A Proof Assistant Based Formalisation of a Subset of Sequential Core Erlang

Péter Bereczky[1](✉), Dániel Horpácsi[1], and Simon Thompson[1,2]

[1] Eötvös Loránd University, Budapest, Hungary
berpeti@inf.elte.hu, daniel-h@elte.hu
[2] University of Kent, Canterbury, UK
S.J.Thompson@kent.ac.uk

Abstract. We present a proof-assistant-based formalisation of a subset of Erlang, intended to serve as a base for proving refactorings correct. After discussing how we reused concepts from related work, we show the syntax and semantics of our formal description, including the abstractions involved (e.g. the concept of a closure). We also present essential properties of the formalisation (e.g. determinism) along with the summary of their machine-checked proofs. Finally, we prove expression pattern equivalences which can be interpreted as simple local refactorings.

Keywords: Erlang formalisation · Formal semantics · Machine-checked formalisation · Operational semantics · Term rewrite system · Coq

1 Introduction

There are a number of language processors, development and refactoring tools for mainstream languages, but most of these tools are not theoretically well-founded: they lack a mathematically precise description of what they do to the source code. In particular, refactoring tools are expected to change programs without affecting their behaviour, but in practice, this property is typically verified by regression testing alone. Higher assurance can be achieved by making a formal argument – a proof – about this property, but neither programming languages nor program transformations are easily formalised.

When arguing about behaviour-preservation of program refactoring, we argue about program semantics. To be able to do this in a precise manner, we need a formal, mathematical definition of the semantics in question, on which to base formal verification. Unfortunately, most programming languages lack fully formal definitions, which makes it challenging to deal with them in formal ways.

Erlang, similarly to its younger siblings like Elixir and LFE, is having its renaissance in implementing instant messaging, e-commerce and fintech. Extensive code bases therefore need to be developed and maintained in these languages, which in turn requires refactoring support. Our work aims to improve trustworthiness of Erlang refactorings via formal verification, and in particular we formalise Core Erlang, which is not only a subset of Erlang, but also the target for translation of Erlang and Elixir in the compiler front-end. The formalisation of Core Erlang can also be seen as a stepping stone toward a definition for the entire Erlang language.

This paper presents the Coq [19] formalisation of a big-step semantics for a subset of sequential Core Erlang. In building this we rely not only on the language specification and the reference implementation, but also on some earlier work on semantics. Using this we made a definition of a semantics that can be properly embedded in Coq, and, on the basis of this, we also proved some basic properties of the semantics and some simple program equivalences. The main contributions of this paper are:

1. The definition of a formal semantics for a sequential subset of Erlang (Core Erlang), based partly on existing formalisations.
2. An implementation for this semantics in the Coq Proof Assistant.
3. Theorems that formalise a number of properties of this formalisation, e.g. determinism, with their machine-checked proofs.
4. Results on program evaluation and equivalence verification using the semantics definition, all formalised in Coq.

The rest of the paper is structured as follows. In Sect. 2 we review the existing formalisations of Core Erlang and Erlang, and compare them in order to help understand the construction of our formal semantics. In Sect. 3 we describe the proposed formal description, including abstractions, syntax, and semantics, while in Sect. 4 we describe a number of applications of the semantics. Section 5 discusses future work and concludes.

2 Related Work

Although there have already been a number of attempts to build a fully-featured formal definition of the Erlang programming language, the existing definitions show varying language coverage, and only some of them, covering mostly the concurrent part of Core Erlang or Erlang, are implemented in a machine-checked proof system. This alone would provide a solid motivation for the work presented in this paper, but our ultimate goal is to prove refactoring-related theorems, such as program equivalences, in the Coq Proof Assistant, based on this semantics.

We have reviewed the extensive related work on formalisations of both Erlang [6,7,9,18] and Core Erlang [5,8,11–13,15,16], incorporating ideas from these sources as appropriate.

The vast majority of related work on Erlang formalisations presents small-step operational semantics. In particular, one of our former project members has already defined most elements of sequential Erlang in the K specification language [9]. We could reformalise this small-step semantics in Coq, but for the proofs to be carried out in the proof assistant, it is too fine-grained. As Owens and others point out, (functional) big-step semantics is a good compromise in terms of amount of detail and ease of use [17]; furthermore, our definition of equivalence does not rely on intermediate execution steps. Thus, our current approach is to define an inductive relational big-step semantics for the language, and down the road we may derive its computable function.

Most papers addressing the formal definition of Erlang focus on the concurrent part of the language, including process management and communication primitives, which is not relevant to our current formalisation goals. Harrison's formalisation of CoErl [8] concentrates on how communication works, and in particular how mailboxes are processed.

Although the papers dealing with the sequential parts tend to present different approaches to defining the semantics, the elements of the language covered and the syntax used to describe them are very similar; there are, however, some minor differences. Some definitions model the language very closely, whilst others abstract away particular aspects; for instance, unlike the work of Neuhäußer et al. [15], the semantics of Lanese et al. [11] describes function applications only for named functions. There is another notable difference in the existing formalisations from the syntactic point of view: some define values as a subclass of expressions by representing them as a distinct syntactic category [6,7,15,18], while others define values as "ground patterns" [11–13,16], that is as a subset of patterns. Both approaches have their advantages and disadvantages, and we discuss this question in more detail in Sect. 3.

We principally used the work by Lanese et al. on defining reversible semantics for Erlang [11–13,16], by defining a language "basically equivalent to a subset of Core Erlang" [16]. Although they do not take Core Erlang functions and their closures into consideration (except for top-level functions), which we needed to define from scratch, their work proved to be a good starting point for defining a big-step operational semantics. In addition, we took the Core Erlang language specification [4] and the Erlang/OTP compiler for Core Erlang as reference points for understanding the basic abstractions of the language in more detail. When defining function applications, we took some ideas from a paper embedding Core Erlang into Prolog [5], and when tackling match expressions, the big-step semantics for FMON [3] proved to be useful. Fredlund's fundamental work [6] was very influential, but his treatment of Erlang formal semantics mainly discusses concurrency.

There were some abstractions missing in almost all papers (e.g. the `let` binding with multiple variables, `letrec`, *map* expressions), for which we had to rely on the informal definitions described in the language specification [4] and the Erlang/OTP compiler. Also, in most of the papers, the global environment is modified at step of the execution; in contrast, our semantics is less fine-grained as side-effects have not been implemented yet. Unfortunately, the official language specification document was written in 2004, and there were some new features (e.g. the *map* data type) introduced to Core Erlang since then. These features do not have an informal description either; however, we took the Erlang/OTP compiler as the reference implementation and build the formalisation on that.

There is a considerable body of work on formalisations of other sequential languages, both functional, as is the case of CakeML [10], and imperative, as in CompCert [14], and indeed the trend to formalising programming language metatheory has been systematised in the POPLmark challenge [2].

3 Formal Semantics of Core Erlang

Here we present our formal definition of Core Erlang formalised in Coq. Throughout this section, we will frequently quote the Coq definition; in some cases, we use the Coq syntax and quote literally, but in case of the semantic rules, we turned the consecutive implications into inference rule notation for better readability. The Coq formalisation is available on Github [1].

3.1 Syntax

This section gives a brief overview of the syntax in our formalisation.

```
Inductive Literal : Type :=
| Atom (s: string)
| Integer (x : Z)
| EmptyList
| EmptyTuple
| EmptyMap.
```

Fig. 1. Syntax of literals

```
Inductive Pattern : Type :=
| PVar (v : Var)
| PLiteral (l : Literal)
| PList (hd tl : Pattern)
| PTuple (t : list Tuple)
```

Fig. 2. Syntax of patterns

The syntax of literals and patterns (Figs. 1 and 2) is based on the papers mentioned in Sect. 2.

For the definition of the syntax of expressions, we need an auxiliary type, which represents function identifiers. In Core Erlang these are pairs of function names and arities (number of arguments). Note that the language allows the overloading of function names as long as it is done with different arities.

$$\texttt{Definition}\ \mathit{FunctionIdentifier} : \texttt{Type} := \mathit{string} \times \mathit{nat}.$$

With the help of this type alias and the previous definitions, we can describe the syntax of expressions (Fig. 3).

As it can be seen, we only include atoms and integers as base type literals in the formalisation. These are representatives of built in data types in the language, and other types (such as floats and binaries) can be added in the future. As mentioned in Sect. 2, our expression syntax is very similar to the existing definitions found in the related work. The main abstractions are based on Fredlund and Vidal's work [6,7,18] and the additional expressions (e.g. `let`, `letrec`, `apply`, `call`) on the Core Erlang specification [4] as well as work by Lanese, Neäuhußer, Nishida and their co-authors [11–13,15,16].

Moreover, in our formalisation, we included the *map* type, primitive operations and function calls are handled alike, and in addition, the *ELet* and *ELetrec* statements handle multiple simultaneous bindings.

```
Inductive Expression : Type :=
| ELiteral (l : Literal)
| EVar (v : Var)
| EFunSig (f : FunctionIdentifier)
| EFun (vl : list Var) (e : Expression)
| EList (hd tl : Expression)
| ETuple (l : list Expression)
| ECall (f: string) (l : list Expression)
| EApply (exp: Expression) (l : list Expression)
| ECase (e : Expression) (l : list Clause)
| ELet (s : list Var) (el : list Expression) (e : Expression)
| ELetrec (fnames : list FunctionIdentifier) (fs[a] : list ((list Var) × Expression)) (e :
Expression)
| EMap (kl vl : list Expression)
 with Clause : Type :=
 | CCons (p : Pattern) (guard e : Expression).
```

[a] This is the list of the defined functions (list of variable lists and body expressions)

Fig. 3. Syntax of expressions

Values. In Core Erlang, literals, lists, tuples, maps, and closures can be values, i.e. results of the evaluation of other expressions. We define values as a separate syntactic category and also include function closures in the definition. Values should be seen as a semantic domain, to which expressions are evaluated (see Fig. 4). This distinction of values allows the semantics to be defined as a big-step relation with a codomain of semantic objects. This approach creates duplication in the syntax, since expression syntax is not reused, but it substantially simplifies building proofs of theorems about values.

```
Inductive Value : Type :=
| VLiteral (l : Literal)
| VClosure[a] (ref : Environment + FunctionIdentifier) (vl : list Var) (e : Expression)
| VList (vhd vtl : Value)
| VTuple (vl : list Value)
| VMap (kl vl : list Value).
```

[a] A closure represents a function definition together with an environment representing the context in which the function was defined: *ref* will be the environment or a reference to it, *vl* will be the function parameter list and *e* will be the body expression. *Environment* is defined in Section 3.2 below.

Fig. 4. Syntax of values

In the upcoming sections, we will use the following syntax abbreviations:

$$tt := VLiteral\ (Atom\ \text{“}true\text{”})$$
$$ff := VLiteral\ (Atom\ \text{“}false\text{”})$$

We now discuss why we this particular approach. As noted in Sect. 2, other approaches are possible: either they are related to patterns (in the work of Lanese et al. [11–13,16]) or to expressions, as in the Erlang formalisations and in the work of Neuhäußer et al. [6,7,15,18]. Moreover, there are two main approach to define the aforementioned relation of values and expressions or patterns:

– Values are not a distinct syntactic category, so they are defined with an explicit subset relation;
– Values are syntactically distinct, and there is no explicit subset relation between values and expressions or patterns [6,7,15,18].

When values are not defined as a distinct syntactic set (or as a semantic domain), a subset relation has to be defined that tells whether an expression represents a value. In Coq, this subset relation is defined by a judgment on expressions, but this would require a proof every time an expression is handled as a value: the elements of a subset are defined by a pair, i.e. the expression itself and a proof that the expression is a value. While this is a feasible approach, it generates lots of unnecessary trivial statements to prove in the dynamic semantics: instead of using a list of values, a list of expressions has to be used, for which proofs must be given about the head and tail being values (see the example in Sect. 3.2 for more details about list evaluation). In addition, the main issue with these approaches is that values do not always form a proper subset of either patterns or expressions [4]: when lambda functions and function identifiers (signatures) are considered, values must include closures (i.e. the normal form of function expressions), which are not included in the expression syntax.

We chose to relate values to expressions, because semantically expressions are evaluated to values and not patterns. In particular, we reused the constructs in the expression syntax in our value definition, and we also included closures, rather than functions as in the work of Neuhäußer et al. [15].

3.2 Semantics

We define a big-step operational semantics for the Core Erlang syntax described in the previous section. In order to do so, we need to define environment types to be included in the evaluation configuration. In particular, we define *environments* which hold values of variable symbols and function identifiers, and separately we define *closure environments* to store closure-local context.

Environment. The variable environment stores the bindings made with pattern matching in parameter passing as well as in `let`, `letrec`, `case` (and `try`) expressions. Note that the bindings may include both variable names and function identifiers, with the latter being associated with function expressions in normal form (closures). In addition, there are top-level functions in the language, and they too are stored in this environment, similarly to those defined with the `letrec` statement.

Top-level, global definitions could be stored in a separate environment in a separate configuration cell, but we decided to handle all bindings in one environment, because this separation would cause a lot of duplication in the semantic rules and in the actual Coq implementation. Therefore, there is one union type to construct a single environment for function identifiers and variables, both local and global. It is worth mentioning that in our case the environment always stores values since Core Erlang evaluation is strict, i.e. an expression first evaluates to some value, then a variable can be bound to this value.

We define the environment in the following way:

$$\texttt{Definition}\ \mathit{Environment} : \texttt{Type} := \mathit{list}\ ((\mathit{Var} + \mathit{FunctionIdentifier}) \times \mathit{Value}).$$

We denote this mapping by Γ in what follows, whilst $\varnothing$ is used to denote the empty environment. We also define a number of helper functions to manage environments, which will be used in formal proofs below. For the sake of simplicity, we omit the actual Coq definitions of these operations and rather provide a short summary of their effect.

- *get_value* Γ *key*: Returns the value associated with *key* in Γ. In the following sections it will be denoted by $\Gamma(\mathit{key})$.
- *insert_value* Γ *key value*: Inserts the (*key,value*) pair into Γ. If this *key* is already included, it will replace the original binding with the new one (according to the Core Erlang specification [4], Sect. 6). The next three function is implemented with this replacing insertion.
- *add_bindings bindings* Γ: Appends to Γ the variable-value pairs given in *bindings*.
- *append_vars_to_env varlist valuelist* Γ: It is used for `let` statements and adds the bindings (variables in *varlist* to values in *valuelist*) to Γ.
- *append_funs_to_env funsiglist param-bodylist* Γ: Appends function identifier-closure pairs to Γ. These closure values are assembled from *param-bodylist* which contains the parameter lists and body expressions.

Closure Environment. In Core Erlang, function expressions evaluate to closures. Closures have to be modeled in the semantics carefully in order to capture the bindings in the context of the closure properly. The following Core Erlang program shows an example where we need to explicitly store a binding context to closures:

```
let X = 42 in
  let Y = fun() -> X in
    let X = 5 in
      apply Y()
```

The semantics needs to make sure that we apply static binding here: the function Y has to return 42 rather than 5. This requires the Y's context to be stored along with its body, which is done by coupling them into a *function closure.*

When evaluating a function expression a closure is created. This value is a copy of the current environment, an expression (the function body), and a variable list (the parameters of the function). The mentioned environment could be encoded with the *VClosure* constructor in the *Value* inductive type using the actual environment (see Fig. 4), however, this cannot be used when the function is recursive. Here is an example:

```
letrec 'x'/0 = fun() -> apply 'x'/0() in apply 'x'/0()
```

In Core Erlang, `letrec` allows definition of recursive functions, so the body of the $'x'/0$ must be evaluated in an environment which stores $'x'/0$ mapped to a closure. But this closure contains the environment in which the body expression must be evaluated and that is the same environment we are trying to define. So the this is a recursively defined structure in embedded closures in the environment where the recursion has no base case. Here is the problem visualized (we denote `apply` $'x'/0()$ with *body*):

$$\{'x'/0 : \mathit{VClosure}\ \{'x'/0 : \mathit{VClosure}\ \{'x'/0 : ...\}\ []\ \mathit{body}\}\ []\ \mathit{body}\}$$

In Coq such constructs or functions cannot be computed without using a clock or fuel [17] which ensures termination. Instead of this we can use the step-by-step unfolding of the environment. This means that, while using the big-step semantics, the environment of the next proving step will be constructed by the current step.

We could use a simple additional attribute in the closure values which marks that the closure is recursive, then it is enough to store the non-recursive part of the environment in the closure. However, if multiple functions are defined at a time and they can potentially apply one another, they do not store information (parameter lists and body expressions) about the other functions. Therefore, we need another approach.

We do not make any syntactic changes to the function body, but we solve this issue by introducing the concept of closure environments. The idea is that the name of the function (variable name or function identifier) is mapped to the application environment (this way, it can be used as a reference). It is enough to encode the function's name with the *VClosure* constructor. This closure environment can only be used together with the use of the environment and items cannot be deleted from it.

$$\texttt{Definition}\ \mathit{Closures} : \texttt{Type} := \mathit{list}\ (\mathit{FunctionIdentifier} \times \mathit{Environment}).$$

All in all, closures will ensure that the functions will be evaluated in the right environments. We also describe the formal evaluation proofs of the examples above in Sect. 4.2. There are two ways of using their evaluation environment (*ref* attribute of *Environment* + *FunctionIdentifier* type):

– Either using the concrete environment from the closure value directly if *ref* is from the type *Environment*;
– Or using the reference and the closure environment to get the evaluation environment when the type of *ref* is *FunctionIdentifier*.

In the next sections, we denote this function-environment mapping with Δ, and $\varnothing$ denotes the empty closure environment. Similarly to ordinary environments, closure environments are managed with a number of simple helper functions; like before, we omit the formal definition of these and provide an informative summary instead.

– *get_env key* Δ: Returns the environment associated with *key* in Δ if *key* is a *FunctionIdentifier*. If *key* is an *Environment*, the function simply returns it. This function is implemented with the help of the next function.
– *get_env_from_closure key* Δ: Returns the environment associated with *key*. If the *key* is not present in the Δ, it returns $\varnothing$.
– *set_closure* Δ *key* Γ: Adds (*key,* Γ) pair to Δ. If *key* exists in Δ, its value will be replaced. Used in the next function.
– *append_funs_to_closure fnames* Δ Γ: Inserts a ($funid_i$, Γ) binding into Δ for every $funid_i$ function identifier in *fnames*.

Dynamic Semantics. The presented semantics, theorems, tests and proofs are available in Coq on the project's Github repository [1].

With the language syntax and the execution environment defined, we are ready to define a big-step semantics for Core Erlang. The operational semantics is denoted by

$$|\Gamma, \Delta, e| \xrightarrow{e} v ::= \mathit{eval\_expr}\ \Gamma\ \Delta\ e\ v$$

where *eval_expr* is the semantic relation in Fig. 5. This means that *e Expression* evaluates to *v Value* in the environment Γ and closure environment Δ. We reused *length*, *combine*, *nth* and *In* from Coq's built-ins [19] in the following definitions.

Prior to presenting the rules of the operational semantics, we define a helper for pointwise evaluation of multiple independent expressions: *eval_all* states that a list of expressions evaluates to a list of values.

$$\begin{aligned}
&\mathit{eval\_all}\ \Gamma\ \Delta\ \mathit{exps}\ \mathit{vals} := \\
&\quad \mathit{length}\ \mathit{exps} = \mathit{length}\ \mathit{vals} \Longrightarrow \\
&\quad (\forall\ (\mathit{exp} : \mathit{Expression}), (\mathit{val} : \mathit{Value}), \\
&\qquad \mathit{In}\ (\mathit{exp}, \mathit{val})\ (\mathit{combine}\ \mathit{exps}\ \mathit{vals}) \Longrightarrow \\
&\qquad\quad |\Gamma, \Delta, \mathit{exp}| \xrightarrow{e} \mathit{val})
\end{aligned}$$

With the help of this proposition, we will be able to define the semantics of function calls, tuples, and expressions of other kinds in a more readable way.

There are two other auxiliary definition which will simplify the main definition:

- $match\_clause$ $(v : Value)$ $(cs : list\ Clause)$ $(i : nat)$ tries to match the ith pattern given in the list of clauses (cs) with the value v. The result is optional; if the ith clause does not match the value, it returns Coq's built-in *None* value while the matching has been successful, it returns the guard and body expressions with the pattern variable-value bindings from the ith clause.
- The *no_previous_match* property states that the clause selection cannot be successful up to the ith clause:

$$\begin{aligned}
&no\_previous\_match\ i\ \Delta\ \Gamma\ \ cs\ v :=\\
&\quad \forall j : nat,\ j < i \Longrightarrow (\forall\ (gg, ee : Expression),\ (bb : list\ (Var \times\ Value)),\\
&\qquad match\_clause\ v\ cs\ j = Some\ (gg,\ ee,\ bb) \Longrightarrow\\
&\qquad\quad (|add\_bindings\ bb\ \Gamma, \Delta, gg\ | \xrightarrow{e} f\!f))
\end{aligned}$$

The formal definition of the proposed operational semantics for Core Erlang is presented in Fig. 5. This figure presents the actual Coq definition, but the inductive cases are formatted as inference rules. In the next paragraphs, we provide short explanations of the less trivial rules.

- Rule 3.7: At first, the case expression e must be evaluated to some v value. Then this v must match to the pattern (*match_clause* function) of the specified ith clause. This match provides the guard, the body expressions of the clause and also the pattern variable binding list. The guard must be evaluated to *tt* in the extended environment with the result of the pattern matching (the binding list mentioned before). The *no_previous_match* states, that for every clause before the ith one the pattern matching cannot succeed or the guard expression evaluates in the extended environment to *ff*. Thereafter the evaluation of the body expression can continue in the above mentioned extended environment.
- Rule 3.8: At first, the parameters must be evaluated to values. Then these values are passed to the auxiliary *eval* function which simulates the behaviour of inter-module function calls (e.g. the addition inter-module call is represented in Coq with the addition of numbers). This results in a value which will be the result of the *ECall* evaluation.
- Rule 3.9: This rule works in similar way that given by de Angelis and co-authors in [5] with the addition of closures. To use this rule, first *exp* has to be evaluated to a closure. Moreover, every parameter must be evaluated to a value. Finally, the closure's body expression evaluates to the result in an extended environment which is constructed from the parameter variable-value bindings and the evaluation environment of the closure. This environment can be acquired from the closure environment indirectly or it is present in the closure value itself (Sect. 3.2).
- Rule 3.10: At first, every expression given must be evaluated to a value. Then the body of the `let` expression must be evaluated in the original environment extended with the variable-value bindings.
- Rule 3.11: From the functions described (a list of variable list and body expressions), closures will be created and appended to the environment and

$\texttt{Inductive}\ \mathit{eval\_expr} : \mathit{Environment} \to \mathit{Closures} \to \mathit{Expression} \to \mathit{Value} \to \texttt{Prop} :=$

$$\frac{}{|\Gamma, \Delta, \mathit{ELiteral}\ l\ | \xrightarrow{e} \mathit{VLiteral}\ l} \quad (3.1) \qquad \frac{}{|\Gamma, \Delta, \mathit{EFunSig}\ \mathit{fsig}| \xrightarrow{e} \Gamma(\mathit{inr}\ \mathit{fsig})} \quad (3.2)$$

$$\frac{}{|\Gamma, \Delta, \mathit{EVar}\ s| \xrightarrow{e} \Gamma(\mathit{inl}\ s)} \quad (3.3) \qquad \frac{}{|\Gamma, \Delta, \mathit{EFun}\ \mathit{vl}\ e| \xrightarrow{e} \mathit{VClosure}\ (\mathit{inl}\ \Gamma)\ \mathit{vl}\ e} \quad (3.4)$$

$$\frac{\mathit{eval\_all}\ \Gamma\ \Delta\ \mathit{exps}\ \mathit{vals}}{|\Gamma, \Delta, \mathit{ETuple}\ \mathit{exps}| \xrightarrow{e} \mathit{VTuple}\ \mathit{vals}} \quad (3.5) \qquad \frac{|\Gamma, \Delta, \mathit{hd}| \xrightarrow{e} \mathit{hdv} \qquad |\Gamma, \Delta, \mathit{tl}| \xrightarrow{e} \mathit{tlv}}{|\Gamma, \Delta, \mathit{EList}\ \mathit{hd}\ \mathit{tl}| \xrightarrow{e} \mathit{VList}\ \mathit{hdv}\ \mathit{tlv}} \quad (3.6)$$

$$\frac{\begin{array}{c} \mathit{match\_clause}\ v\ \mathit{cs}\ i = \mathit{Some}\ (\mathit{guard}, \mathit{exp}, \mathit{bindings}) \\ |\mathit{add\_bindings}\ \mathit{bindings}\ \Gamma, \Delta, \mathit{guard}| \xrightarrow{e} \mathit{tt} \\ |\mathit{add\_bindings}\ \mathit{bindings}\ \Gamma, \Delta, \mathit{exp}| \xrightarrow{e} v' \\ |\Gamma, \Delta, e| \xrightarrow{e} v \\ \mathit{no\_previous\_match}\ i\ \Delta\ \Gamma\ \mathit{cs}\ v \end{array}}{|\Gamma, \Delta, \mathit{ECase}\ e\ \mathit{cs}| \xrightarrow{e} v'} \quad (3.7)$$

$$\frac{\mathit{eval\_all}\ \Gamma\ \Delta\ \mathit{params}\ \mathit{vals} \qquad \mathit{eval}\ \mathit{fname}\ \mathit{vals} = v}{|\Gamma, \Delta, \mathit{ECall}\ \mathit{fname}\ \mathit{params}| \xrightarrow{e} v} \quad (3.8)$$

$$\frac{\begin{array}{c} \mathit{eval\_all}\ \Gamma\ \Delta\ \mathit{params}\ \mathit{vals} \qquad |\Gamma, \Delta, \mathit{exp}| \xrightarrow{e} \mathit{VClosure}\ \mathit{ref}\ \mathit{var\_list}\ \mathit{body} \\ |\mathit{append\_vars\_to\_env}\ \mathit{var\_list}\ \mathit{vals}\ (\mathit{get\_env}\ \mathit{ref}\ \Delta), \Delta, \mathit{body}| \xrightarrow{e} v \end{array}}{|\Gamma, \Delta, \mathit{EApply}\ \mathit{exp}\ \mathit{params}| \xrightarrow{e} v} \quad (3.9)$$

$$\frac{\mathit{eval\_all}\ \Gamma\ \Delta\ \mathit{exps}\ \mathit{vals} \qquad |\mathit{append\_vars\_to\_env}\ \mathit{vars}\ \mathit{vals}\ \Gamma, \Delta, e| \xrightarrow{e} v}{|\Gamma, \Delta, \mathit{ELet}\ \mathit{vars}\ \mathit{exps}\ e| \xrightarrow{e} v} \quad (3.10)$$

For the following rule we introduce $\Gamma' ::= \mathit{append\_funs\_to\_env}\ \mathit{fnames}\ \mathit{funs}\ \Gamma$

$$\frac{\begin{array}{c} \mathit{length}\ \mathit{funs} = \mathit{length}\ \mathit{fnames} \\ |\Gamma', \mathit{append\_funs\_to\_closure}\ \mathit{fnames}\ \Delta\ \Gamma', e| \xrightarrow{e} v \end{array}}{|\Gamma, \Delta, \mathit{ELetrec}\ \mathit{fnames}\ \mathit{funs}\ e| \xrightarrow{e} v} \quad (3.11)$$

$$\frac{\mathit{eval\_all}\ \Gamma\ \Delta\ \mathit{kl}\ \mathit{kvals} \qquad \mathit{eval\_all}\ \Gamma\ \Delta\ \mathit{vl}\ \mathit{vvals} \qquad \mathit{length}\ \mathit{kl} = \mathit{length}\ \mathit{vl}}{|\Gamma, \Delta, \mathit{EMap}\ \mathit{kl}\ \mathit{vl}| \xrightarrow{e} \mathit{VMap}\ \mathit{kvals}\ \mathit{vvals}} \quad (3.12)$$

Fig. 5. The big-step operational semantics of Core Erlang

closure environment associated with the given function identifiers (*fnames*). In these modified contexts the evaluation continues.
- Rule 3.12: Introduces the evaluation for maps. This rule states that every key in the map's key list and value list must be evaluated to values resulting in

two lists of values (for the map keys and their associated values) from which the value map is constructed[1].

We also note that this big-step definition has been partly based on the small-step definition introduced by Lanese [11,13], Nishida [16], and Neuhasser[15] and in some aspects on the big-step semantics in Focaltest [3] and de Angelis' symbolic evaluation [5]. In addition, for most of the language elements discussed, an informal definition is available in the language specification [4].

After discussing these rules, we show an example where the approach in which values are defined as a subset of expressions is more difficult to work with. Let us consider a unary operator (val) on expressions which marks the values of the expressions. With the help of this operator, the type of values can be defined:

$$\mathit{Value} ::= \{e : \mathit{Expression} \mid e\ \mathit{val}\}.$$

Let us consider the key ways in which this would modify our semantics.

- $\mathit{Environment} \to \mathit{Closures} \to \mathit{Expression} \to \mathit{Expression} \to$ `Prop` would be the type of $\mathit{eval\_expr}$. This way we need an additional proposition which states that values are expressions in normal form, *i.e.* they cannot be used on the left side of the rewriting rules.
- The expressions which are in normal form could not be rewritten.
- Function definitions have to be handled as values
- Because of the strictness of Core Erlang, the derivation rules change, additional checks are needed in the preconditions, e.g. in the Rule 3.6:

$$\frac{\begin{array}{cc} \mathit{tlv}\ \mathit{val} & |\Gamma, \Delta, \mathit{hd}| \xrightarrow{e} \mathit{hdv} \lor \mathit{hd} = \mathit{hdv} \\ \mathit{hdv}\ \mathit{val} & |\Gamma, \Delta, \mathit{tl}| \xrightarrow{e} \mathit{tlv} \lor \mathit{tl} = \mathit{tlv} \end{array}}{|\Gamma, \Delta, \mathit{EList}\ \mathit{hd}\ \mathit{tl}| \xrightarrow{e} \mathit{EList}\ \mathit{hdv}\ \mathit{tlv}}$$

This approach has the same expressive power as the presented one, but it has more preconditions to prove while using it. For reason, argue that our formalisation is easier to use.

Proofs of Properties of the Semantics. We have formalised and proved theorems about the attributes of the operations, auxiliary functions and the semantics. We present two examples here, together with sketches of their proofs.

Theorem 1 (Determinism)
$\forall\ (\Gamma : \mathit{Environment}), (\Delta : \mathit{Closures}),\ (e : \mathit{Expression}),\ (v_1 : \mathit{Value}),$
$|\Gamma, \Delta, e| \xrightarrow{e} v_1 \implies (\forall v_2 : \mathit{Value},\ |\Gamma, \Delta, e| \xrightarrow{e} v_2 \implies v_1 = v_2).$

Proof. Induction by the construction of the semantics.

[1] In the future, this evaluation has to be modified, because the normal form of maps cannot contain duplicate keys, moreover it is ordered based on these keys.

- Rules 3.1, 3.2, 3.3 and 3.4 are trivial: e.g. a value literal can only be derivated from its expression counterpart.
- Rules 3.5 and 3.12 are similar, a map is basically a double tuple in the current semantics. According to the induction hypothesis each element in the expression tuple can be evaluated to a single value, so the tuple itself evaluates to the tuple which contains these values. The proof for maps is similar.
- Rule 3.6 The head and the tail expression of the list can be evaluated to a single head and tail value according to the induction hypotheses. So the list constructed from the head and tail expressions can only be evaluated to the value list constructed from the head and tail values.
- Rule 3.7 The induction hypothesis states that the base and the clause body and guard expressions evaluate deterministically. The clause selector functions are also deterministic, so there is only one possible way to select a body expression to evaluate.
- The other cases are similar to those presented above.

□

Theorem 2 (Commutativity). $\forall$ *(v, v' : Value),*
eval "plus"$[v; v']$ $=$ *eval "plus"*$[v'; v]$.

Proof. First we separate cases based on the all possible construction of values (5 constructors, *v* and *v'* values, that is 25 cases). In every case where either of the values is not an integer literal, the *eval* function results in the same error value (in this version we can not distinguish errors) on both side of the equality.

One case is remaining, when both *v* and *v'* are integer literals. In this case the definition of *eval* is the addition of these numbers, and the commutativity of this addition has already been proven in the Coq standard library [19]. □

It is important to note that if exceptions are included in the formalisation, then this theorem probably would not be correct as it stands, and would need to be adjusted.

4 Application and Testing the Semantics

In this section we present some use cases. First, we elaborate on the verification of the semantics definition by testing it against the Erlang/OTP compiler, then we show some examples on how we used the formalisation for deriving program behaviour and for proving program equivalence.

4.1 Testing the Semantics

Due to a lack of an up-to-date language specification, we validated the correctness of our semantics definition by comparing it to the behaviour of the code emitted by the Erlang/OTP compiler.

To test our formal semantics, we first used equivalence partitioning. We have written tests both in Coq (v 8.11.0) and in Core Erlang (OTP 22.0) for every type of expression defined in our formalisation (i.e. for every possible inference rule application). There are also some special complex expressions that require separate test cases (e.g. using bound variables in `let` expressions, application of recursive functions, returned functions etc.). In the future, we plan to automate the evaluation of both Coq and Core Erlang code and comparison of the results.

Apart from the formal expression evaluation examples, the proofs about the properties of the semantics (e.g. determinism) and the expression equivalences also provided an additional layer of assurance about complying to the behaviour of the Erlang/OTP compiler.

4.2 Formal Expression Evaluation

Here we demonstrate how Core Erlang expressions are evaluated in the formal semantics. For readability, we use concrete Core Erlang syntax in the proofs, and trivial statements (e.g. the use of Rule 3.9) are omitted from the proof tree. The first example shows how to evaluate a simple expression with binding:

$$\dfrac{\dfrac{\overline{\{X:5\}(X)=5}}{|\{X:5\},\varnothing,X| \xrightarrow{e} 5}\ 3.3}{|\varnothing,\varnothing,\texttt{let}\ X=5\ \texttt{in}\ X| \xrightarrow{e} 5}\ 3.10$$

The next example is the first one mentioned in Sect. 3.2 and intends to demonstrate the purpose of the closure values. Here at the application of 3.9 it is shown that the body of the application is evaluated in the environment given by the closure. For readability, we denote the inner `let` $X = 5$ `in apply` $Y()$ expression with *exp*.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\{X:42\}(X)=42}}{|\{X:42\},\varnothing,X| \xrightarrow{e} 42}\ 3.3}{|\{X:5, Y: \mathit{VClosure}\ (\mathit{inl}\ \{X:42\})\ []\ X\},\varnothing,\texttt{apply}\ Y()| \xrightarrow{e} 42}\ 3.9}{|\{X:42, Y: \mathit{VClosure}\ (\mathit{inl}\ \{X:42\})\ []\ X\},\varnothing,\mathit{exp}| \xrightarrow{e} 42}\ 3.10}{|\{X:42\},\varnothing,\texttt{let}\ Y=\texttt{fun}()\to X\ \texttt{in}\ \mathit{exp}| \xrightarrow{e} 42}\ 3.10}{|\varnothing,\varnothing,\texttt{let}\ X=42\ \texttt{in}\ \texttt{let}\ Y=\texttt{fun}()\to X\ \texttt{in}\ \mathit{exp}| \xrightarrow{e} 42}\ 3.10$$

Next we show the previous example, but now using a recursive function between the two `let` expressions in order to demonstrate the use of the closure environment. For readability we denote $\mathit{VClosure}\ (\mathit{inr}\ 'f'/0)\ []\ X$ with *clos* and the inner `let` $X = 5$ `in apply` $'f'/0()$ with *exp*, just like before.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\{X:42, 'f'/0: clos\}(X)=42}}{|\{X:42,'f'/0:clos\},\{'f'/0:\{X:42,'f'/0:clos\}\},X| \xrightarrow{e} 42}\;3.3}{|\{X:5,'f'/0:clos\},\{'f'/0:\{X:42,'f'/0:clos\}\},\texttt{apply}\ Y()| \xrightarrow{e} 42}\;3.9}{|\{X:42,'f'/0:clos\},\{'f'/0:\{X:42,'f'/0:clos\}\},exp| \xrightarrow{e} 42}\;3.10}{|\{X:42\},\varnothing,\texttt{letrec}\ 'f'/0=\texttt{fun}()\rightarrow X\ \texttt{in}\ exp| \xrightarrow{e} 42}\;3.11}{|\varnothing,\varnothing,\texttt{let}\ X=42\ \texttt{in}\ \texttt{letrec}\ 'f'/0=\texttt{fun}()\rightarrow X\ \texttt{in}\ exp| \xrightarrow{e} 42}\;3.10$$

At the point of the use of Rule 3.11 we save in the closure environment the current local environment extended with the closure value of the function bound to this function's identifier. This way, later in the evaluation, this environment can be used (e.g. when using Rule 3.3).

The last example is the second one mentioned in Sect. 3.2 and cannot be evaluated in our formalisation, because of divergence. For readability we introduce $\Gamma := \{'x'/0 : \mathit{VClosure}\ (\mathit{inr}\ 'x'/0)\ []\ (\texttt{apply}\ 'x'/0())\}$ (the environment after the binding is added).

$$\dfrac{\dfrac{\dfrac{\cdots}{|\Gamma,\{'x'/0:\Gamma\},\texttt{apply}\ 'x'/0()| \xrightarrow{e} ??}\;3.9}{|\Gamma,\{'x'/0:\Gamma\},\texttt{apply}\ 'x'/0()| \xrightarrow{e} ??}\;3.9}{|\varnothing,\varnothing,\texttt{letrec}\ 'x'/0=\texttt{fun}()\rightarrow\texttt{apply}\ 'x'/0()\ \texttt{in}\ \texttt{apply}\ 'x'/0()| \xrightarrow{e} ??}\;3.11$$

4.3 Expression Equivalence Proofs

Last but not least, let us present some expression equivalence proofs demonstrating the usability of this semantics definition implemented in Coq. This is a significant result of the paper since our ultimate goal with the formalisation is to prove refactorings correct.

For the simplicity, we use $+$ to refer to the *append_vars_to_env* function and also for the addition inter-module call (i.e. $e_1 + e_2$ will denote the expression *ECall "plus"* $[e_1, e_2]$ in the following proofs) both in proofs and quoted code.

First, we present a rather simple example of expression equivalence.

Example 1 (Swapping variable values).

```
let X = 5 in let Y = 6 in X + Y
```

is equivalent to

```
let X = 6 in let Y = 5 in X + Y
```

Proof. This example can be proved by specialising Example 2 with concrete values. □

Also a more abstract local refactoring also can be proved correct in our system.

Example 2 (Swapping variable expressions). If we make the following assumptions:

$$
\begin{array}{ll}
|\Gamma, \Delta, e_1| \xrightarrow{e} v_1 & |\Gamma + \{A : v_2\}, \Delta, e_1| \xrightarrow{e} v_1 \\
|\Gamma, \Delta, e_2| \xrightarrow{e} v_2 & |\Gamma + \{A : v_1\}, \Delta, e_2| \xrightarrow{e} v_2 \\
A \neq B &
\end{array}
$$

then

```
let A = e1 in let B = e2 in A + B
```

is equivalent to

```
let A = e2 in let B = e1 in A + B
```

Proof. First, we present the problem formalised.

$$
\begin{aligned}
&\forall\ (\Gamma : \mathit{Environment}), (\Delta : \mathit{Closures}), (t : \mathit{Value}), (A, B : \mathit{Var}) \\
&\qquad |\Gamma, \Delta, e_1| \xrightarrow{e} v_1 \Longrightarrow |\Gamma + \{A : v_2\}, \Delta, e_1| \xrightarrow{e} v_1 \Longrightarrow \\
&\qquad |\Gamma, \Delta, e_2| \xrightarrow{e} v_2 \Longrightarrow |\Gamma + \{A : v_1\}, \Delta, e_2| \xrightarrow{e} v_2 \Longrightarrow A \neq B \Longrightarrow \\
&|\Gamma, \Delta, \mathit{ELet}\ [A]\ [e_1]\ (\mathit{ELet}\ [B]\ [e_2] \\
&\qquad (\mathit{ECall}\ \text{``}\mathit{plus}\text{''}\ \ [\mathit{EVar}\ A; \mathit{EVar}\ B]))| \xrightarrow{e} t \Longleftrightarrow \\
&|\Gamma, \Delta, \mathit{ELet}\ [A]\ [e_2]\ (\mathit{ELet}\ [B]\ [e_1] \\
&\qquad (\mathit{ECall}\ \text{``}\mathit{plus}\text{''}\ \ [\mathit{EVar}\ A; \mathit{EVar}\ B]))| \xrightarrow{e} t
\end{aligned}
$$

The two directions of this equivalence are proved in exactly the same way, so only the forward ($\Longrightarrow$) direction is presented here.

Now the main hypothesis has two `let` statements in itself. These statements could have only been evaluated with Rule 3.10, i.e. there are two values (v_1 and v_2 because of the determinism and the assumptions) to which e_1 and e_2 evaluates:

$$|\Gamma, \Delta, e_1| \xrightarrow{e} v_1 \text{ and } |\Gamma + \{A : v_1\}, \Delta, e_2| \xrightarrow{e} v_2$$

It is important to note, that during the evaluation of the inner `let`, A has already been bound to v_1. Moreover a new hypothesis also appeared:

$$|\Gamma + \{A : v_1, B : v_2\}, \Delta, A + B| \xrightarrow{e} t$$

This hypothesis implies that $t = \mathit{eval}\ \text{``}\mathit{plus}\text{''}\ [v_1, v_2]$ because of the evaluation with Rule 3.8 and 3.3, also when we add variables to the environment, the existing binding will be replaced.

Now, the goal can be solved with the construction of a derivation tree. We denote $\Gamma + \{A : v_2, B : v_1\}$ with Γ_v.

$$\dfrac{\dfrac{}{|\Gamma, \Delta, e_2| \xrightarrow{e} v_2} \qquad \dfrac{|\Gamma_v, \Delta, A + B| \xrightarrow{e} t \qquad \dfrac{}{|\Gamma + \{A : v_2\}, \Delta, e_1| \xrightarrow{e} v_1}}{|\Gamma + \{A : v_2\}, \Delta, \texttt{let } B = e_1 \texttt{ in } A + B| \xrightarrow{e} t}\;3.10}{|\Gamma, \Delta, \texttt{let } A = e_2 \texttt{ in let } B = e_1 \texttt{ in } A + B| \xrightarrow{e} t}\;3.10$$

According to our assumptions, e_1 and e_2 evaluates to v_1 and v_2 in Γ and also in its extensions which contains bindings to A. Now for the addition, the following derivation tree can be used.

$$\dfrac{\dfrac{\Gamma_v(B) = v_1}{|\Gamma_v, \Delta, B| \xrightarrow{e} v_1}\;3.3 \qquad \dfrac{\Gamma_v(A) = v_2}{|\Gamma_v, \Delta, A| \xrightarrow{e} v_2}\;3.3 \qquad \textit{eval “plus”}\ [v_2, v_1] = t}{|\Gamma_v, \Delta, A + B| \xrightarrow{e} t}\;3.8$$

We can use the Rule 3.8 to evaluate the addition. The parameter variables will evaluate to v_2 and v_1 because of the replacing insertion mentioned before. With this knowledge, we get: *eval “plus”* $[v_2, v_1] = t$. As mentioned before $t =$ *eval “plus”* $[v_1, v_2]$. So it is sufficient to prove, that:

$$\textit{eval “plus”}\ [v_2, v_1] = \textit{eval “plus”}\ [v_1, v_2] \tag{4.1}$$

The commutativity of *eval* (Theorem 2) can be used to solve this equality. □

Is it possible to replace the assumptions of Example 2 with statements about e_1 and e_2 not containing the variables A and B? Not directly; it it would require a theorem stating the evaluation of an expression that does not contain the variable A does not change in the extended environment which contains a binding of the variable A. This statement is not true for closure values, because they potentially save their evaluation environment which would differ in this case.

Now, we prove a similar simple local refactoring (this example is also generalised over the A, B variables).

Example 3 (Swapping variables in simultaneous let). If we assume that $A \neq B$ then

```
    let <A, B> = <e1, e2> in A + B
```

is equivalent to

```
    let <A, B> = <e2, e1> in A + B
```

Proof. The proof for this example is very similar to the proof for Example 2. The only difference is that one step is enough to evaluate the `let` expression. Inside it both e_1 and e_2 expressions evaluate in the same environment and that is the reason why no assumptions are needed. □

Finally, we show another simple local refactoring about moving an expression to a function.

Example 4 (Moving an expression to a function).

```
e
```

is equivalent to

```
let A = fun() -> e in
  apply A()
```

Proof. In this case, both directions should be proved. At first, we formalise the problem:

$$\forall\ (\varGamma : \mathit{Environment}), (\varDelta : \mathit{Closures}), (t : \mathit{Value}), (A : \mathit{Var})$$
$$|\varGamma, \varDelta, e| \xrightarrow{e} t \Longleftrightarrow$$
$$|\varGamma, \varDelta, \mathit{ELet}\ [A]\ [\mathit{EFun}\ []\ e]\ (\mathit{EApply}\ (\mathit{EVar}\ A)\ [])| \xrightarrow{e} t$$

$\Longrightarrow$ direction:

This can be proved by the construction of a derivation tree. We denote $\varGamma + \{A : \mathit{VClosure}\ (\mathit{inl}\ \varGamma)\ []\ e\}$ with $\varGamma_A$ and the value $\mathit{VClosure}\ (\mathit{inl}\ \varGamma)\ []\ e$ with cl in the tree.

$$\dfrac{\dfrac{}{|\varGamma, \varDelta, \mathtt{fun}() \to e| \xrightarrow{e} cl}\ 3.3 \qquad \dfrac{\dfrac{}{|\varGamma_A, \varDelta, A| \xrightarrow{e} cl}\ 3.4 \qquad \dfrac{}{|\varGamma, \varDelta, e| \xrightarrow{e} t}\ \text{Hypo.}}{|\varGamma_A, \varDelta, \mathtt{apply}\ A()| \xrightarrow{e} t}\ 3.9}{|\varGamma, \varDelta, \mathtt{let}\ A = \mathtt{fun}() \to e\ \mathtt{in}\ \mathtt{apply}\ A()| \xrightarrow{e} t}\ 3.10$$

$\Longleftarrow$ direction:

This can be proved by the deconstruction of the hypothesis for the `let` expression. First only the 3.10 could have been used for the evaluation. This means that the function evaluates to some value, i.e. to the closure $\mathit{VClosure}\ (\mathit{inl}\ \varGamma)\ []\ e$, because of Rule 3.4. We get a new hypothesis:

$$|\varGamma + \{A : \mathit{VClosure}\ (\mathit{inl}\ \varGamma)\ []\ e\}, \varDelta, \mathtt{apply}\ A()| \xrightarrow{e} t$$

Then the evaluation continued with the application of Rule 3.9. This means, that the A variable evaluates to the above mentioned closure (because Rule 3.3 and the replacing insertion into the environment) and the body of this closure evaluates to t in the closure's stored environment extended with the parameter-value bindings (in this case there is none). This means we get the following hypothesis: $|\varGamma, \varDelta, e| \xrightarrow{e} t$ which is exactly what we want to prove. □

To prove these examples in Coq, a significant number of lemmas were needed, such as the expansion of lists, the commutativity of *eval*, and so forth. However, the proofs mostly consist of the combination of hypotheses similar to the proofs in this paper. Although sometimes additional case analyses were needed, resulting in lots of sub-goals, these were solved similarly. In the future, these proofs should be simplified with the introduction of smart tactics and additional lemmas.

5 Future Work and Conclusion

Using Coq we have formalised a substantial subset of (Core) Erlang together with its semantics, and proved results on the formalisation itself as well as establishing a number of program equivalences in that semantics. Use of this formalisation is demanding in practice, partly because the Coq Proof Assistant makes its users write down proofs explicitly step by step. Of course this is a necessity of the correctness, however, this property results in lengthy proofs. This work is a first step in our project to establish a platform on which we can build and prove correct a range of refactorings for an existing programming language: Erlang.

There are several ways to enhance our formalisation. We intend to focus first on extending the semantics with additional expressions (e.g. binaries); formalising exceptions and exception handling, so that we can distinguish between different errors and also divergence; and handling and logging side-effects. To improve the formalisation we will create new lemmas, theorems and tactics to shorten the Coq implementation of the proofs; formalise and prove more refactoring strategies; and move to automate the testing process, running and comparing the results of the Core Erlang code and the theorems automatically.

Our longer-term goals include extending the work to Erlang (semantics and syntax), including concurrency, and distinguishing primitive operations and inter-module calls. The ultimate goal of the project is to change the core of a scheme-based refactoring system to a formally verified core.

Acknowledgements. The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, "Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications (3IN)").

Project no. ED_18-1-2019-0030 (Application domain specific highly reliable IT solutions subprogramme) has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme funding scheme.

References

1. Core Erlang Formalization. https://github.com/harp-project/Core-Erlang-Formalization/tree/general-proofs-with-clos-env. Accessed 11 Jun 2020
2. Aydemir, B.E., et al.: Mechanized metatheory for the masses: the PoplMark challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005). https://doi.org/10.1007/11541868_4
3. Carlier, M., Dubois, C., Gotlieb, A.: A first step in the design of a formally verified constraint-based testing tool: FocalTest. In: Brucker, A.D., Julliand, J. (eds.) TAP 2012. LNCS, vol. 7305, pp. 35–50. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30473-6_5
4. Carlsson, R., et al.: Core Erlang 1.0 language specification (2004)
5. De Angelis, E., et al.: Bounded symbolic execution for runtime error detection of Erlang programs. In: Kahsai, T., Vidal, G. (eds.) Proceedings 5th Workshop on Horn Clauses for Verification and Synthesis, HCVS 2018. EPTCS, vol. 278 (2018). https://doi.org/10.4204/EPTCS.278.4

6. Fredlund, L.Å.: A framework for reasoning about Erlang code. Ph.D. thesis, Royal Institute of Technology, Stockholm (2001)
7. Fredlund, L.Å., Gurov, D., Noll, T., Dam, M., Arts, T., Chugunov, G.: A verification tool for ERLANG. Int. J. Softw. Tools Technol. Transf. **4**(4), 405–420 (2002). https://doi.org/10.1007/s100090100071
8. Harrison, J.R.: Towards an Isabelle/HOL formalisation of core Erlang. In: Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang, Erlang 2017. ACM (2017). https://doi.org/10.1145/3123569.3123576
9. Kőszegi, J.: KErl: Executable semantics for Erlang. In: CEUR Workshop Proceedings, vol. 2046, pp. 144–160 (2018)
10. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Principles of Programming Languages (POPL), January 2014. ACM Press (2014). https://doi.org/10.1145/2535838.2535841
11. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. J. Log. Algebr. Methods Program. **100** (2018). https://doi.org/10.1016/j.jlamp.2018.06.004
12. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr: a causal-consistent reversible debugger for Erlang. In: Gallagher, J.P., Sulzmann, M. (eds.) FLOPS 2018. LNCS, vol. 10818, pp. 247–263. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-90686-7_16
13. Lanese, I., Sangiorgi, D., Zavattaro, G.: Playing with bisimulation in Erlang. In: Boreale, M., Corradini, F., Loreti, M., Pugliese, R. (eds.) Models, Languages, and Tools for Concurrent and Distributed Programming. LNCS, vol. 11665, pp. 71–91. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21485-2_6
14. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52** (2009). https://doi.org/10.1145/1538788.1538814
15. Neuhäußer, M.R., Noll, T.: Abstraction and model checking of core erlang programs in Maude. In: Denker, G., Talcott, C.L. (eds.) 6th International Workshop on Rewriting Logic and its Applications, WRLA 2006. ENTCS, vol. 174. Elsevier (2007). https://doi.org/10.1016/j.entcs.2007.06.013
16. Nishida, N., Palacios, A., Vidal, G.: A reversible semantics for Erlang. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 259–274. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4_15
17. Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K.: Functional big-step semantics. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 589–615. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_23
18. Vidal, G.: Towards symbolic execution in Erlang. In: Voronkov, A., Virbitskaite, I. (eds.) PSI 2014. LNCS, vol. 8974, pp. 351–360. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46823-4_28
19. The Coq Proof Assistant Documentation. https://coq.inria.fr/documentation. Accessed 21 Feb 2020