keywords Core Erlang, Erlang, communication, static analysis, type inference

Erlang '18, September 29, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5824-8/18/09…$15.00
https://doi.org/10.1145/3239332.3242765

ACM Reference Format:
Joseph Harrison. 2018. Automatic Detection of Core Erlang Message Passing Errors. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang (Erlang '18), September 29, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3239332.3242765

1 Introduction

In order to quickly detect messaging discrepancies in Erlang programs *during* development, we have developed a command line tool to perform a lightweight type-based static analysis of Core Erlang source files.

Erlang's communication model is powerful: Sends do not block, while `receive` operations are selective, and mailboxes permit out-of-order message reception. Combined with `receive` timeouts, distributed nodes, and unbounded mailboxes, Erlang programmers can quickly build resilient high-level concurrent applications.

An unfortunate consequence is that the communication model can be difficult to reason about: subtle message-passing behaviours and incorrect implementations of communicating components can lead to undesired behaviour and performance problems.

In figure 1, we have a simple communicating system. The client communicates with the server, getting values from it, printing them, and setting a decremented value, and recursing. The client returns `'ok'` when done, while the server loops forever.

The server keeps track of the value of N, sending replies to each of the client's messages. There is a subtle error in this code, though: the client never receive's the server's bare `'ack'` messages in response to the `'set'` message tuple. These acks will sit in the mailbox of the client process forever, consuming more and more memory. This might not seem too severe if the client process terminates quickly, but consider the implications if this were a long-running server process.

An alternative implementation is shown in figure 2, with the start and server code remaining the same. In this module, the client receives the bare `'ack'` messages, but the server's first `receive` clause is unused, as the client never sends a `'get'` message!

Note the subtlety of these errors: they cause no deadlock, are not simple typos, do not necessarily alter the intended behaviour of the program, but the messaging discrepancies

```erlang
-module(example1).
-export([start/1,server/1]).

start(N) ->
    S = spawn(?MODULE, server, [0]),
    client(S, N).

server(N) ->
    receive
        {get, From} ->
            From ! {ack, N},
            server(N);
        {set, N2, From} ->
            From ! ack,
            server(N2)
    end.

client(_Server, 0) ->
    ok;
client(Server, N) ->
    Server ! {get, self()},
    receive
        {ack, M} -> io:format("~p~n", [M])
    end,
    Server ! {set, N-1, self()},
    client(Server, N-1).
```

Figure 1. An Erlang program with orphan messages

```erlang
-module(example2).
-export([start/1,server/1]).

start(N) ->
    S = spawn(?MODULE, server, [0]),
    client(S, N).

server(N) ->
    receive
        {get, From} ->
            From ! {ack, N},
            server(N);
        {set, N2, From} ->
            From ! ack,
            server(N2)
    end.

client(_Server, 0) ->
    ok;
client(Server, N) ->
    Server ! {set, N-1, self()},
    receive
        ack -> io:format("ok~n")
    end,
    client(Server, N-1).
```

Figure 2. A different implementation of the `client` function

remain. In the first instance, messages accumulate in the server's mailbox, consuming memory and degrading performance. In the second, there are no tell-tale signs at runtime.

On the surface, we cannot assign blame with absolute certainty. We do not know which — if any — of the communicating components are correctly implemented, and we do not even know if the resulting behaviour is the intended one.

Regardless of the ultimate answer to the question, an unresolved discrepancy remains in both modules. We argue that these discrepancies should be quickly and effectively reported to the programmer *during* editing, so they can take early action.

We have built a prototype tool in its very early stages [12] which serves as a foundation to detect errors similar to these, using the Core Erlang intermediate representation as our base [3]. A brief overview of critical parts of Core Erlang is given in Section 2.

Building on a lightweight subtyping system for Erlang messages as a proof-of-concept (Section 3), we present a definition of *message compatibility*, which determines whether a message will be received by a `receive` expression based entirely on its type (Section 4).

We then detail the type inference method used in our tool to assign types to `receive` expressions (Section 5.4) and sent messages (Section 5.5. This involves performing type inference on a subset of Core Erlang's pattern language in Section 5.1, and a subset of valid guard expressions in Section 5.2).

This is followed by a discussion of the pure Erlang implementation of our tool in Section 6. After, we discuss the kinds of analyses we perform: detection of potential orphan messages, identifying dead receive clauses, and calculating an upper bound on the number of clauses in a `receive` expression required to receive a given message (Section 7). We finish off with a comparison of our tool to related theory and other analysis tools for Erlang in Section 8, before considering the current and potential usefulness of the work in Section 9.

2 A quick tour of Core Erlang

Most Erlang programmers will never encounter Core Erlang during their time with the language. An exception to this is if they have ever invoked `erlc` with the `+to_core` flag:

```
erlc +to_core module.erl
```

This generates a file at `module.core` containing the Core Erlang representation of `module.erl`. This is possible as Core Erlang is an *intermediate representation* used in the Erlang/OTP compiler. These days, Erlang source code passes

```
1  eq_checker(X) ->
2      receive
3          {eq, M, M, From} ->
4              From ! true;
5          {eq, _M, _N, From} ->
6              From ! false;
7          {guess, X, From} ->
8              From ! you_got_it
9      end,
10     eq_checker(X).
```

(a) A function in Erlang

```
1  'eq_checker'/1 = fun (_@c0) ->
2    do
3      receive
4        <{'eq',M,_@c2,From}>
5          when call 'erlang':'=:=' (_@c2, M) ->
6            call 'erlang':'!' (From, 'true')
7        <{'eq',_X_M,_X_N,From}> when 'true' ->
8            call 'erlang':'!' (From, 'false')
9        <{'guess',_@c3,From}>
10         when call 'erlang':'=:=' (_@c3, _@c0) ->
11           call 'erlang':'!' (From, 'you_got_it')
12       after 'infinity' -> 'ok'
13     apply 'eq_checker'/1 (_@c0)
```

(b) The same function, in Core Erlang

Figure 3. Code compiled from Erlang to Core Erlang using `erlc`

through a representation in Core Erlang on its way to becoming BEAM bytecode.

This language has a specification which defines its syntax and semantics [4]. While it lacks several syntactic conveniences used by Erlang programmers, all Erlang programs can be represented in Core Erlang. Eager readers looking for a longer tour of the language and its features will enjoy [3] which introduces Core Erlang at greater length, while the specification at [4] documents the language in full.

Let's quickly explore this language with an illustrating example. The code in figure 3a shows an Erlang function which performs some communication, receiving messages and replying to them. We use pattern matching in the clauses, and send messages using the `!` operator. After the receive, we call the function again recursively.

After a quick invocation of the Erlang compiler on the code in figure 3a, the Core Erlang code in figure 3b is generated. Note that the Erlang/OTP compiler does not necessarily produce Core Erlang code which uses its full feature set, a fact we exploit in Section 6.

Core Erlang lacks several of Erlang's conveniences, while also making some analyses easier.

Clauses always have guards Guards must always be present on clauses, and they must evaluate to the atoms `'true'` or `'false'`.

When the Erlang compiler is compiling a clause without a guard, it inserts a trivially true one on its way to Core Erlang.

No operators Core Erlang doesn't have Erlang's convenient infix operators.

How do we call `!` and `=:=` then? We call the functions named `'!'` and `'=:='` in the ever-important `'erlang'` module.

No bound variables in patterns All variables in patterns must be free, and they are bound when the pattern is matched.

On line 7 of our Erlang source, we check whether the second element of the message tuple is equal to the already-bound variable X. In the compiled Core Erlang version, this is an equality check in the guard on line 10.

No repeated variables in patterns Apart from bit patterns, a variable may only occur once in a pattern.

The Erlang compiler has again compensated for this by pushing the work into the guard. The pattern on line 3 of the Erlang source has a repeated variable M, while in the compiled Core Erlang version on line 4, a new variable has been introduced, with the equality check taking place in the guard on line 5.

3 Type System

We have chosen a handful of types from Erlang: atoms, pids, booleans, tuples and unions. We believe that these form a representative example of Erlang's entire type system: we have some distinct primitive types, a subtype of a primitive type, and a compound type. A union of types (typically written $T_1|\ldots|T_n$ in Erlang) represents a choice between any of its elements, while the *any*() type represents the top type, and *none*() represents the bottom. We intend to extend our tool to support Erlang's full type system in the future.

The subtyping relation $\leq$ in figure 4 includes axioms of Erlang's type system: the hierarchy of the primitive types is axiomatised, while the *any*() type is placed at the top of the hierarchy, with all types derived from it. Although we can generally introduce compound types, we have only included tuples at this stage. Tuples with an undefined size (i.e. the universe of all tuples) are a subtype of *any*() (T-TupleAny). A tuple of a specific size n is a subtype of this via rule T-TupleSize, where the types of its elements must be within the universe of types. The notation *tuple*() denotes all tuples, while *tuple*([]) represents a tuple with zero elements. The

$$\frac{}{atom() \leq any()}\ \text{T-Atom} \qquad \frac{}{boolean() \leq atom()}\ \text{T-Boolean} \qquad \frac{}{pid() \leq any()}\ \text{T-Pid}$$

$$\frac{s : S \qquad S \leq T}{\langle s, S\rangle \leq T}\ \text{T-Singleton1} \qquad \frac{s : S \qquad s =:= t \qquad S \leq T}{\langle s, S\rangle \leq \langle t, T\rangle}\ \text{T-Singleton2} \qquad \frac{}{tuple() \leq any()}\ \text{T-TupleAny}$$

$$\frac{S_1 \leq any() \qquad \ldots \qquad S_n \leq any()}{tuple([S_1, \ldots, S_n]) \leq tuple()}\ \text{T-TupleSize} \qquad \frac{S_1 \leq T_1 \qquad \ldots \qquad S_n \leq T_n}{tuple([S_1, \ldots, S_n]) \leq tuple([T_1, \ldots, T_n])}\ \text{T-Tuple}$$

$$\frac{}{T \leq T}\ \text{T-Refl} \qquad \frac{S \leq S' \qquad S' \leq T}{S \leq T}\ \text{T-Trans}$$

Figure 4. Subtyping rules for a featherweight Erlang type system

subtyping relation distributes across tuples in T-Tuple, which maintains order and size.

Erlang's type system also admits singletons. The user-defined Erlang type $b()$ defines a singleton for the $atom()$ value `'foo'`:

```
-type b() :: 'foo'.
```

In our type system, we represent these singletons as a pair $\langle v, T\rangle$, where v is the Erlang literal, and T is the Erlang type of v ($v : T$) (T-Singleton1). We can derive an equivalent definition of $b()$ from $any()$ in our type system:

$$\frac{'foo' : atom() \qquad \overline{atom() \leq any()}}{\langle 'foo', atom()\rangle \leq any()}$$

Only singletons are subtypes of singletons, and we must preserve value equality. T-Singleton2 ensures that the subtyping relation is preserved on the tagged types, while the use of Erlang's `=:=` operator is used to ensure the value of the singleton is preserved across type boundaries.

Finally, we define the complement of a type T to be all types *except* T and its subtypes. Formally:

$$\overline{T} = \bigcup \{x \mid x \leq any() \wedge x \not\leq T\}$$

The subtyping relation forms a preorder, and we introduce type unions and intersections in figure 5, which distribute over one another [2]. Intersection and union also distribute across compound types: the elements of a tuple are compared element-wise.

4 Message Compatibility

Our analysis relies on the principle of *compatible messages*. A message is compatible with a `receive` expression if that `receive` expression is capable of selecting a clause and removing that message from the mailbox.

In order for a message to be compatible with a `receive` expression, we know:

1. A pattern match has been successful for a `receive` clause; and
2. The guard associated with that clause (if any) has evaluated to `'true'`

If we consider the type of the message, we also know, for the same clause:

1. Any "structure" of the message matches the "structure" of the pattern
2. Some type assertions in the guard may have returned `'true'`

Specifically, "structure" here refers to the fact that Erlang patterns can be used to assert compound type constraints. For example, the pattern in this `receive` expression asserts that the message is a tuple with two elements:

```
receive
    {A,B} -> ok
end
```

Furthermore, we can infer additional type information about a message from the guards associated with the clause. Any call to a function in the `erlang` module of the form `is_type(A)` (where `type` is the name of an Erlang type, and `A` is a variable which occurs in the pattern) can be seen as a type assertion on the message. For example, this block asserts that the message's type is a tuple with two elements, *and* that the second element is an atom:

```
receive
    {A,B} when is_atom(B) -> ok
end
```

Beyond types, patterns and guards may also restrict the values of the messages they accept. This receive expression only accepts messages with two elements, where the first is the atom `'ok'`, and the second is a pid:

```
receive
    {'ok',B} when is_pid(B) -> ok
end
```

It is essential to consider all the clauses of a `receive` together when determining message compatibility, due to the way clauses interact. We cannot consider clauses in isolation.

$$\frac{S \leq U \qquad T \leq U}{S \cup T \leq U} \qquad \frac{S \leq T}{S \leq T \cup U} \qquad \frac{S \leq U}{S \leq T \cup U}$$

$$\frac{S_1 \cup T_1 \leq U_1 \qquad \ldots \qquad S_n \cup T_n \leq U_n}{tuple([S_1, \ldots, S_n]) \cup tuple([T_1, \ldots, T_n]) \leq tuple([U_1, \ldots, U_n])}$$

(a) Union

$$\frac{S \leq T \qquad S \leq U}{S \leq T \cap U} \qquad \frac{S \leq U}{S \cap T \leq U} \qquad \frac{T \leq U}{S \cap T \leq U}$$

$$\frac{S_1 \leq T_1 \cap U_1 \qquad \ldots \qquad S_n \leq T_n \cap U_n}{tuple([S_1, \ldots, S_n]) \leq tuple([T_1, \ldots, T_n]) \cap tuple([U_1, \ldots, U_n])}$$

(b) Intersection

Figure 5. Binary operations on types

Instead, we must take the union of all constituent clause types:

```
receive
    A when is_pid(A) -> ok;
    A when not is_pid(A) -> ok
end
```

While the first clause accepts messages of type $pid()$, and the second accepts messages of type $\overline{pid()}$, the union of these types when the `receive` expression is considered as a whole is $any()$.

Ultimately, a combination of the pattern and guard in a clause can be used to determine the type and value of messages it will accept. Furthermore, we must consider all clauses together to determine the correct type. This leads us to the definition of message compatibility:

Definition 4.1. A message m is **compatible** with a `receive` expression r when:

$$T_m \leq T_r$$

where T_m is the type of the message (Section 5.5), and T_r is the type of the receive (Section 5.4).

5 Type Inference

Erlang source code is compiled to Core Erlang before type inference is performed. We perform our type inference on the Core Erlang AST, not the original Erlang source. In addition to the points explored in Section 2, the Erlang Compiler also performs some checks and optimisations for us. While these are not guaranteed by the Core Erlang specification, they are known behaviours of the Erlang compiler:

- some constant folding has been performed[1]; and
- preliminary dead code detection has taken place, generating warnings

One slight annoyance of this compilation is that Erlang's short-circuiting `andalso` and `orelse` operators are not represented. As Core Erlang is call-by-value, these short-circuiting operators are compiled into simple `case` expressions. The

```
message_info() ->
  #{{'client',1} =>
    #{'Server' => {'server',0}}}
```

Figure 6. Destination map example currently required by our tool

left-hand side of the boolean operation is placed in the argument of the `case`, while the `'true'` and `'false'` branches are populated according to the truth tables of the boolean operation via a dedicated function[2] in the `v3_core` compiler module [9].

Each function definition is traversed to find its `receive` expressions which are translated into types according to Section 5.4, and the process for translating calls to the `send` function is described in Section 5.5.

Unfortunately, our tool does not have any data or control flow analyses yet. This means that we cannot infer the destination of sent messages automatically. As an intermediate solution, we currently use a special Erlang map, which the user must provide in the `message_info/0` function in the same module as the functions under analysis.

The top level of the map contains function name and arity pairs as keys, which correspond to functions defined in that module. The associated values of these keys themselves contain another map. This inner map describes, for the given function, which variable names (keys) refer to processes executing the code which other functions (values).

An example of the function is shown in figure 6. It states that in the body of function `client/1`, the variable `Server` refers to a process which is executing `server/0`.

5.1 Pattern Translation

A clause's pattern contains type information, notably about the structure of compound types: we can determine the size of tuples and lists from the pattern alone.

We perform type inference over a fragment of Core Erlang's pattern syntax: variable names, pattern aliases, literals, and tuples. We do not perform type inference for lists (which

[1] `v3_core_opt`

[2] `make_bool_switch_guard`

$$P[\![V]\!] = any() \quad \text{(PVar)}$$
$$P[\![e = V]\!] = P[\![e]\!] \quad \text{(PAlias)}$$
$$P[\![t]\!] = \langle t, T\rangle \quad \text{where } t : T \quad \text{(PLiteral)}$$
$$P[\![\{e_1, \ldots, e_n\}]\!] = tuple([P[\![e_1]\!], \ldots, P[\![e_n]\!]]) \quad \text{(PTuple)}$$

Figure 7. Typing rules for Core Erlang patterns

are formed of cons cells), or bitstrings (which may contain repeated variables).

For this fragment of patterns, we define the translation function P in Figure 7. This function recursively infers the type of a Core Erlang pattern.

Any occurrence of a variable in a pattern is inferred to have the *any*() type, since it will freely bind with any value during pattern matching. The variable in a pattern alias is ignored for the same reason, while processing of the aliased pattern continues. This is only true for Core Erlang patterns: variables in patterns *always* bind freely, unlike Erlang.

The occurrence of an atomic (i.e. not compound) literal in a pattern is assigned a singleton type as described in Section 3. The tagged type for this singleton is the Erlang type of the term. The atom `'ok'` in a pattern has the inferred type $\langle 'ok', atom()\rangle$.

Compound literals (like a tuple consisting entirely of atoms) have singleton types in their atomic elements, not at their root. For example:

$$P[\![\{'a','b'\}]\!] = tuple([\langle 'a', atom()\rangle, \langle 'b', atom()\rangle])$$
$$P[\![\{'a','b'\}]\!] \neq \langle \{a, b\}, tuple(atom(), atom())\rangle$$

Finally, tuples are inferred to have the *tuple*() type; the types of its elements are inferred recursively.

The function P *cannot* infer the type *none*() for any pattern in our fragment of the syntax. Without repeated variables, it is impossible to construct a pattern which will not match at least one value.

5.2 Guard Translation

We have modelled a fragment of Core Erlang's guard syntax, which includes support for checking the types described in Section 3. While Core Erlang syntactically permits arbitrary expressions within guards, the specification [4] restricts the permitted expressions, and the Erlang compiler [9] is known to currently generate specific syntax.

As such, we support logical conjunction via and, disjunction via or, and calls to type test BIFs where the sole argument is the name of a variable *which occurs in the pattern*. Guards pass when they evaluate to `'true'`, and fail when they evaluate to `'false'`.

The function G in Figure 8 performs type inference for guards of this form. The guard `'true'` is inferred to have

$$G[\![\texttt{'true'}]\!]_\rho = any() \quad \text{(GTrue)}$$
$$G[\![\texttt{'false'}]\!]_\rho = none() \quad \text{(GFalse)}$$
$$G[\![g_1 \texttt{ and } g_2]\!]_\rho = G[\![g_1]\!]_\rho \cap G[\![g_2]\!]_\rho \quad \text{(GAnd)}$$
$$G[\![g_1 \texttt{ or } g_2]\!]_\rho = G[\![g_1]\!]_\rho \cup G[\![g_2]\!]_\rho \quad \text{(GOr)}$$
$$G[\![is\_T(V)]\!]_\rho = \rho\ V\ T \quad \text{(GTest)}$$
$$G\left[\!\!\left[\begin{array}{l}\texttt{case } g_1 \texttt{ of}\\ \texttt{'true'} \rightarrow g2\\ \texttt{'false'} \rightarrow g3\end{array}\right]\!\!\right]_\rho = \begin{array}{l}(G[\![g_1]\!]_\rho \cap G[\![g_2]\!]_\rho) \cup\\ \overline{(G[\![g_1]\!]_\rho \cap G[\![g_3]\!]_\rho)}\end{array} \quad \text{(GCase)}$$

Figure 8. Guard type inference rules

$$vmap(f, v) = [v \mapsto f] \quad (1)$$
$$vmap(f, t) = [\,] \quad (2)$$
$$vmap(f, p = V) = [V \mapsto f] + vmap(f, p) \quad (3)$$
$$vmap(f, \{p_1, \ldots, p_n\}) = vmap(\lambda x.tuple([fx, \ldots, any()]), p_1) + \cdots + vmap(\lambda x.tuple([any(), \ldots, fx)], p_n) \quad (4)$$

Figure 9. Variable mapping function for variables in guards

type *any*(), since it will always pass, and `'false'` is inferred to have type *none*(), since it will always fail.

Logical conjunction calculates the intersection of the types inferred for each operand, since the value being matched must satisfy *both* types. Disjunction calculates the union, as the value must match *at least one* type. Negation is simply the complement of the type inferred for its operand, as it matches all values that its operand does not.

A call to a type test BIF requires more consideration. All of these type test BIFs have similar names of the form `is_name`, where `name` is the name of an Erlang type. Looking at the following `receive` expression, we can see that the guard asserts that the second element of the received message is an atom:

```
receive
  {A, B} when is_atom(B) -> ok
end
```

But when we consider the guard in isolation, we cannot determine what the variable B is really referring to: the second element of a top-level tuple. To solve this problem we introduce a mapping ρ which has the type:

$$\rho : vname \rightarrow (T \rightarrow T)$$

This function can be seen as an environment which maps variable names to functions which consume one type to produce another. When a call to a type test BIF is encountered

(where its argument is a variable which occurs in the associated pattern), we want to *project* the type assertion to the correct *position* within the compound datatype of the pattern.

In our example, we would like:

$$\rho\ B\ atom() = tuple([any(), atom()])$$

which is a type asserting that the second element of a two-element tuple is $atom()$. Projecting types in guards in this way allows us to easily unify the types of guards and patterns via intersection as the type of the guard will contain types in the correct "positions" for compound type variables which occur in the pattern.

These environments are generated using the function $vmap$ in figure 9. The first argument is a function, and the second is a pattern. For a pattern, the first call to the function uses the identity function $\lambda x.x$ as its first argument. Again referring to our example:

$$\begin{aligned} vmap(\lambda x.x, \{A, B\}) &= vmap(\lambda x.tuple([(\lambda x.x)\ x, any()]), A) + \\ &\quad vmap(\lambda x.tuple([any(), (\lambda x.x)\ x)], B) \\ &= [A \mapsto (\lambda x.tuple([(\lambda x.x)\ x, any()]))] + \\ &\quad [B \mapsto (\lambda x.tuple([any(), (\lambda x.x)\ x]))] \end{aligned}$$

and with $\rho = vmap(\lambda x.x, \{A, B\})$, the application of the guard translation function:

$$\begin{aligned} G[\![is\_atom(B)]\!]_\rho &= \rho\ B\ atom() \\ &= tuple([any(), atom()]) \end{aligned}$$

maps the $atom()$ type from the `is_atom` call to the second element of the tuple.

When this type is intersected with the type of the pattern, we get:

$$\begin{aligned} &= P[\![\{A, B\}]\!] \cap G[\![\texttt{is\_atom}(B)]\!]_\rho \\ &= tuple([any(), any()]) \cap tuple([any(), atom()]) \\ &= tuple([any(), atom()]) \end{aligned}$$

as expected.

5.3 Short-Circuiting Boolean Operators

Although the `andalso` and `orelse` operators are erased in Core Erlang, we can easily infer a type for them in the context of guards.

These short-circuiting operators can be used to control side-effects, for example sending a shutdown messsage to an imaginary actor only if it is running:

```
safe_shutdown() ->
  alive(X) andalso send_shutdown_msg(X).
```

Conveniently, however, the function calls permitted in Erlang guards are restricted, intended to avoid side effects. Furthermore, the Core Erlang specification states in §6.7 that *guards must not have observable side-effects* [4].

The lack of side effects in all `case` expressions in Core Erlang guards ensures that we can derive the correct types via the GCase rule:

Lemma 5.1. *Assuming the Erlang keywords* `and` *and* `andalso` *are semantically equivalent in the side-effect free context of guards, then for all guards g_1, g_2 and environments ρ:*

$$G[\![g_1\ \texttt{and}\ g_2]\!]_\rho = G[\![g_1\ \texttt{andalso}\ g_2]\!]_\rho$$

Proof. Assume $G[\![g_1]\!]_\rho = T_1$ and $G[\![g_2]\!]_\rho = T_2$, then:

$$G[\![g_1\ \texttt{and}\ g_2]\!]_\rho = T_1 \cap T_2$$

The Erlang expression g_1 `andalso` g_2 is compiled to the following Core Erlang case expression:

```
case g1 of
    'true' -> g2
    'false' -> false
```

which has the type:

$$\begin{aligned} G\left[\!\!\left[\begin{array}{l}\texttt{case}\ g_1\ \texttt{of} \\ \quad \texttt{'true'} \rightarrow g_2 \\ \quad \texttt{'false'} \rightarrow false\end{array}\right]\!\!\right]_\rho &= (G[\![g_1]\!]_\rho \cap G[\![g_2]\!]_\rho) \cup \\ &\quad \overline{(G[\![g_1]\!]_\rho} \cap G[\![\texttt{'false'}]\!]_\rho) \\ &= (T_1 \cap T_2) \cup (\overline{T_1} \cap none()) \\ &= (T_1 \cap T_2) \cup none() \\ &= T_1 \cap T_2 \end{aligned}$$

□

A similar trivial proof exists for `or` and `orelse`.

5.4 Receive Translation

A Core Erlang clause (be it in a `receive` or `case expression`) has the general form:

$$< p_1, \ldots, p_n >\ \texttt{when}\ e_1\ \texttt{->}\ e_2$$

where p_1 through p_n are patterns, e_1 is the guard expression, and e_2 is the body. It is an error for any clause in a Core Erlang `receive` expression to contain more than one pattern, so the clauses can equivalently be written as:

$$p\ \texttt{when}\ e_1\ \texttt{->}\ e_2$$

The type of a Core Erlang `receive` clause is the intersection of the pattern type from figure 7 and guard type from figure 8:

$$CL[\![p\ \texttt{when}\ e_1\ \texttt{->}\ e_2]\!] = P[\![p]\!] \cap G[\![e_1]\!]_{vmap(p, \lambda x.x)}$$

Bringing all of this together, we can now easily infer types for *entire* `receive` expressions. The type for each constituent clause is inferred, and then the union of all clause types is

used as the type of the `receive` expression:

$$R\left[\!\!\left[\begin{array}{l}\texttt{receive}\\ \quad Clause_1\\ \quad \ldots\\ \quad Clause_n\\ \texttt{after}\ e_1\ \texttt{->}\ e_2\end{array}\right]\!\!\right] = \begin{array}{l} CL[\![Clause_1]\!]\cup\\ \cdots\cup\\ CL[\![Clause_n]\!]\end{array}$$

Therefore, the following `receive` expression:

```
receive
  X when is_atom(X) -> e1
  X when is_pid(X) -> e2
end
```

is inferred to have type $atom() \cup pid()$, as the union of the clause types.

5.5 Send Translation

The Core Erlang AST of the module is traversed again, this time to find calls to `'erlang':'!'/2`, which is the primary function for sending messages in Erlang/OTP. Although we are aware of other send functions[3], we do not identify or analyse them in the current version of the prototype. This means we identify sends of the form:

```
Variable ! Message
```

which is a call to the `erlang:'!'/2` function when desugared.

First, the name of the destination is looked up in the destination map (Section 5) for the function undergoing translation. If the destination is not a key in the map, an exception is thrown, aborting translation of the send.

The type of the message itself is determined using the function P in Figure 7, the same as is used for patterns. The structure of the message is used to determine an *upper bound* for its type: all occurrences of variables in the message body are inferred to have the $any()$ type. This ultimately results in an over-approximation of the message type:

```
f(X, Dest) when is_atom(X) ->
  Dest ! X.
```

It is obvious here that the variable X has type $atom()$, even though we assign it the type $any()$ due to our lack of data-flow analysis at this early stage.

The result of this translation is a tuple $(Dest, P[\![X]\!])$ of the message destination and the inferred type of its contents. This is repeated for all sends in the body of the function. For a function f:

$$sends(f) = \left\{(Dest,\ P[\![X]\!]) \;\middle|\; \texttt{call 'erlang':'!'}\ X \in f\right\}$$

[3] `send_after/3`, `send_after/4`, `send_nosuspend/2`, `send_nosuspend/3`

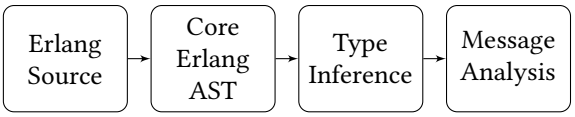

Figure 10. Analysis Process

6 Implementation

The translation and analysis has been implemented entirely in Erlang, available on the command line via use of `escript`. It has no non-Erlang dependencies, mostly relying on the compiler libraries distributed with Erlang/OTP. The whole analysis process is shown in figure 10.

The representation of Core Erlang syntax trees inside the Erlang/OTP compiler could be considered unstable, and is arguably not intended for end-user consumption. With the intention of increasing maintainability, we do not manipulate the ASTs directly. Instead, we use the compiler's `cerl` module to handle them, which presents a more stable API which performs manipulation and extraction on our behalf.

Our tool is invoked with a single argument, the name of an Erlang source file for analysis:

```
./pdis module.erl
```

The Erlang source code is compiled to Core Erlang using `compile:file`, using the `to_core` option, among others[4]. The Erlang compiler also does some heavy lifting for us at this stage, notably constant folding and some basic dead code analysis.

At each stage of the translation processes described in Section 5, annotations are added where necessary: the source location is copied from the Core Erlang AST node, and the AST node itself is added to the inferred type. This provides a rich infrastructure for reporting errors at the expense of a small amount of memory.

For each function definition in the AST, the sends and `receives` are translated and their results stored in a map of function names to sends/`receives`.

If, during the translation of guards, an intermediate or final type of $none()$ is inferred, an exception is immediately thrown, and the clause is excluded. In practice, an inferred type of $none()$ usually indicates contradictory or unsatisfiable guards. The exception is reported to the user as an error message.

6.1 Compilation of Guard Expressions

The Erlang compiler [9] sometimes performs transformations to guard expressions which introduce new variables, function calls, and blocks of syntax (Section 2). Some of these transformations unintentionally obfuscate the apparent meaning of guards, and complicate our type inference. To simplify the analysis of guards, we convert them into a "normal form" beforehand: specific `try`/`catch` expressions

[4] `[binary,bin_opt_info,debug_info,report,return,verbose]`

are eliminated when they are deemed unnecessary for type inference, `erlc`'s boolean coercion in guards is removed, and all variable bindings are substituted.

Errors The Erlang reference manual [8] states that when evaluation of a guard expression fails, the whole guard fails. The Core Erlang specification, however, states that an exception raised during evaluation of a guard causes the outer expression to abruptly complete with that exception. To preserve the behaviour described in the Erlang reference manual, the compiler sometimes wraps guard expressions in a `try`/`catch` expression, of which only one form is permitted inside Core Erlang guards. We detect these blocks and remove them, replacing them with the original sub-expression.

Boolean Coercion The behaviour of Core Erlang guards is undefined when evaluation results in values other than `'true'` or `'false'`. The Erlang compiler therefore introduces an equality check when it is unsure of a guard's return value: if the compiler cannot guarantee that $g1$ will evaluate to a boolean atom, it will wrap it with a call to `=:=` of the form $g1$ `=:= 'true'`. Luckily, this call is also marked as compiler-generated, so we check (and remove) these boolean coercions.

Variable Bindings Guards which contain nested boolean operations (e.g. chained uses of conjunction and disjunction) are often compiled out into a normal form where both arguments are variable names. This is achieved through abstraction. A guard of the form g_1 and g_2 may compile to:

```
let _@c0 = g1 in let _@c1 = g2 in
↪  erlang:and(_@c0,_@c1)
```

Interestingly, these `let` bindings are not tagged as compiler-generated. Regardless, we substitute all variables in guards for simplicity. This leaves us with a guard where all variables either refer to the pattern, or some outer scope.

This operation is generally unsafe in a call-by-value language such as Core Erlang, as this could cause duplicated side-effects. Guards, however, are side-effect free, so we believe the translated form is semantically equivalent.

7 Results

The preliminary version of the tool is capable of detecting some simple errors in Erlang programs. The tool can detect some contradictory guard expressions, orphan messages, and can also determine an upper bound for the number of clauses in a `receive` required to satisfy the type of a given message. The tool is currently quite fragile, supporting only a portion of the Core Erlang syntax. We have run the tool on many hand-crafted examples, including the examples from this manuscript.

7.1 Impossible Guards

The tool can detect impossible guards and clauses in much the same way as Dialyzer. If we consider the Erlang source

```
1 -module(imposs).
2 -export([f/0]).
3
4 f() ->
5  receive
6   X when is_atom(X), is_pid(X) -> ok
7  end.
```

Figure 11. A receive clause with impossible guards

in figure 11, we can see on line 6 that the clause can never match: no value can be both an atom and pid simultaneously:

Running our tool on the above Erlang source file produces the following warnings:

```
imposs.erl:6:WARN: This clause will never
↪  match as it has the type none():
  X when is_atom(X), is_pid(X) -> ok

imposs.erl:6:WARN: cannot intersect types
↪  atom() and pid() in this guard:
  X when is_atom(X), is_pid(X) -> ok
```

while the (trimmed) output from Dialyzer shows a similar message to ours:

```
imposs.erl:4: Function f/0 has no local return
imposs.erl:6: Guard test is_pid(X::atom()) can
↪  never succeed
```

7.2 Orphan Message Detection

The source code translation has provided us with an inferred type for each send and `receive` operation in a Core Erlang module.

Each function in a module is considered individually. For that function, each send operation is checked: The destination of the message is looked up in the map in `message_info`. If the destination is undefined, or the function it refers to does not exist, analysis is skipped and a warning is generated. For sends with a valid destination, each `receive`type in the destination is checked to determine whether the content of the sent message is compatible (Section 4). If the message is compatible, the message is marked as *non-orphan*. If no compatible `receive` is found, the message is marked as *potentially orphan*.

For each message that is marked *potentially orphan*, a warning is generated. We currently generate warnings even where an orphan message might not exist due to the over-approximation of variable types in messages (Section 5.1). This is why we consider these messages to be *potentially* orphan.

Definition 7.1. Assume that a module containing the function f has been analysed as described in Section 5. This analysis has yielded a set of sends $sends(f)$ of the form $(dest, T_s)$, where $dest$ is the name of the message destination as per

the `message_info` map, and T_s is the inferred type of the message.

Also assume that *dest* is a function defined in the module, and that $receives(dest)$ is the set of inferred *receive types* for its body.

Then, we can say that the function f is known not to produce any orphan messages if:

$$\forall(dest, T_s) \in sends(f).\ \exists T_r \in receives(dest).\ T_s \leq T_r$$

That is to say that the function f does not send orphan messages if, for all sends in its body, the type of a receive clause in the destination function is a subtype of the sent message. This means that a clause exists where we know the pattern will match, and that the guard will evaluate to `true`.

In figure 1 we showed an example of an orphan message on line 10. When we pass the source file through our tool, we are given the following warning relating to an orphan message:

```
example1.erl:10:WARN: Send never received in
↪  destination (orphan message?):
  From ! ack,
```

This warning is produced as no `receive` clause in the client can be found which satisfies the type $\langle$`'ack'`, $atom()\rangle$, which is a bare atom of value `'ack'`.

7.3 Upper bound on receive clauses

When a send is matched against a `receive`, we can also determine how many clauses were required to obtain this match. Starting with only the first clause, we take a union of an increasing number of clause types until the message is a subtype of the union. At this point, we have an upper bound on the number of clauses in the `receive` which are required to match the message.

Again taking the example in figure 1, we get the following output from our tool:

```
example1.erl:11:OK: Send received on line 23
↪  in 1 clauses or less:
From ! {ack, N},

example1.erl:21:OK: Send received on line 10
↪  in 1 clauses or less:
Server ! {get, self()},

example1.erl:25:OK: Send received on line 13
↪  in 2 clauses or less:
Server ! {set, N-1, self()},
```

Note that when the tool's output says "in n clauses or less", the tool was able to infer that the union of the types of the first n `receive` clauses is the first n for which the sent message is a subtype.

```
1  server() ->
2    receive
3      start ->
4        receive
5          {A,B} -> f(A,B)
6        end
7    end,
8    server().
9
10 client(Server) ->
11   Server ! {some,atoms}.
```

Figure 12. An Erlang program with a message compatibility dependency

7.4 Variable Renaming

One pain point of the compilation to Core Erlang is variable renaming. The Erlang compiler renames variables at will, most notably those which we rely on in our destination map described in Section 5. Currently, the tool is susceptible to these renamings, and we lose track of the original variable name.

Some engineering is required to overcome this, but it is not a blocking issue. It is feasible to build a mapping to track variable renaming, as we have both access to the compiler and space in the ASTs for annotations.

8 Related Work

With regards to static analysis, our tool is most comparable to Dialyzer, which can automatically detect type discrepancies (among others). Indeed, Dialyzer can detect impossible type checks like those in figure 11 through its use of success typings [14] and an internal constraint system:

```
imposs.erl:6: Guard test is_pid(X::atom())
↪  can never succeed
```

Its inclusion with Erlang/OTP is advantageous from an adoption perspective, with users not required to download any additional software. The generation of a PLT on first use, however, can prove lengthy. The success typing system in Dialyzer is only one example of types for Erlang, though: previous work has explored soft typing [17], a subtyping system for a larger portion of the language [15], and current work is exploring a more pay-as-you-go subtyping system [19].

The work of Christakis and Sagonas extended Dialyzer to verify similar safety properties of communications at the Core Erlang level, using its success type system to detect orphan messages, unused receives, and more [5].

Dialyzer is much more complete than our prototype: we do not have nearly as much coverage of the Core Erlang syntax, and its extensive infrastructure allows it to detect many classes of type errors automatically.

From a theoretical perspective, the analysis currently does not consider the ordering and potential dependencies of messages. The compatibility of one message might ultimately depend on the compatibility of another. Figure 12 shows a program which has such a dependency. The client is incompatible with the server: The server is only compatible with tuples of size 2 when it is *also* compatible with the atom `start`. Adding a send of that atom *anywhere* in the system will make this server compatible with tuples of size 2: any process in the system can send the message *at any time*.

This property is arguably closer in spirit to a protocol or specification seen in session type systems. Moreover, it is akin to a model of the system, perhaps being more suited to existing Erlang model checking tools.

It may prove fruitful to integrate with Dialyzer's analyses in the future, in a manner similar to Christakis and Sagonas. Several years of engineering and integration with Erlang/OTP has made Dialyzer a useful static analysis tool for the functional components of Erlang.

In a similar vein to type specifications for Erlang functions, we will introduce communication specifications for functions, too. We do not intend to make these full protocol specifications as typically seen in session type systems. Instead, we want to enable the user to provide lightweight and ad-hoc specifications for their communicating components, with a potential extension to OTP behaviours such as `gen_server` and the newer `gen_statem`.

We believe that this ad-hoc approach to communication specification will yield the most useful tool: Erlang systems are inherently ad-hoc, with independent and distributed applications of different origin communicating freely.

As the tool targets Core Erlang, it is also feasible to integrate with Elixir and other languages which use it as an IR. Although the analysis will operate in the same way, the quality of the tool output will rely on the presence of AST annotations which relate back to the original source code.

Model checking is another approach which has been widely used to verify safety properties of Erlang programs. Soter [6] uses a combination of static analysis and infinite state model checking for a large fragment of Erlang. Usefully, the tool is also available for testing via a web interface, with support for visualisation of the intermediate representations used within [7].

Concuerror [11] is a stateless model checker for Erlang programs. It systematically tests Erlang programs using process interleaving to automatically detect errors. The authors of Concuerror compare their work to McErlang [10], a model checker with full support for the type system. That tool replaces part of the Erlang runtime to simulate processes.

Session types have been successfully applied to Erlang systems, too. In a true following of Erlang's philosophies, session types have been used to recover misbehaving OTP components at runtime via the use of a user-provided protocol specification [16]. The detectEr tool also performs runtime verification of Erlang systems, synthesising monitors to check formal correctness properties at runtime [1].

Automatic tools like Godel [13] and Dialyzer are useful: the tools operate automatically, without requiring the developer to create large or unwieldy specifications for their system. Dialyzer has the added benefit of allowing the programmer to specify types in an ad-hoc manner, annotating only the functions they want to.

9 Discussion & Future Work

Our tool is in its early stages, but it is already making progress towards detecting orhpan messages and dead `receive` clauses. We can currently identify *potential* orphan messages and *possibly* dead `receive` clauses. At the Core Erlang level, our type inference system over-approximates the type of messages due to our lack of data and control flow analyses.

We believe this places the tool in an advisory role during the development process. As we can only provide an approximation with the current prototype, integration of the tool with a development environment would allow us to direct programmers to the sites of *potential* communications problems in their code. Going forwards, we must be careful not to drown the programmer in excessive warnings.

At the moment, the tool considers modules in isolation. All of the examples we provided show an intra-module analysis. Ultimately, this hinders our ability to identify *definitely* dead `receive` clauses: other modules or applications may bring the clause to life. We plan to embed message and receive type metadata inside compiled Erlang modules so that the interaction *between* modules can be considered. This has the potential to be done at either at compile time (by considering the source of an entire release at once), or at load time (by instrumenting the code server). The same principle of message compatibility applies in an inter-module context.

The keen reader will also notice the lack of types for `not` in guards. This is due to the difficulty of defining the sub-typing relation over union types, intersection types, and product types *in the presence of negation types*. We are currently using the work of [18] to implement negation types symbolically, as it is not possible to represent them concretely.

Next, we will focus on basic data and control flow analyses to improve our type inference on messages, so that we can produce a better approximation. At the moment, inferring the type *any*() for variables leads to a significant *over*-approximation of the real type. Furthermore, we are ditching the variable-to-destination mapping requirement as quickly as possible in favour of more automatic methods, as it is cumbersome and fragile.

In the spirit of other type inference systems which permit *optional* type annotations, we are also developing an ad-hoc specification language for send and receive types. These

specifications are likely to be less expressive than full protocols (like in session type systems), but should prove useful in allowing the programmer to specify the communicating behaviour of their Erlang code.

Ultimately, the tool can analyse a fragment of Core Erlang's syntax to produce approximate types for sent and received messages, being able to identify potential communication discrepancies at compile time, written purely in Erlang, without deferring to external solvers. We believe our concept of *message compatibility* serves as a good lightweight method of comparing sends and receives, and we are working to extend the tool to take full advantage of it.

Acknowledgments

The author would like to thank his PhD supervisor Professor Simon J. Thompson for his continued support in this research, and additional thanks to the three reviewers of this manuscript who greatly helped in its presentation and discussion.

This work is supported by an EPSRC studentship. Grant No. 1647970.

References

[1] Duncan Paul Attard, Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. 2017. *Behavioural Types: From Theory to Tools*. River Publishers, Chapter 3, 49–76.

[2] F. Barbanera, M. Dezaniciancaglini, and U. Deliguoro. 1995. Intersection and Union Types: Syntax and Semantics. *Information and Computation* 119, 2 (1995), 202 – 230. https://doi.org/10.1006/inco.1995.1086

[3] Richard Carlsson. 2001. An introduction to Core Erlang. In *PLI'01 Erlang Workshop*.

[4] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. 2004. *Core Erlang 1.0.3 language specification*. Technical Report. Uppsala University.

[5] Maria Christakis and Konstantinos Sagonas. 2011. Detection of Asynchronous Message Passing Errors Using Static Analysis. In *Practical Aspects of Declarative Languages*. Springer, Berlin, Heidelberg, 5–18.

[6] Emanuele D'Osualdo, Jonathan Kochems, and C. H. Luke Ong. 2013. Automatic Verification of Erlang-Style Concurrency. In *Static Analysis*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 454–476.

[7] Emanuele D'Osualdo, Jonathan Kochems, and Luke Ong. 2013. SOTER - Safety verifier fOr The ERlang language. https://mjolnir.cs.ox.ac.uk/soter/

[8] Erlang/OTP Team. 2018. *Erlang Reference Manual*. Ericsson AB. v9.3, Chapter: Expressions.

[9] Erlang/OTP Team. 2018. Erlang/OTP. http://erlang.org/download/otp_src_20.3.tar.gz v20.3.

[10] Lars 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] Alkis Gotovos, Maria Christakis, and Konstantinos Sagonas. 2011. Test-driven Development of Concurrent Programs Using Concuerror. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang (Erlang '11)*. ACM, New York, NY, USA, 51–61. https://doi.org/10.1145/2034654.2034664

[12] Joe Harrison. 2018. sigwinch28/pdis: Erlang'18 artifact. https://doi.org/10.5281/zenodo.1318356

[13] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A Static Verification Framework for Message Passing in Go Using Behavioural Types. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1137–1148. https://doi.org/10.1145/3180155.3180157

[14] Tobias Lindahl and Konstantinos Sagonas. 2006. Practical Type Inference Based on Success Typings. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '06)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/1140335.1140356

[15] Simon Marlow and Philip Wadler. 1997. A Practical Subtyping System for Erlang. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 136–149. https://doi.org/10.1145/258948.258962

[16] Rumyana Neykova and Nobuko Yoshida. 2017. Let It Recover: Multiparty Protocol-induced Recovery. In *Proceedings of the 26th International Conference on Compiler Construction (CC 2017)*. ACM, New York, NY, USA, 98–108. https://doi.org/10.1145/3033019.3033031

[17] Sven-Olof Nyström. 2003. A Soft-typing System for Erlang. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang (ERLANG '03)*. ACM, New York, NY, USA, 56–71. https://doi.org/10.1145/940880.940888

[18] David J. Pearce. 2013. Sound and Complete Flow Typing with Unions, Intersections and Negations. In *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 7737 (VMCAI 2013)*. Springer-Verlag, Berlin, Heidelberg, 335–354. https://doi.org/10.1007/978-3-642-35873-9_21

[19] Josef Svenningsson. 2018. A gradual type system. (06 2018). Talk at Code BEAM STO.