A Proof Assistant Based Formalisation of Core Erlang

Péter Bereczky[1][0000−0003−3183−0712], Dániel Horpácsi[1][0000−0003−0261−0091], and Simon Thompson[2][0000−0002−2350−301X]

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

Abstract. Our research is part of a wider project that aims to investigate and reason about the correctness of scheme-based source code transformations on Erlang programs. In order to formally reason about the definition of a programming language and the software that has been built with it, we need a mathematically rigorous description of that language. In this paper, we present our 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. closures). We also present essential properties of the formalisation (e.g. determinism) along with their machine-checked proofs. Finally, we prove the correctness of some simple refactoring strategies.

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 verified by regression testing only. Higher assurance can be achieved by making a formal argument (i.e. 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 programming language semantics in question, which enables formal verification. Unfortunately, most programming languages lack fully formal definitions, which makes it challenging to deal with them in formal ways. Since we are dedicated to improve trustworthiness of Erlang

refactorings via formal verification, we put effort in formalising Erlang and its functional core called Core Erlang. Core Erlang is not merely a subset of Erlang; in fact, Erlang (along with other functional languages) translates to Core Erlang as part of the compilation process.

This paper presents work on the Coq-formalisation of a big-step semantics for Core Erlang. In general, if formal semantics is not available for a particular language, one can take the language specification and the reference implementation to build a formalisation thereon; in our case, we could rely not only on these artifacts, but also on some previously published semantics definitions. Thus, we reviewed the existing papers on the Core Erlang language and its semantics, distilled, merged and extended them, to obtain a definition that can be properly embedded in Coq. Not only did we build a mechanically verifiable definition, but we also proved some basic properties of the semantics as well as proved simple program equivalences.

In particular, the main contributions of this paper are:

1. A formal semantics definition for a sequential subset of Erlang (Core Erlang), partly based on existing formalisations.
2. An implementation for this semantics in the Coq proof assistant.
3. Theorems that formalise the 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 the Coq proof assistant.

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

2 Related work

Although there have already been a number of attempts to build a full-featured formal definition of the Erlang programming language, most are not complete, and none is implemented in a machine-checked proof system. Since we intend to formalise refactoring-related theorems in the Coq proof assistant, we decided to write an executable formal semantics for Erlang in Coq, and as a stepping stone, we built the formalisation of Core Erlang. We were intended to reuse existing results, so we have reviewed the related work on existing paper-based formalisations of both Erlang and Core Erlang. In this section, we present a survey of these and point out how we managed to reuse elements of the existing resources.

As a matter of fact, most of the papers addressing the formal definition of Erlang focus on the parallel part (process management and communication primitives) of the language, which is not relevant to our current formalisation goals. Nevertheless, there are some sources that deal with the sequential part as well. Although they tend to present different approaches as to defining the semantics, the captured elements of the language and their syntax appears to be very similar in each paper. There are slight differences in the level of detail: some definitions model the core language very closely, whilst some are more abstract. For example, unlike [10], [4] does not differentiate function applications and calls, nor does describe `let` statements. From the syntax richness point of view, the cited resources can be organized into two groups: [4, 5, 11] presenting fewer abstractions and [6–10] discussing a richer language syntax.

There was another notable difference in the existing formalisations from the syntax point of view: some define values as a subset of expressions distinguished by defining them in a different syntactic category [4, 5, 9, 11], and some define values as "ground patterns" [6–8, 10], i.e. subset of patterns. Both approaches have their advantages and disadvantages, we will discuss this question in more detail in Section 3.

We principally used the work by Lanese et al. on defining reversible semantics for Erlang [6–8, 10], who define a language "basically equivalent to a subset of Core Erlang" [10]; their work proved to be a good starting point for defining a big-step operational semantics. Besides, we took the Core Erlang Documentation [3] and the reference compiler for Core Erlang as compasses for understanding the basic abstractions of the language in more detail.

It is worth mentioning that the vast majority of the abovementioned related work present small-step operational semantics, therefore rather than reusing the existing formalisations, we only adapted the main ideas thereof. Moreover, the work by Lanese et al. does not take Erlang functions and their closures into consideration (except for top-level functions), thus we needed to define these from scratch. There is also a denotational semantics for sequential Core Erlang defined in [4], but the Erlang formal semantics section discusses parallel parts mainly.

We have found that there are some abstractions missing in [6–8, 10] (e.g. `let` binding with multiple variables, the `letrec` statement), for which we had to rely on the informal definitions described in [3] and in the case of `letrec`, the formalisation in [9]. Also, in the former papers, the global environment is modified in every single step of the execution; in contrast, our semantics works less fine-grained as side-effects have been not 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 reference implementation and build the formalisation thereon.

3 Formal semantics of Core Erlang

After reviewing related work, 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 entire formalisation is available on Github [2].

3.1 Syntax

This section gives a brief overview on 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 : Tuple)
with Tuple : Type :=
| TNil
| TCons (hd : Pattern) (tl : Tuple).
```

Fig. 2. Syntax of patterns

The syntax of literals and patterns (Figures 1 and 2) was basically borrowed from the papers mentioned in the previous section. The only addition is the `map` construction (*EmptyMap* literal); float literals are left out, because in our applications, they can be handled as they were integers. The tuple pattern is represented with a list, which is constructed inductively. For technical reasons, Coq's built-in lists cannot be used here: Coq cannot ensure fixpoints' decreasing arguments for implicit mutual induction (e.g rose tree types).

For the definition of the syntax of expressions, we need the following auxiliary type:

```
Definition FunctionSignature : Type := string × nat.
```

With the help of this type alias and the previous definitions, we can describe the syntax of the expressions (Figure 3). As mentioned in Section 2, our expression syntax is very similar to the existing definitions found in the related work. The main abstractions are based on [4,5,11] and the additional expressions (e.g. `let`, `letrec`, `apply`, `call`) on [3,6–10]. However, 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 bindings at the same time.

Values In Core Erlang, literals, lists, tuples, maps, and closures can be values, i.e. results of the evaluation of other expressions. As pointed out in Section 2,

```
Inductive Expression : Type :=
| ELiteral (l : Literal)
| EVar (v : Var)
| EFunSig (f : FunctionSignature)
| 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 FunctionSignature) (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

there were two approaches discussed in the related work: either values are related to patterns [6–8, 10] or values are related to expressions [4, 5, 9, 11]. We have decided to relate values to expressions, because semantically values are derived from expressions and not patterns. Moreover, there are three methods to define the aforementioned relation of values and expressions:

- Values are not a distinct syntactic category, so they are defined with an explicit subset relation;
- Values are syntactically distinct and used in the definition of expressions [4, 4, 11];
- Values are syntactically distinct, but there is no explicit subset relation between values and expressions or patterns [9].

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 practically defined with 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. This is a feasible approach, but generates lots of unnecessary otherwise trivial statements to prove in the dynamic semantics: instead of using a list of values, a list of expressions has to be used where proofs must be given about the head and tail being values; see the example in Section 3.2 about the list evaluation. In addition, the main issue with these approaches is that values do not always form a proper subset of patterns or expressions [3]: when lambda functions and function signatures are considered, values must include closures, which, on the other hand, are not present in the syntax.

For the above reasons, we define values separately from syntax, but unlike [9], we rather include function closures in the definition than the functions themselves. In fact, we define values as a semantic domain, to which expressions are evaluated (see Figure 4). This distinction of values allows the semantics to be defined in a big-step way with domain changing (from expressions to values). Naturally, this approach causes duplication in the syntax definition (i.e. value syntax is not reused, unlike in [4,5,11]), but it saves a lot when proving theorems about values.

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

[a] The closure means a function and an application environment, where *ref* will be a reference to the environment, *vl* will be the function parameter list and *e* will be the body expression.

Fig. 4. Syntax of values

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

$$tt := VLiteral\ (Atom\ "true")$$
$$ff := VLiteral\ (Atom\ "false")$$

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 signatures, 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`) statements. Note that the bindings may include both variable names and functions signatures, with the latter being associated with function expressions (closures). In addition, there are top-level functions in the language, they too are stored in this environment, similarly to those defined with the `letrec` statement. In the rest of the paper, we will call these "signature" functions.

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 names 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. This means that before binding a variable to an expression, the expression is guaranteed to be evaluated to a value.

We define the environment in the following way:

`Definition` *Environment* : `Type` := *list* ((*Var* + *FunctionSignature*) × *Value*).

In the next sections, we denote this mapping with Γ, whilst $\varnothing$ denotes the empty environment. We also define a number of helper functions to manage environments, which will be used in formal proofs in the following sections. 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.
- *insert_value* Γ *key value*: Inserts the $(key, value)$ pair into Γ. If this *key* has already been present, it will overwrite its existing value.
- *add_bindings bindings* Γ: Appends to Γ the variable-value bindings given in *bindings*.
- *append_vars_to_env varlist valuelist* Γ: It is used for `let` statements, and modifies the closure values in *valuelist* (rewrites their environment reference to the variable specified in the `let`), and adds the bindings (*varlist* elements to modified *valuelist* elements) to Γ.
- *append_funs_to_env funsiglist param-bodylist* Γ: Appends to Γ function signature-closure pairs. The closures are constructed from *param-bodylist* which contains parameter lists and body expressions.

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

```
let X = 5 in
  let Y = fun() -> X in
    let X = 10 in
      apply Y()
```

The semantics needs to make sure that we apply static binding here: the function Y has to return 5 rather than 10. This requires the Y's context stored along with its body, which is done by coupling them into a *function closure*.

When binding a name or signature to a function (either with `let` or `letrec` statement), a closure is created. This is a copy of the current environment, an expression (the function body), and a variable list (the parameters of the function) associated with the name of the function; in fact, we get a mapping from the function's name or signature to its definition environment.

This information could be encoded with the *VClosure* constructor in the *Value* inductive type using the environment instead of the *ref* reference to it

(see Figure 4), however, this cannot be used when the function is recursive. Here is an example:

```
letrec 'f1'/0 = fun() -> apply 'f1'/0()
```

In Core Erlang, `letrec` allows definition of recursive functions, so the body of the `'f1'/0` must be evaluated in an environment which stores `'f1'/0` mapped to a closure. But this closure contains the environment in which the body expression must be evaluated which is the same environment mentioned before. So the this is a recursion in embedded closures in the environment. Here is the problem visualized:

{'f1'/0 : *VClosure* {'f1'/0 : *VClosure* {'f1'/0 : ...}} [] (`apply 'f1'/0()`) }

We do not apply any syntactical 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 signature) 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.

`Definition` *Closures* : `Type` := *list* ((*Var* + *FunctionSignature*) × *Environment*).

When a function application happens, the evaluation environment should use this closure. Let us see the original example with this idea.

In that case, we want to return 5, and not 10. So when applying the Y function, it must not be evaluated in the actual environment $\{X : 10\}$, but in the one where Y was defined $\{X : 5\}$. The environment associated with Y in the closure mapping is exactly this environment, so the evaluation can continue in this one (extended with the parameter variable binding, if there are parameters). All in all, closures will ensure that the functions will be evaluated in the right environments (a similar fully formal example is described in Section 4.2).

In the next sections, we denote this 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 Γ: Returns the environment associated with *key* in Δ. Results default parameter Γ, if *key* is the empty string (""). Implemented with the *get_env_from_closure* 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 key* Γ: Adds (key, Γ) pair to Δ. If *key* exists in Δ, its value will be overwritten. Used in the *append* functions.

- *check_closure* Δ *key*: Removes from Δ. *key* (and its associated environment). Used in the implementation of *append_vars_to_closure*[3].
- *append_vars_to_closure vars vals* Δ Γ: Inserts variable bindings to Γ into Δ. Only those variables will be considered, which are bound to a closure value (this is the reason, why *exps* is needed).
- *append_funs_to_closure fnames* Δ Γ: Inserts function signature to Γ bindings into Δ

Dynamic Semantics With the language syntax and the execution environment defined, we are ready to define the big-step semantics for Core Erlang. 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. 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. In this definition, we reuse *length*, *combine* and *In* from Coq's built-ins [1].

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

There is another auxiliary definition we will simplify the definition with: (*match_clause v cs i*) tries to match the ith pattern given in the list of clauses (*cs*) with the value v. The result is optional, on successful matching it returns the ith clause guard and body expressions with the pattern variable-value bindings, otherwise it returns *None*.

The formal definition of the proposed operational semantics for Core Erlang is presented in Figure 5. We remind the reader that the figure presents the actual Coq definition, but the inductive cases are formatted as inference rules. We also note that this big-step definition is partly based on the small-step definition discussed in [7–10]. In addition, for most of the language elements defined an informal definition is available in [3]. In the next paragraphs, we provide short explanations of 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 specified ith clause (*match_clause* function). This match provides the guard, the body expressions of the clause and also the pattern variable binding list. The guard must evaluate to true in the extended environment with the binding list. The *no_previous_match* states, that for every j which is smaller than i the match cannot happen or the guard expression evaluates in the extended environment to false. Thereafter the evaluation of the body expression can continue in this environment.

[3] This is needed when a variable was bound to a non-closure value and before this binding the variable had been bound to a closure

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

$$\frac{}{(\Gamma, \Delta, ELiteral\ l) \xrightarrow{e} VLiteral\ l} \quad (3.1) \qquad \frac{}{(\Gamma, \Delta, EVar\ s) \xrightarrow{e} get\_value\ \Gamma\ (inl\ s)} \quad (3.2)$$

$$\frac{}{(\Gamma, \Delta, EFunSig\ fsig) \xrightarrow{e} get\_value\ \Gamma\ (inr\ fsig)} \quad (3.3)$$

$$\frac{}{(\Gamma, \Delta, EFun\ vl\ e) \xrightarrow{e} VClosure\ ""\ vl\ e} \quad (3.4)$$

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

For the next rule we introduce $no\_previous\_match\ i\ \Delta\ \Gamma\ cs\ v := (\forall j : nat,\ j < i \to match\_clause\ v\ cs\ j = None \vee (\forall\ (gg, ee : Expression),\ (bb : list\ (Var \times Value),\ match\_clause\ v\ cs\ j = Some\ (gg,\ ee,\ bb) \to ((add\_bindings\ bb\ \Gamma, \Delta, gg) \xrightarrow{e} ff)))$.

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

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

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

$$\frac{\begin{array}{c} eval\_all\ \Gamma\ \Delta\ exps\ vals \\ (append\_vars\_to\_env\ vars\ vals\ \Gamma, append\_vars\_to\_closure\ vars\ vals\ \Delta\ \Gamma, e) \xrightarrow{e} v \end{array}}{(\Gamma, \Delta, ELet\ vars\ exps\ e) \xrightarrow{e} v} \quad (3.10)$$

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

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

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

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

– 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 be-

haviour of inter-module function calls. This results in a value which will be the result of the *ECall* evaluation.
- Rule 3.9: 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 referenced environment of the closure. This referenced environment can be acquired from the closure environment.
- Rule 3.10: At first, every expression given must evaluate to values. Then the body of the `let` expression must be evaluated in the original environment extended with the variable-value bindings. The closure environment has to be extended also in order to define functions in `let`, so for every value here the *append_vars_to_closure* function decides, if it is a closure, and adds the variable-closure binding to the environment (also modifies the closure's environment reference, as mentioned before).
- Rule 3.11: This rule is very similar to the rule 3.10, except no expressions must be evaluated here at first. From the described functions (list of variable list and body expressions), closures will be created and appended to the environment and closure environment. In these ones the evaluation continues.
- Rule 3.12 Introduces the evaluation for maps. This rule states that every key in the key list and value list must be evaluated to values resulting in two value lists (for keys and values) from which the value map is constructed.

After discussing these rules, we show an example why the other approach (where values are defined as a subset of expressions) is more difficult to use. 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: $Value ::= \{e : Expression \mid e\ val\}$. Let see how does this would modify our semantics in a few key points:

- The type of the *eval_expr* would be $Environment \times Closures \times Expression \to Expression \to Prop$. This implies, that a theorem is needed that states about the expression evaluation to values.
- Because of the strictness of Core Erlang, the derivation rules change, for example:
  - The expression literals could not be rewritten (because those are values)
  - Function definitions must be handled as values
  - Additional checks needed in the preconditions, e.g. the rule 3.6:

$$\frac{\begin{array}{cc} tlv\ val & (\Gamma, \Delta, hd) \xrightarrow{e} hdv \vee hd = hdv \\ hdv\ val & (\Gamma, \Delta, tl) \xrightarrow{e} tlv \vee tl = tlv \end{array}}{(\Gamma, \Delta, EList\ hd\ tl) \xrightarrow{e} VList\ hdv\ tlv}$$

- Additional theorems are needed, e.g. from values there is no possible rewriting

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

Proof about properties of the semantics When proving statements about inductive types, induction principles are needed. These principles describe, how a P property can be proved for every element of an inductive type. For example the induction principle for natural numbers is the following.

To prove $\forall n : nat,\ P(n)$ it is enough to prove:

1. $P(0)$
2. $\forall n : nat,\ P(n) \to P(S\ n)$

At this point, we encountered a difficulty. Coq uses a mechanism, that can guess these principles based on the construction of the inductive type. Unfortunately this mechanism does not work always, like in the case of rose trees (trees can be constructed with the $Empty$ and the $Node\ (l : list\ RoseTree)$ constructors) it can not guess the second part of the principle.

To prove $\forall t : RoseTree, P(t)$ it is enough to prove:

1. $P(Empty)$
2. $\forall l : list\ RoseTree,\ P(Node\ l)$ which should be $\forall l : list\ RoseTree,\ (\forall e : RoseTree,\ In\ e\ l \to P(e)) \to P(Node\ l)$

Nevertheless, Coq also allows to define these principles by hand (e.g. as hypotheses), and this approach was needed for expressions and the operational semantics by far. After this solution, we managed to formalise and prove theorems. In this paper we present two of these.

Theorem 1 (Determinism). $\forall$ *(Γ : Environment), (Δ : Closures), (e : Expression), (v_1 : Value), (Γ, Δ, e)* $\xrightarrow{e}$ $v_1 \Rightarrow$ *($\forall v_2$: Value, (Γ, Δ, e)* $\xrightarrow{e}$ $v_2 \Rightarrow v_1 = v_2$*).*

Proof. Induction by the construction of the semantics.

- 3.1, 3.2, 3.3 and 3.4 are trivial. E.g. a value literal can only be derivated from its expression counterpart.
- 3.5 and 3.12 are similar, 3.12 is basically a double tuple. 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 proofs for maps is similar.
- 3.6 The head and the tail of the list can be evaluated to a single head and tail value, so the result list constructed from these ones only one way.
- 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 proofs can be constructed based on parts of the previous ones.

□

Theorem 2 (Commutativity). $\forall$ *(e, e' : Expression), (Δ : Closures), (Γ : Environment), (t : Value),*

(Γ, Δ, ECall "plus"%string [e ; e']) $\xrightarrow{e}$ $t \Leftrightarrow$
(Γ, Δ, ECall "plus"%string [e' ; e]) $\xrightarrow{e}$ t.

Proof. The proof is the same to both directions of the equivalence, so only the $\Rightarrow$ direction is described here.

First the information is needed about the construction of the main hypothesis (in this case: (Γ, Δ, *ECall* "plus"%*string* [e ; e'])$\xrightarrow{e} t$). This information is two values to which e and e' evaluates.

With the help of these values (in a swapped order), 3.8 can be applied to the goal. There three statements has to be proven:

- The length of the parameter list is the same as the variable list: both are two.
- The parameters evaluate to the given values: the information from the main hypothesis includes these statements.
- The auxiliary *eval* function with the *"plus"* parameter is commutative: this function is represented with the mathematical addition, that is why it is commutative (if one of the parameters is not a number, then it will result an error value).

□

The complete proofs of these theorems (along with examples) are available in Coq on the project's Github repository [2].

4 Application and testing of the semantics

In the previous section we have defined a big-step operational semantics for the sequential part of the Core Erlang language, which we also formalised in the Coq proof assistant.

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

4.1 Testing of 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 official compiler.

To test our formal semantics, we used equivalence partitioning. We have written tests both in Coq and in Core Erlang (OTP version 22.0) for every expression defined in our formalisation. Moreover, there have also been special complex expressions that have needed separate test cases (e.g. using bound variables in `let` expressions, application of recursive functions, etc.). All test cases are available in our Github repository [2].

4.2 Formal program evaluation

Now let us demonstrate how Core Erlang programs are evaluated in the formal semantics. For the sake of readability, we use concrete Core Erlang syntax in the

proofs, and trivial statements are omitted from the proof tree. All examples are formalised in Coq, available in our Github repository [2].

The first example shows how to evaluate a simple expression with binding:

$$\dfrac{\dfrac{\overline{\{X:5\}(X)=5}}{(\{X:5\},\varnothing,X)\xrightarrow{e}5}\,3.2}{(\varnothing,\varnothing,\text{ let } X=5 \text{ in } X)\xrightarrow{e}5}\,3.10$$

The second example is intended to demonstrate the purpose of the closure environment. 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 environment.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\{X:42\}(X)=42}}{(\{X:42\},\{Y:\{X:42\}\},X)\xrightarrow{e}42}\,3.2}{(\{X:5,Y:\mathit{VClosure}\ Y\ []\ X\},\{Y:\{X:42\}\},\mathit{apply}\ Y())\xrightarrow{e}42}\,3.9}{(\{X:42,Y:\mathit{VClosure}\ Y\ []\ X\},\{Y:\{X:42\}\},\mathit{let}\ X\ =5\ \mathit{in}\ \mathit{apply}\ Y())\xrightarrow{e}42}\,3.10}{(\{X:42\},\varnothing,\text{ let } Y=\mathit{fun}()\to X\ \mathit{in}\ \mathit{let}\ X=5\ \mathit{in}\ \mathit{apply}\ Y())\xrightarrow{e}42}\,3.10}{(\varnothing,\varnothing,\text{ let } X=42 \text{ in let } Y=\mathit{fun}()\to X\ \mathit{in}\ \mathit{let}\ X=5\ \mathit{in}\ \mathit{apply}\ Y())\xrightarrow{e}42}\,3.10$$

The third example cannot be evaluated in our formalisation, because of infinite recursion. Let $\Gamma := \{'x'/0 : \mathit{VClosure}\ 'x'/0\ []\ \mathit{apply}'x'/0()\}$ (the environment after the binding is added).

$$\dfrac{\dfrac{\dfrac{\ldots}{(\Gamma,\{'x'/0:\Gamma\},\text{ apply } 'x'/0())\xrightarrow{e}??}\,3.9}{(\Gamma,\{'x'/0:\Gamma\},\text{ apply } 'x'/0())\xrightarrow{e}??}\,3.9}{(\varnothing,\varnothing,\text{ letrec } 'x'/0=\text{fun}()\to\text{ apply } 'x'/0() \text{ in apply } 'x'/0())\xrightarrow{e}??}\,3.11$$

4.3 Program equivalence proofs

Last but not least, let us expose some program 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. Like before, the machine-checked proofs of these theorems are available in the project's Github repository [2].

First, we present a rather simple example of program 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. The formal description of the example looks like the following (using abstract syntax for this one step):

$$
\begin{aligned}
&\forall t : Value,\\
&(\varnothing, \varnothing, ELet\ [\text{“}X''] \ [ELiteral\ (Integer\ 5)](ELet\ [\text{“}Y''] \ [ELiteral\ (Integer\ 6)]\\
&\quad (ECall\ \textit{“plus”}\ [EVar\ \text{“}X''; EVar\ \text{“}Y''])) \xrightarrow{e} t \Longleftrightarrow\\
&(\varnothing, \varnothing, ELet\ [\text{“}X''] \ [ELiteral\ (Integer\ 6)](ELet\ [\text{“}Y''] \ [ELiteral\ (Integer\ 5)]\\
&\quad (ECall\ \textit{“plus”}\ [EVar\ \text{“}X''; EVar\ \text{“}Y''])) \xrightarrow{e} t
\end{aligned}
$$

Both directions of this equivalence are proven exactly the same way, so only the $\Longrightarrow$ direction is presented here. This way, the hypothesis is the left side of the equivalence.

First, this hypothesis should be decomposed. From the two `let` statements, it is known that the 5 and 6 expression literals can be evaluated only to their value counterparts (because of the determinism). These ones will be associated with X and Y in the evaluation environment for the addition operator (*ECall* “plus”). When this statement is evaluated, then it yields the following hypothesis:

$$t = eval\ \textit{“plus”}\ [VLiteral\ (Integer\ 5); VLiteral\ (Integer\ 6)]$$

Furthermore, our goal can be proven with the derivation tree presented below. In this tree the trivial parts of the proofs are not described for readability (these are e.g. that the 5 and 6 expression literals evaluate to their value counterparts, the length of the expression or variable lists are the same as the evaluated value lists, etc.).

$$
\dfrac{\dfrac{\dfrac{eval\ \textit{“plus”}\ \ [6;5] = t}{(\{X:6, Y:5\}, \varnothing, X+Y) \xrightarrow{e} t}\ 3.8}{(\{X:6\}, \varnothing, let\ Y = 5\ in\ X+Y) \xrightarrow{e} t}\ 3.10}{(\varnothing, \varnothing, let\ X = 6\ in\ let\ Y = 5\ in\ X+Y) \xrightarrow{e} t}\ 3.10
$$

The only remaining goal is to prove that $eval\ \textit{“plus”}\ \ [6;5] = t$. We have already stated, that $t = eval\ \text{“}plus\text{”}\ [5;6]$, so it is sufficient to prove:

$$eval\ \textit{“plus”}\ [6;5] = eval\ \textit{“plus”}\ [5;6]$$

The commutativity of the mathematical addition can be used here (because of the representation of `eval`), so we can swap the 5 and 6 values in the parameter list. After this modification, we get reflexivity. □

With the help of the same chain of thoughts, a more abstract refactoring also can be proven in our system.

Example 2 (Swapping variable expressions). If e_1 and e_2 does not contain the variables X and Y, then

let X = e1 **in let** Y = e2 **in** X + Y

is equivalent to

let X = e2 **in let** Y = e1 **in** X + Y

Proof. Similar to the Example 1, at first, this one should also be described in our formalisation. We prove this example with initial empty environment, but this environment could be generalized.

$$
\begin{aligned}
&\forall t : Value,\\
&(\varnothing, \varnothing, ELet\ [\text{“}X''\text{”}]\ [e_1]\ (ELet\ [\text{“}Y''\text{”}]\ [e_2]\\
&\qquad (ECall\ \text{“}plus''\ [EVar\ \text{“}X''; EVar\ \text{“}Y''])) \xrightarrow{e} t \Longleftrightarrow\\
&(\varnothing, \varnothing, ELet\ [\text{“}X''\text{”}]\ [e_2]\ (ELet\ [\text{“}Y''\text{”}]\ [e_1]\\
&\qquad (ECall\ \text{“}plus''\ [EVar\ \text{“}X''; EVar\ \text{“}Y''])) \xrightarrow{e} t
\end{aligned}
$$

The directions of this equivalence are proven exactly the same way, so only the $\Longrightarrow$ direction is presented here.

Now the main hypothesis has two `let` statements in itself. Similarly to the Example 1, these statements could only be derivated with rule 3.10, i.e. there are two values (v_1 and v_2) to which e_1 and e_2 evaluates. It is important to mention, that these evaluations can happen in every environment which contains only X and Y (same goes for the closure environment)[4], because neither e_1 nor e_2 contains X and Y. Moreover there appeared also a hypothesis: ({X : v_1', Y : v_2'}[5], Δ, *ECall* “plus” [*EVar* “X” ; *EVar* “Y”]) $\xrightarrow{e}$ t (for some Δ which is not giving us any useful additional information). This hypothesis implies that $t =$ *eval “plus”* $[v_1', v_2']$.

Furthermore, the goal can be solved with the construction of a derivation tree.

$$
\dfrac{\dfrac{\dfrac{eval\ \text{“}plus\text{”}\ [v_2'', v_1''] = t}{(\{X : v_2'', Y : v_1''\}, \Delta', X + Y) \xrightarrow{e} t}\ 3.8}{(\{X : v_2''\}, \Delta, let\ Y = e_1\ in\ X + Y) \xrightarrow{e} t}\ 3.10}{(\varnothing, \varnothing, let\ X = e_2\ in\ let\ Y = e_1\ in\ X + Y) \xrightarrow{e} t}\ 3.10
$$

[4] If we are trying to prove for an initial environment which is not empty, then here these environments can be the initial environment, or its extension with X or Y or both.

[5] $v_1' = v_1$ if v_1 is not a closure, in that case v_1 can be written like this: *VClosure* “” *varlist body*. Then let v_1' be *VClosure* “X” *varlist body*. Same goes for v_2'. This behaviour is present because of the *append_vars_to_env* function.

As mentioned before, e_1 and e_2 evaluates to v_1 and v_2 in the initial environment (in this case it is $\varnothing$, but this could be generalized) and the extended environments (X or Y or both appended to the initial environment) too. So when the rule 3.10 applies, we can give a proof that e_2 and e_1 evaluates to v_2 and v_1. Because of the definition of *append_vars_to_env*, the `lets` can modify these values to v_2'' and v_1'' just like before ($v_1' = v_1''$ is not always true, if they are closures, then their referenced environment differs), so these values are inserted in the environment associated with X and Y.

After making this statement, we can use the rule 3.8 to evaluate the "plus". The parameter variables will evaluate to v_2'' and v_1''. 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:

$$eval \text{ "plus" } [v_2'', v_1''] = eval \text{ "plus" } [v_1', v_2']$$

Moreover, $v_1' = v_1''$, if they are not closures (same is true for v_2' and v_2''), so for them, the commutativity of `eval` can be used to solve this equality. If either of v_1' or v_2' is a closure, then its v_1'' or v_2'' pair is a closure too, so in both side the `eval` function will result the same error value, and these ones are equal. □

To prove these examples in Coq, a significant number of lemmas were needed (like the exposition of lists, the commutativity of the `eval`, etc.), however the proof mostly consists of the combination of hypotheses similar to the proofs in this paper. Although sometimes additional case separations were needed which resulted in lots of subgoals, these ones were solved very similarly producing code duplication. In the future, these proofs should become simpler with the introduction of smart tactics and additional lemmas.

Moreover, in the concrete implementation for Example 2 we used another thought as the first hypothesis: If e_1 does not contain the variables X and Y, then it will evaluate to the same value in the environments combined from these variables. This statements also stands for e_2.

4.4 Evaluation

We showed that our formal semantics is a powerful tool. We managed to formalise and prove theorems, programs, program equivalence examples. This proves that the semantics is usable indeed. With this one we have a powerful tool to argue about sequential Core Erlang programs. In the previous sections we also mentioned some other approaches to formalise this semantics, and showed why our way is more usable for our purpose.

On the other hand, it also can be seen that this formalisation is not simple to use either in practice, partly because the Coq Proof Assistant makes its users to write down every triviality too. Of course this is a necessity of the correctness, however, this property results in complex proofs. As a possibility for future work, it would be very useful to create smart tactics, to simplify out proofs and examples. In addition, this semantics is not complete yet, so it cannot be used for any Core Erlang expression.

5 Summary

5.1 Future work

There are several ways to enhance our formalisation, we are going to focus mainly on these short term goals:

- Extend semantics with additional expressions (e.g. `try`);
- Handle errors (`try` statement);
- Handle and log side effects;
- Create new lemmas, theorems and tactics to shorten the Coq implementation of the proofs;
- Formalise and prove more refactoring strategy.

Our long term goals include:

- Advance to Erlang (semantics and syntax);
- Distinct primitive operations and inter-module calls;
- Formalize the parallel semantics too.

The final goal of our project is to change the core of a scheme-based refactoring system to a formally proven core.

5.2 Conclusion

In this study, we discussed why a language formalisation is needed, then briefly the goal of our project (to prove refactoring correctness). To reach this objective, Erlang was chosen as the prototype language, then several existing Erlang formalisations were compared. Based on these ones, a new natural semantics was introduced for a subset of Erlang. This one was also formalised in Coq Proof Assistant along with essential theorems, proofs (like determinism) and formal expression evaluation examples. We also showed proofs about the meaning-preservation of simple refactoring strategies with our formal semantics. In the future, we are intended to extend this formalisation with additional Erlang statements, error handling and more equivalence examples.

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. The Coq Proof Assistant Documentation. https://coq.inria.fr/documentation, accessed: 2020-01-08
2. Core Erlang Formalization. https://github.com/harp-project/Core-Erlang-Formalization/tree/syntacitcal-values, accessed: 2020.01.10
3. Carlsson, R., Gustavsson, B., Johansson, E., Lindgren, T., Nyström, S.O., Pettersson, M., Virding, R.: Core Erlang 1.0 language specification (2004)
4. Fredlund, L.Å.: A framework for reasoning about Erlang code. Ph.D. thesis, Mikroelektronik och informationsteknik (2001)
5. Fredlund, L.Å., Gurov, D., Noll, T., Dam, M., Arts, T., Chugunov, G.: A verification tool for Erlang. International Journal on Software Tools for Technology Transfer **4**(4), 405–420 (2003)
6. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: Cauder: a causal-consistent reversible debugger for Erlang. In: International Symposium on Functional and Logic Programming. pp. 247–263. Springer (2018)
7. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. Journal of Logical and Algebraic Methods in Programming **100**, 71–97 (2018)
8. Lanese, I., Sangiorgi, D., Zavattaro, G.: Playing with Bisimulation in Erlang. In: Models, Languages, and Tools for Concurrent and Distributed Programming, pp. 71–91. Springer (2019)
9. Neuhäußer, M., Noll, T.: Abstraction and model checking of Core Erlang programs in Maude. Electronic Notes in Theoretical Computer Science **176**(4), 147–163 (2007)
10. Nishida, N., Palacios, A., Vidal, G.: A reversible semantics for Erlang. In: International Symposium on Logic-Based Program Synthesis and Transformation. pp. 259–274. Springer (2016)
11. Vidal, G.: Towards symbolic execution in Erlang. In: International Andrei Ershov Memorial Conference on Perspectives of System Informatics. pp. 351–360. Springer (2014)