Towards a Proof-Assistant-Based Formalization of Erlang

Péter Bereczky[1]

[1]Faculty of Informatics, Eötvös Loránd University, Hungary

November 27, 2019

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 argue about the definitions of a programming language and the software that has been built with it, we need a scientifically rigorous description of that language. In this paper, we present our preliminary, proof-assistant-based formalization of a subset of Erlang, intended to serve as a base for refactoring correctness proofs in the future. After discussing how we reused concepts from related work, we show the syntax and semantics of our formal description, including involved abstractions (e.g. closures). We also present essential properties (e.g. determinism) along with their machine-checked proofs.

Contents

1 Introduction **3**

2 Comparing previous work **3**
2.1 Core Erlang 4

3 Language Syntax **4**
3.1 Values 5

4 Dynamic Semantics **6**
4.1 Environment 6
4.2 Closures 7
4.3 The semantic rules of Core Erlang 7
4.4 Theorems 10

5 Coq implementation **10**
5.1 Syntax 10
5.2 Semantics 10
5.3 Theorems and proofs 11

6 Examples **11**
6.1 Program examples 11
6.2 Equivalence example 12

7 Future work **12**

8 Summary **13**

1 Introduction

Arguing about the correctness of software is challenging because most of the programming languages nowadays do not have a formal description (or the developers do not use it). To discuss this topic about a program, it is not enough to have an informal definition and knowing how the code was implemented. A formal description is needed for the programming language. Of course, describing the abstract syntax and formal semantics of a language is not an easy task.

In addition, there are also a lot of refactoring tools that can change parts of the source code without modifying the meaning of a program, but what guarantees this behavior? What if there was a formal definition of the language which is used in the refactoring process. If this one were present, proofs could be written about the refactorings or refactoring schemes (e.g. renaming a variable, defining a function, etc.).

The initial idea of this project was to formally prove the correctness of an existing refactoring tool that used refactoring schemes. Moreover, it would be beneficial, if the formal definition of the language were implemented too. This way allows easier modifications in the future and a degree of automation (executable semantics).

This paper focuses on the formalization (a natural semantics it is possible) of a sequential subset of Erlang in a machine-checked way with the help of a proof system.

2 Comparing previous work

Although there have already been a lot of Erlang or Erlang subset formal definitions, neither of them is implemented with a machine-checked proof system. So the main idea of the project was to create an executable formal semantics for the sequential subset of Erlang in the Coq Proof Assistant based on existing formalizations in the referenced articles and documents.

Unfortunately, most of the papers focus on the parallel parts of Erlang or Erlang subsets which is not suitable for our goal, because we want to investigate the sequential parts only, however, the abstract syntax of the language is basically the same in every paper which has been found by us, so the abstract syntax of out formal semantics was based on these (especially [4] and [5]).

We had found only a few papers that had included the semantics of the sequential parts of Erlang (or Erlang subsets), but our definition was able to be built from these. Especially from [5], because its language is "basically equivalent to a subset of Core Erlang". It's worth mentioning that this article presents a small-step operational semantics for this subset, so we can not use it without modifications, however, our semantics can be based on it in a few aspects. Moreover, this paper does not take Erlang functions and their closures into consideration except the top-level functions, that we also aim to formalize.

Besides, there was another source, we used for our formalization which is the Core Erlang Language Specification [1]. We have found that there are some simplifications in [5] which we do not intend to use (e.g.the `let` statement should bind multiple variables at once, the `letrec` statement) so we needed at least the informal definitions of these. Also, in the article [5] the global environment is modified in every step of the execution. Our semantics does not work like this, because the side-effects have been not implemented yet. Unfortunately, the language specification was written in 2000, and there were some new features introduced to Erlang (and Core Erlang). For these reasons, we extended this formalization in a few aspects.

2.1 Core Erlang

In the previous section, we mentioned Core Erlang many times. We chose to formalize a sequential subset of Core Erlang due to the following reasons:

- The number of sources about the formalization of Core Erlang or its subsets.
- Core Erlang is a subset of Erlang.
- From Erlang, Core Erlang can be compiled (e.g. with `to_core` switch). Moreover, the compilation process of Erlang contains Core Erlang translation.
- Some of the modern functional languages also translate to Core Erlang (e.g. Elixir), so the semantics of it could be useful not only in our project.

3 Language Syntax

The language syntax is based on [5] and the documentation of Core Erlang [1].

`Var` ::= `string`
`ErlModule` ::= $s, f_1, ..., f_n$
`ErlFunction` ::= $fsig, fun$
`FunctionSignature` ::= $string \times \mathbb{N}$
`Literal` ::= Atom s | Integer z | [] [a] | {} [b] | #{} [c]
`Pattern` ::= X | l | $[p_1 \mid p_2]$ | $\{p_1, .., p_n\}$
`Expression` ::= l | X | fun | $[hd|tl]$ | $\{e_1, .., e_n\}$
| call $fsig(e_1, .., e_n)$
| apply $X(e_1,.., e_n)$
| apply $fsig(e_1,.., e_n)$
| let $X_1, .., X_n = e_1, .., e_n$ in e
| letrec $fsig_1, .., fsig_n = fun_1, .., fun_n$ in e
| case e of $cl_1,..,cl_n$
| $\#\{ek_1 \rightarrow ev_1, .., ek_n \rightarrow ev_n\}$ [d]
`Clause` ::= p when $guard \rightarrow e$
`Fun` ::= fun$(X_1, .., X_n) \rightarrow e$

[a] Empty list
[b] Empty tuple
[c] Empty map
[d] This is a map structure and ek, ev are lists of Expression

Figure 1: Syntax of Core Erlang

Consider p a `Pattern`, $e, guard, hd, tl$ an `Expression`, l a `Literal`, X, Y, Z a `Var`, $fsig$ a `FunctionSignature`, cl a `Clause`, fun a `Function`, f an `ErlFunction`, s a `string` (and also their indexed and comma versions).

The syntax of the language can be found in Figure 1. For example `Expression` can be constructed with a constructor named `call` with parameters of `FunctionSignature` and n `Expression` types.

So a Core Erlang module consists of top-level functions (`ErlFunctions`). Every one of these has a function signature with a name and the arity, parameter names (variables)

and body expression. Expressions can be constructed in various ways, some of them are self-explanatory, but we provide some explanation:

- Literals, variables, functions, lists, tuples are `Expressions`.
- We have not distinct inter-module calls and primitive operations yet, both can be constructed with `call`.
- We distinguish the application of variable (local, defined with `let` statement) and signature (top-level or recursive defined with `letrec` statement) functions, and these can be constructed with `apply` using different parameters.
- With the `let` expression local variables can be defined.
- With the `letrec` expression local (recursive) functions can be defined.
- With the `case` conditional expressions can be defined.

Patterns and expressions are distinct types because according to [1] patterns also can have constructors, that expressions do not.

For more details about the syntax, visit [1] and [5].

There was another existing approach to define abstract syntax. This is written in [2], [3] and [6]. The main difference between this and the presented approach is that in these papers the authors distinguish values from expressions syntactically. Taking the fact into consideration, that later we want to formalize Core Erlang in Coq, it is easier not to distinct values this way, because this will result in several additional types.

3.1 Values

Everything in Core Erlang evaluates to values, so it is needed to have a value representation in this formalization. We construct the values with a unary operator (val). This concept is based on the syntactic value definition in [2], [3] and [6].

Definition 3.1. `ValueJudgment` $\subseteq$ `Expression`

$$\frac{}{l \text{ val}} \tag{3.1}$$

$$\frac{}{fun \text{ val}} \tag{3.2}$$

$$\frac{hd \text{ val} \wedge tl \text{ val}}{([hd|tl]) \text{ val}} \tag{3.3}$$

$$\frac{\forall i \in [1..n] : e_i \text{ val}}{\{e_1, .., e_n\} \text{ val}} \tag{3.4}$$

$$\frac{\forall i \in [1..n] : ev_i \text{ val} \wedge ek_i \text{ val}}{\#\{ek_1 \to ev_1, .., ek_n \to ev_n\} \text{ val}} \tag{3.5}$$

Now let us look at this unary operator. 3.1 and 3.2 states that every literal and function is a value. 3.3, 3.4 and 3.5 claim the same about lists, tuples, and maps only and only if the parameters of these constructions are values.

With the help of this relation, we can define the type `Value`:

Definition 3.2. `Value` ::= $\{e : \texttt{Expression} \mid e \text{ val}\}$

So the `Value`s are a subset of the `Expression`s. It is worth mentioning that errors and exceptions are also values. We will use this type in the environment later.

Remark: In Coq, these types can be defined with set restriction. The result is similar to product types. Its elements consist of an `Expression` and a proof stating that this expression is a value (using the `ValueJudgement`). According to proof irrelevancy, two `Value`s are equal, only and only if their expressions are equal. The proofs are not taken into consideration.

4 Dynamic Semantics

4.1 Environment

In this section consider X, X_1, ... as elements of `Var + FunctionSignature` type.

Environments are needed to know which variables are bound to which values. Moreover, in Core Erlang, there are also function signatures that can be bound to function expressions (these are top-level functions or can be defined with the `letrec` statement), let us call them "signature" functions.

We can state, that the environment stores values, because Core Erlang evaluation is strict, so before binding a variable to an expression, that one will be evaluated to a value.

Our first thought was to distinguish the top-level and local environments. The top-level environment had stored the "signature" functions, the local everything else. On the other hand, this decision made a lot of duplication in the semantic rules and in the actual Coq implementation too. For this reason, we decided to use union type to construct a global environment for function names and variables too.

Variable and function signature binding can happen when processing parameter lists or `let`, `letrec`, `case` statements. These bindings are mappings to values, so the environment is actually a map which can be represented with a list of pairs:

Definition 4.1. `Environment` ::= $[(\texttt{Var + FunctionSignature}) \times \texttt{Value}]$

In the following, we denote this mapping with θ, $\varnothing$ if it is empty and the update of an existing environment with $\theta[X_1 \to e_1, .., X_n \to e_n]$ [1]. The associated values can be get by the $\theta(X)$ notation.

Remark: At first, it seemed easier to implement the environments not with values, but with expressions, however, this way does not fit well with the fact that Core Erlang evaluation is strict, so every expression will be evaluated to values before binding to a variable (or function signature). Furthermore, it would be much harder to prove some theorems.

[1] When X_i and X_j are equal, then their updated value will be e_i if $i > j$ e_j otherwise. In fact Core Erlang assumes the uniqueness of these variables [1].

4.2 Closures

Why are closures needed and what are closures? Let us see an example Core Erlang program.

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

In this case, what should return the Y function? 5 or 10? How can the behavior be described in both cases (especially in the second)?

If 10 were returned, then we could state that the result of the function depends on the concrete evaluation environment. This means that the function could be modified outside of its body. This is usually not wanted behavior, that is why most programming languages (Core Erlang too) return 10 in this case. The problem is solved with closures.

When defining a function (either with `let` or `letrec` statement), a closure is created. This is a copy of the actual environment associated with the name of the function (so it is a mapping from the function's name or signature to its definition environment).

Definition 4.2. `Closures` ::= [(`Var + FunctionSignature`) × `Environment`]

When a function application happens, then the evaluation environment should use this closure. Let's see the example.

In this case, we want to return 5, and not 10. So when applying the Y function, it must not be evaluated in the actual environment $\{X \to 10\}$, but in the one where Y was defined $\{X \to 5\}$. The closure associated with Y 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.

In the case of referencing a function (and not evaluating it), Core Erlang results with the closure associated with the function name. How to implement this behavior is still under discussion.

In the future, we denote the list of closures (elements from the `Closures` type) with ϕ and $\varnothing$ if it is empty. We mark the update for existing closures with $\phi[X_1 \to fun_1, .., X_n \to fun_n]$.

4.3 The semantic rules of Core Erlang

Eventually, everything is ready to define the big-step semantics for Core Erlang. In the following rules, we introduce $evalParams$ for readability.

$evalParams ::= \forall i \in [1..n] : ((\theta, \phi, e_i) \xrightarrow{e} e_i' \vee e_i = e_i') \wedge e_i'$ val

$evalParams$ states, that every e_i expression will be evaluated to e_i' value, or e_i is already a value, so it cannot be evaluated (therefore it is equal to e_i'). This is intended to capture the strictness of Core Erlang.

Planning decisions

- We preferred natural (big-step) semantics over small-step because it's more convenient to use and it can fulfill our aim.

- Our decision was not to include a reflexive rule (which states that every expression can be rewritten to itself) in our formalization. Although, that way the semantics would be much easier to use (and describe in Coq), the proving (especially for determinism) would become very hard.
- We based our solution on the small-step semantics of [5] and on the language specification [1].

Definition 4.3 (The dynamic semantics of Core Erlang).
$\xrightarrow{e} \subseteq (\texttt{Environment} \times \texttt{Closures} \times \texttt{Expression}) \times \texttt{Expression}$

$$\frac{}{(\theta, \phi, X) \xrightarrow{e} \theta(X)} \tag{4.1}$$

$$\frac{evalParams \quad \{e_1, .., e_n\} \neq \{e'_1, .., e'_n\}}{(\theta, \phi, \{e_1, .., e_n\}) \xrightarrow{e} \{e'_1, .., e'_n\}} \tag{4.2}$$

$$\frac{\begin{array}{lll}(\theta, \phi, hd) \xrightarrow{e} hd' \vee hd = hd' & & hd' \text{ val} \\ (\theta, \phi, tl) \xrightarrow{e} tl' \vee tl = tl' & tl' \text{ val} & [hd|tl] \neq [hd'|tl']\end{array}}{(\theta, \phi, [hd|tl]) \xrightarrow{e} [hd'|tl']} \tag{4.3}$$

$$\frac{evalParams \quad eval(\theta, fsig, e'_1, .., e'_n) = e'}{(\theta, \phi, \text{ call } fsig\ e_1, .., e_n) \xrightarrow{e} e'} \tag{4.4}$$

$$\frac{evalParams \quad (\phi(fsig)[X_1 \to e'_1, .., X_n \to e'_n], \phi, \theta(fsig)) \xrightarrow{e} e' \vee e = e' \quad e' \text{ val}}{(\theta, \phi, \text{ apply } fsig\ e_1, .., e_n) \xrightarrow{e} e'} \tag{4.5}$$

$$\frac{evalParams \quad (\phi(X)[X_1 \to e'_1, .., X_n \to e'_n], \phi, \theta(X)) \xrightarrow{e} e' \vee e = e' \quad e' \text{ val}}{(\theta, \phi, \text{ apply } X\ e_1, .., e_n) \xrightarrow{e} e'} \tag{4.6}$$

$$\frac{\begin{array}{c}evalParams \\ (\theta[X_1 \to e'_1, .., X_n \to e'_n], \phi[X_{i1} \to \theta, .., X_{ik} \to \theta], e) \xrightarrow{e} e' \vee e = e' \quad e' \text{ val}\end{array}}{(\theta, \phi, \text{ let } X_1, .., X_n = e_1, .., e_n \text{ in } e) \xrightarrow{e} e'} \tag{4.7}$$

For the next rule let's consider $\theta' := \theta[fsig_1 \to fun_1, .., fsig_n \to fun_n]$.

$$\frac{(\theta', \phi[fsig_1 \to \theta', .., fsig_n \to \theta'], e) \xrightarrow{e} e' \vee e = e' \quad e' \text{ val}}{(\theta, \phi, \text{ letrec } fsig_1, .., fsig_n = fun_1, .., fun_n \text{ in } e) \xrightarrow{e} e'} \tag{4.8}$$

$$\frac{evalParams \quad \forall i \in [1..n] : ek_i \text{ val} \quad \{ev_1, .., ev_n\} \neq \{ev'_1, .., ev'_n\}}{(\theta, \phi, \#\{ek_1 \to ev_1, .., ek_n \to ev_n\}) \xrightarrow{e} \#\{ek_1 \to ev'_1, .., ek_n \to ev'_n\}} \tag{4.9}$$

We wanted to ensure that our semantics was deterministic, so we did not include reflexive rule. In this way, in *evalParams* the statement $... \vee e_i = e_i' ...$ was needed, because some of the e_i expressions could be values, which cannot be evaluated to other expressions. One additional assertion was needed, that the original expression was not the same as the evaluated, because that would allow the evaluation of values (e.g. $[] = []$ and $[]$ is a value, so $(\varnothing, \varnothing, \{[]\}) \xrightarrow{e} \{[]\}$).

Now let us look at the individual rules in detail.

- 4.1 claims that a variable will be evaluated to its stored value in the θ environment.
- 4.2 claims that every expression is the tuple will be evaluated to a value and the result tuple will contain these values in the original order.
- 4.3 claims that the head and tail of the list will be evaluated to values and the result list will have the value head and tail as its head and tail.
- 4.4 claims that every parameter will be evaluated to values and these (with the environment) will be passed to the `eval` auxiliary function which is intended to simulate the behavior of Core Erlang primitive operations and inter-module calls. This `eval` should return a value.
- 4.5 and 4.6 behave similarly. First, both evaluate their parameters to values. Then $\theta(fsig)$ or $\theta(X)$ will tell the associated function's body (which is an expression) that evaluates to the result (e'). But this can only happen in the associated environment which is got by the $\phi(X)$ or $\phi(fsig)$ statements, let us call it θ'. Moreover, this θ' must be extended with the bindings of variables of the function $(X_1, .., X_n)$ to the evaluated parameters. The difference between 4.5 and 4.6 is that 4.5 is written for "signature" (top-level and `letrec` functions), 4.6 for "variable" (local) functions.
- 4.7 claims that every expression on the right-hand side of the equal sign will be evaluated to values. Then the environment will be extended with the "variable to (right-hand side) value" bindings. In addition, the closures must also be updated with the variables which are bound to function expressions and the original environment pairs. After that, the expression in the `in` clause will be evaluated in the updated environment and closures to e' which is the result of the rule.
- 4.8 is very similar to 4.7 except there are no right-hand side expressions that need to be evaluated to values and only function signatures need to be bound to functions. These bindings will happen and result in an updated environment. The closures update is a little tricky. In `letrec` statements recursive functions can be defined. So instead of adding pairs of the function signatures and the original environment to the closures, we pass the updated environment. With the help of these two, the evaluation of e can continue, resulting in e'.
- 4.9 is very similar to 4.2. The expressions in the value list will be evaluated to values (the key list contains values originally), and the result map has the original keys and the evaluated values.

It is important to mention `try` and `case` statement's being under discussion, so this is the reason that we do not present semantic rules for these. In addition, there is no derivation for `Expressions` constructed with `EFunction`. We have not figured out how to convert a closure to a value. Until that, we use `EFunctions` as values.

4.4 Theorems

The following essential theorems are proved in the Coq Proof Assistant.

Theorem 4.1 (Value transition). $\forall e$: `Expression`, e *val* $\implies$ $(\forall t$: *Expression*, $\forall \theta$: `Environment`, $\forall \phi$: `Closures`, $\neg(\theta, \phi, e) \xrightarrow{e} t)$

Proof. Induction by e val.■

Theorem 4.2 (Determinism). $\forall \theta$: `Environment`, $\forall \phi$: `Closures`, $\forall e, v_1$: `Expression`, $(\theta, \phi, e) \xrightarrow{e} v_1 \implies (\forall v_2$: `Expression`, $(\theta, \phi, e) \xrightarrow{e} v_2 \implies v_1 = v_2)$

Proof. Induction by $\xrightarrow{e}$.■

5 Coq implementation

5.1 Syntax

We formalized the types mentioned in Section 3 with (nested) inductive types. Coq generates for every inductive type an induction principle based on the constructors. Unfortunately, this is not correct in the case of nested types, so we had to write our induction principles (these are axioms).

We decided to use lists to implement the $e_1, .., e_n$ expressions in constructors, semantic rules, etc. Fortunately, Coq has built-in list type with proved properties.

`ValueJudgement` was implemented also with an inductive type and a notation (val).

Values Coq has a built-in functionality for set restriction. With the help of this, the `Value` type was created. Basically, `Value`s are pairs that contain an `Expression` and a proof about the expression being a value (e val). An axiom was also created about value comparison. When comparing two `Value`s, it is sufficient to compare only the `Expression`s, if those are equal, the values are equal (this is called proof irrelevancy).

5.2 Semantics

We had had the opportunity to define union types in Coq, so we could describe the type `Value + FunctionSignature` very similarly. The `Environment` and `Closures` were implemented with built-in lists of pairs (with the previous union type). The standard implementation of maps was not used, but it might be in the future. A few helper functions had to be defined, because of this. The inquiry of values and functions (from an `Environment`) or environments (from `Closures`) is possible with the `get_value` or `get_env_from_closure` functions (similar to the $\theta(X)$ or $\phi(X)$). There are also some functions that are intended to update the environment or closures, these are divided into two groups: updating based on `Var`s or `FunctionSignature`s.

Because of these functions, equality was needed between `Var + FunctionSignature` elements. Two of these are equal when both are from the same type and they are equal in that type (two signatures are equal when their name and arity are equal).

The operational semantics was also implemented with the help of the inductive types and an induction principle also was defined for this type. The rules were implemented using standard lists for e'_i expressions. There was an interesting idea to use: when writing a proposition for all elements in a list, let us say P, then there are at least two ways:

- $\forall i : nat, i < n \implies P(l[i])$
- $\forall e : Expression, l.contains(e) \implies P(e)$

We used the second approach, but we may change to the first in the future if we want to formalize side effects.

There are also helper functions which can initialize closures and environment based on a module (described in Coq). The auxiliary `eval` function in the future needs to be extended, because now only supports a few primitive operations (e.g. a commutative `plus` operation for two parameters).

5.3 Theorems and proofs

We managed to formalize both 4.1 and 4.2 in Coq. We also proved them with the help of the induction hypothesis mentioned above (for the operational semantics inductive type).

Value transition This theorem states, that from values no expression can be derivated. We used induction by the construction of values to prove this one. We also needed some helper lemmas about list equality (if from every expression of a list nothing can be derivated, and from these expressions other values can be derivated or the values and the expressions are the same we get, that these lists are the same) when trying to prove the property for maps and tuples.

Determinism This theorem states, that if a value can be derivated from an expression, then all the values which can be derivated from it, are the same. To prove this, we used the theorem about value transition and helper lemmas about list equality with slightly other premises than the abovementioned ones (for every expression constructed with the help of lists e.g. tuples, applications, etc.). Also, an axiom was used which describes the proof irrelevancy.

6 Examples

6.1 Program examples

In order to provide readable documentation, we ignore the trivial proofs (e.g. $evalParams$, e' val). Let us see some program examples to demonstrate our semantics.

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

The next example demonstrates why recursive functions cannot be written in `let` statements. ($\top$ means the error expression.)

$$\dfrac{\dfrac{\dfrac{\overline{}}{(\varnothing, \{X \to \varnothing\}, \text{apply } X()) \xrightarrow{e} \top}}{(\{X \to (\text{fun}() \to \text{apply } X())\}, \{X \to \varnothing\}, \text{apply } X()) \xrightarrow{e} \top}}{(\varnothing, \varnothing, \text{let } X = \text{fun}() \to \text{apply } X() \text{ in apply } X()) \xrightarrow{e} \top}$$

In this example, the derivation terminates abnormally, because X is not defined in the environment. The next example shows the difference between `let` and `letrec`.

$$\frac{\dfrac{\dfrac{\dots}{(\{X \to (\text{fun}() \to \text{apply } X())\}, \{X \to \{X \to (\text{fun}() \to \text{apply } X())\}\}, \text{apply } X()) \xrightarrow{e} ??}}{(\{X \to (\text{fun}() \to \text{apply } X())\}, \{X \to \{X \to (\text{fun}() \to \text{apply } X())\}\}, \text{apply } X()) \xrightarrow{e} ??}}{(\varnothing, \varnothing, \text{letrec } X = \text{fun}() \to \text{apply } X() \text{ in apply } X()) \xrightarrow{e} ??}$$

In this example, the derivation cannot be terminated, because the described statement contains an infinitely recursive function. This is because, when evaluating a `letrec`, the updated (with variables bound to functions) environments will be passed to the closures.

There are also a lot of other examples in Coq in addition to the source code of the formalization.

6.2 Equivalence example

The following example is also proved in Coq.

Theorem 6.1 (Call-plus commutativity)**.** $(\varnothing, \varnothing, \mathit{call}\ (\text{"}\mathit{plus}\text{"}, 2)(e, e')) \xrightarrow{e} t \Leftrightarrow (\varnothing, \varnothing, \mathit{call}\ (\text{"}\mathit{plus}\text{"}, 2)(e', e))$.

Proof. It is sufficient to prove only $\Rightarrow$ direction, because the other way is exactly the same. If this `call` evaluates to t, then (because rule 4.4) $\mathit{eval}((\text{"}\mathit{plus}\text{"}, 2), \varnothing, [e, e']) = t$ (with the `eval` auxiliary function) and according to $\mathit{evalParams}$, $(\varnothing, \varnothing, e) \xrightarrow{e} v_1$ and $(\varnothing, \varnothing, e') \xrightarrow{e} v_2$ where v_1, v_2 are values. With the help of this information, the $(\varnothing, \varnothing,$ call $(\text{"}\mathit{plus}\text{"}, 2)(e', e))$ can be derivated with rule 4.4. There are values, to which e' and e can be evaluated, these are the previous v_2 and v_1, and that satisfies $\mathit{evalParams}$. Only the `eval` equality is left. This states, that $\mathit{eval}((\text{"}\mathit{plus}\text{"}, 2), \varnothing, [e,' e]) = t = \mathit{eval}((\text{"}\mathit{plus}\text{"}, 2), \varnothing, [e, e'])$. Fortunately, the auxiliary function's `plus` operator is commutative (see section 5.2), so these are equal. ■

7 Future work

There are a lot of ways to enhance this formalization, we are focusing mainly on these:

- Extend semantics with the missing rules (`try` and `case`)
- Handle errors (`try` statement)
- Handle and log side effects

Our long term goals:

- 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.

8 Summary

In this study, we discussed why a language formalization 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 formalizations were compared. Based on these ones, a new natural semantics was introduced for a subset of Erlang. This one was also formalized in Coq Proof Assistant along with essential theorems, proofs (like determinism) and examples. In the future, we are intended to extend this formalization with additional Erlang statements, error handling and 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).

References

[1] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. Core Erlang 1.0 language specification. *Information Technology Department, Uppsala University, Tech. Rep*, 2000.

[2] Lars-Åke Fredlund. *A framework for reasoning about Erlang code.* PhD thesis, Mikroelektronik och informationsteknik, 2001.

[3] Lars-Åke Fredlund, Dilian Gurov, Thomas Noll, Mads Dam, Thomas Arts, and Gennady Chugunov. A verification tool for Erlang. *International Journal on Software Tools for Technology Transfer*, 4(4):405–420, 2003.

[4] Martin Neuhäußer and Thomas Noll. Abstraction and model checking of Core Erlang programs in Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):147–163, 2007.

[5] Naoki Nishida, Adrián Palacios, and Germán Vidal. A reversible semantics for Erlang. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 259–274. Springer, 2016.

[6] Germán Vidal. Towards symbolic execution in Erlang. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 351–360. Springer, 2014.