Machine-Checked Natural Semantics for Core Erlang: Exceptions and Side Effects

Péter Bereczky
Dániel Horpácsi
berpeti@inf.elte.hu
daniel-h@elte.hu
Eötvös Loránd University
Budapest, Hungary

Simon J. Thompson
S.J.Thompson@kent.ac.uk
University of Kent
Canterbury, UK
Eötvös Loránd University
Budapest, Hungary

Abstract

This research is part of a wider project that aims to investigate and reason about the correctness of scheme-based source code transformations of Erlang programs. In order to formally reason about the definition of a programming language and the software built using it, we need a mathematically rigorous description of that language.

In this paper, we present an extended natural semantics for Core Erlang based on our previous formalisation implemented with the Coq Proof Assistant. This extension includes the concepts of exceptions and side effects, moreover, some modifications and updates are also discussed. Then we describe theorems about the properties of this formalisation (e.g. determinism), formal expression evaluation and equivalence examples. These equivalences can be interpreted as simple local refactorings.

CCS Concepts: • **Theory of computation → Operational semantics**; **Program verification**; **Functional constructs**.

Keywords: formal semantics, natural semantics, Erlang, Coq, exceptions, side effects

ACM Reference Format:
Péter Bereczky, Dániel Horpácsi, and Simon J. Thompson. 2020. Machine-Checked Natural Semantics for Core Erlang: Exceptions and Side Effects. In *Proceedings of the 19th ACM SIGPLAN International Workshop on Erlang (Erlang '20), August 23, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3406085.3409008

Erlang '20, August 23, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8049-2/20/08…$15.00
https://doi.org/10.1145/3406085.3409008

1 Introduction

There are a number of language processors, development and refactoring tools for programming languages, but most of these tools are not theoretically well-founded: they lack a formally precise description of how exactly the source code is affected by them. In particular, refactoring tools are expected to change programs without modifying their behaviour, but in practice, usually only regression testing is used to verify this property. 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, program semantics and semantic equivalence need to be considered. A formal, mathematical definition of the programming language semantics in question is clearly needed for formal verification. Since the project of which this study is part of is dedicated to improve trustworthiness of Erlang refactorings [15] via formal verification, effort has been put in formalising Erlang and its functional core, i.e. Core Erlang. Erlang (along with other functional languages, e.g. Elixir [12]) translates to Core Erlang as part of the compilation process; thus, a proper formalisation of Core Erlang may contribute to the studies of all languages in the BEAM family. It is worth noting that we do not limit our formalisation to the subset of Core Erlang emitted by the Erlang compiler, but we aim at formalising the entire Core Erlang programming language.

After completing the formalisation of the main features of sequential Core Erlang (see [3]), we started to investigate other language features, such as the concept of exceptions and side effects. This required us to re-iterate each step of our previous work, including surveying related work and making design decisions on level of abstraction and encoding in Coq. To test the semantics, a collection of examples have been written, along with proofs about basic properties of this formalisation, like determinism[1]. In order to demonstrate the applicability of our semantics definition, some simple

[1]In theory, Core Erlang is non-deterministic, but the reference implementation employs a leftmost-innermost evaluation strategy according to Neuhäußer and Noll [24] which complies to the behaviour of the presented formalisation.

expression pattern equivalences have also been formalised and proved; these can be seen as simple local refactorings.

The main contributions of this paper are:

1. The extension of our former formal semantics [3] with the concept of exceptions and side effects.
2. The implementation of this extension in the Coq Proof Assistant (version 8.11.2).
3. Updated and new theorems that formalise a number of properties of this extended formalisation, e.g. determinism, with their machine-checked proofs.
4. Results on program evaluation and equivalence verification using the updated semantics definition, all formalised in the Coq Proof Assistant.

2 Related Work

The ultimate goal of our project is to prove refactoring-related theorems in Coq. This is supported by the formalisation of the semantics of Erlang in Coq in a way that enables flawless verification of program and expression equivalence. Core Erlang was chosen as a stepping stone towards this goal, because not only a subset of Erlang, but Erlang and other functional languages, such as Elixir [12] and LFE, translate to it during compilation. While formalising our semantics, we reviewed and compared extensive related work on both Erlang [10, 11, 17, 27] and Core Erlang [7, 13, 19–21, 24, 25] and chose an approach that can be properly embedded into Coq. For this goal, also the language specification [5] was considered along with the reference implementation (Erlang/OTP version 22.3).

The abstract syntax definitions of Core Erlang were alike in all papers about this language; however, there were slight differences in the abstraction level. For example, in the work of Neuhäußer and Noll [24] `let` expressions can handle multiple variable bindings simultaneously, while the semantics of Nishida et. al. [25] abstracts over this attribute. Unfortunately, in the syntax respect, papers about Erlang could not really be considered, because there are significant differences in the syntax of Core Erlang and that of Erlang.

However, we could borrow ideas from papers on Erlang semantics when deciding on how we formalise values. There were two main approaches to define expressions in normal form: the work of Lanese et al. [19–21, 25] considers values as a subset of patterns ("ground patterns"), while other papers [10, 11, 24, 27] define values as a subset of expressions. In addition, the work of Neuhäußer and Noll [24] also considers functions as values; however, according to the language specification [5] closures are the values of function expressions.

While formalising the dynamic semantics of Core Erlang, the small-step semantics of a Core Erlang-like language defined by Lanese et al. [19–21, 25] was fundamental for our big-step semantics. In addition, some concepts were taken from other papers, such as the recursive closure concept from Reynolds research on definitional interpreters [26] and match expressions from the work of Carlier et. al. [4].

Unfortunately, there were abstractions which were not present in any of the discussed sources (e.g. *map expressions*). For such constructs, we could only rely on the reference implementation and the language specification [5]. Due to the fact that the latter was written in 2004, it misses a number of recent features of the language, including the aforementioned concept of *map expressions*, for which the reference compiler remained as our only source for definition of behaviour. This is the reason why our formalisation follows the behaviour of the Erlang/OTP compiler, rather than the language specification (mostly in cases of unspecified behaviour and evaluation order).

We have already written a paper [3] on the core semantics of Core Erlang, which discusses the related work on the formalisation of basic sequential language features in more detail. In this current paper, we focus on the extensions of our previous formalisation, including the definition of exceptions and other side effects. In the rest of this section, we overview the related work on these particular language features.

Like in other languages, exceptions in Core Erlang cause side effects (they implement non-structured control), and at the same time, they evaluate to special exception values. Exceptions need to be propagated in the control flow until a handler is found. This behaviour can be defined with natural (big-step) semantics in general, by checking the values of sub-expressions for exceptional values in the evaluation of compound expressions. This technique is taught in university courses [9], as well as it is employed in programming language semantics research [16, 28]. It is worth noting that there is another way to represent and propagate exceptions: rather than treating them as exceptional values, one can accumulate them as side effects in the program execution and implement exception handling based on the side effect list [14]. In our definition, we employ the former technique and explicitly evaluate expressions to exceptional values.

While formalising side effects, two different approaches were considered. The first is mentioned in the work of Horpácsi et. al. [14], where side effects are represented as traces. This approach, as mentioned before, handles exceptions and side effects alike. The other option is to interpret and model side effects faithfully with a number of semantics configuration cells, like in the C semantics by Ellison and Rosu [8] which uses over 60 cells. We concluded that for our current proofs, it is enough to formalise side effects on a logging level, and later on we may add more details.

There is also a considerable body of work on formalisations of other sequential languages, both functional, as is the case of CakeML [18], and imperative, as in CompCert [22], and indeed the trend to formalising programming language metatheory has been systematised in the POPLmark challenge [2].

3 The Core Semantics

This section briefly summarises our previous work [3] on defining natural semantics for Core Erlang. We repeat some of the underlying abstractions we have introduced in the previous work in order to ease the comprehension of this paper, whilst we also introduce some modifications that facilitate the extension of the semantics definition.

Throughout the following sections, the Coq code is frequently quoted in order to highlight the fact that this formalisation is machine-checked. Nevertheless, the inductive constructors of the transition relation in the operational semantics are presented in inference rule notation for better readability.

3.1 Abstract Syntax

In our previous paper we covered the basic sequential language features of Core Erlang, such as literals for simple and compound types, function abstraction and application, `let` and `letrec`. When defining the language in the Coq Proof Assistant, we apply deep embedding of the abstract syntax.

Figure 1 presents the context-free syntax of literals and patterns, while the syntax of the considered Core Erlang expressions can be seen in Figure 2. Currently, value lists [5] are only supported inside `let` expressions (to handle multiple simultaneous bindings), but `case` and `try` expressions can be formalised similarly in this regard.

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

Inductive Pattern : Type :=
| PVar (v : Var)
| PLit (l : Literal)
| PCons (hd tl : Pattern)
| PTuple (t : list Pattern)
| PNil.
```

Figure 1. Syntax of literals and patterns

Here we note that apart from some technical changes, this syntax definition is identical to the one presented in [3].

3.2 Values

While defining values (the normal form of expressions), we have related them to expressions in a similar way as the authors of the Erlang papers and Neuhäußer and Noll [10, 11, 24, 27]. Furthermore, in our approach function expressions are evaluated to *closures*, which capture the expressions' context and properly implements *EFun* as a binder. The details of this closure representation are described in our former work [3], yet this paper proposes some modifications to the idea which will be discussed in Section 7. Figure 3 shows the definition of expression values.

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

Inductive Expression : Type :=
| ENil
| ELit (l : Literal)
| EVar (v : Var)
| EFunId (f : FunctionIdentifier)
| EFun (vl : list Var) (e : Expression)
| ECons (hd tl : Expression)
| ETuple (l : list Expression)
| ECall (f : string) (l : list Expression)
| EApp (exp : Expression) (l : list Expression)
| ECase (e : Expression) (patts : list Pattern)
        (guards : list Expression) (bodies : list Expression)
| ELet (s : list Var) (el : list Expression) (e : Expression)
| ELetRec (fids : list FunctionIdentifier) (vls : list (list Var))
          (bodies : list Expression) (e : Expression)
| EMap (kl vl : list Expression).
```

Figure 2. Syntax of expressions

```
Definition FunctionExpression := (list Var) × Expression.

Inductive Value : Type :=
| VNil
| VLit (l : Literal)
| VClos (ref : Environment) (ext : list (FunctionIdentifier ×
    FunctionExpression)) (vl : list Var) (e : Expression)
| VCons (vhd vtl : Value)
| VTuple (vl : list Value)
| VMap (kl vl : list Value).
```

Figure 3. Semantic domain

The *ext* parameter of the closure constructor is called *environmental extension*, and is denoted by $\langle f_1 : fun(p_1) \rightarrow b_1, f_2 : fun(p_2) \rightarrow b_2, .., f_n : fun(p_n) \rightarrow b_n \rangle$ if only the elements $(f_1, (p_1, b_1)), (f_2, (p_2, b_2)), .., (f_n, (p_n, b_n))$ are contained in it. The use of closures will be discussed after introducing environments.

Note that this *Value* type is not strict enough, it includes elements that cannot be values of expressions. In particular, Core Erlang's *map values* cannot contain duplicate keys, and their elements are ordered based on their keys. In order to comply with this, we add additional restrictions on the elements of the *Value* type, by using the following helper functions:

- The *bValue_eq_dec* v_1 v_2 function decides the equality of v_1 and v_2. This helps us avoid key duplicates.
- *value_less* v_1 v_2 decides whether v_1 is less than v_2, yielding a Boolean value. There is a total order on Core Erlang values [1], with the following ordering between values of different types: *numbers* < *atoms* <

closures < *tuples* < *maps* < *lists*. It is important to note that the reference implementation, based on a numeric encoding of functions, defines an opaque ordering between function closures. In our formalisation, we adopt this idea by associating closures with a unique identifier and ordering closures based on the order defined for the numeric identifiers. Technically, this requires using an additional cell in the configuration of the big-step semantics (on both sides of ⇓) to numerate these identifiers, but for the sake of readability, we omit this detail in the paper.

With these helper functions, the *make_value_map* function creates an ordered value map from the lists of keys and values (the exact definition can be found in the formalisation [23]).

3.3 Environment

Expressions can contain free variables and function references, so we need an environment for the evaluation, which maps identifiers to values. The environment has to capture both variables and function identifiers, with the latter only being associated with function closures. Having this representation for environments, we can encode top-level functions as `letrec` expressions, and use a single union type for both local and global environments:

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

The environment will be denoted by Γ in the rest of the paper, while $\varnothing$ denotes the empty environment. If an environment contains only the $x_1 - v_1, x_2 - v_2, .., x_n - v_n$ bindings, the notation $\{x_1 : v_1, x_2 : v_2, \ldots, x_n : v_n\}$ will be used. To manage environments, several helper functions have been defined, most of these are described in our former work [3]; however, one has been significantly reworked in this paper:

$$\mathit{append\_funs\_to\_env}\ \mathit{fids}\ \mathit{paramlists}\ \mathit{bodies}\ \Gamma$$

This is used for `letrec` statements, adds function identifier (from *fids*) - closure bindings to Γ. The bound closures are constructed in the following way: the ith closure with the ith parameter list (from *paramlists*), ith body (from *bodies*) and Γ reference environment. Their environmental extensions (collection of function identifier - parameter list - body triples) are constructed from the same *fids*, *paramlists* and *bodies* parameters (this is similar to zipping three lists).

Examples. Let us demonstrate how closures are created and applied. Consider the following Core Erlang snippet:

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

While evaluating expressions, static binding should be applied; that is, the `X` in the function `Y` should evaluate to the value of the outer instance of `X`, not the one that is present in the scope of the application. In particular, the above expression should evaluate to 42.

To simulate this behaviour, we store the current environment in the closure value at the point of the function definition, and use it later on when evaluating the function body by applying the closure. In this particular example, the closure value would be $\mathit{VClos}\ \{X : 42\}\ \langle\rangle\ []\ X$.

However, in case of recursive functions, it is challenging to describe an inherently recursive environment in Coq, where all functions are required to always terminate. Let us consider a recursive function defined with a `letrec` expression:

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

The evaluation environment of the application of this function should be $\{'x'/0 : \mathit{VClos}\ \{'x'/0 : \mathit{VClos}\ \{'x'/0 : \ldots\}\ \langle\rangle\ []\ (\texttt{apply}\ 'x'/0())\}\ \langle\rangle\ []\ (\texttt{apply}\ 'x'/0())\}$ without using the environmental extension. This environment is endlessly recursive, which cannot be explicitly represented and defined as a function in Coq.

This problem was solved by the environmental extension. This construct stores the possibly recursive functions (aside the non-recursive current environment) defined at a time in form of function identifier - parameter list - body triples (pair of pairs in the implementation). These triples are used when applying a function to extend the body's evaluation environment (the stored environment) with closures constructed from the stored environmental extension. These closures use the originally stored environment and environmental extension, because all of these functions were defined at a time, so their contexts are the same. The *get_env* helper function is used for this process, it unfolds one level of the recursive environment with every recursive invocation. With this thought, the correct closure value for the previous recursive function will be the following: $\mathit{VClos}\ \varnothing\ \langle'x'/0 : \texttt{fun}() \rightarrow (\texttt{apply}\ 'x'/0())\rangle\ []\ (\texttt{apply}\ 'x'/0())$.

This problem was addressed by our previous paper [3], but the solution discussed there is not always applicable (see Section 7 for details), however, the fundamental thought was the same: the environment of the next evaluation step should be defined by the current step in the big-step semantics.

3.4 Core Dynamic Semantics

The biggest change between this update and our previous semantics is the omitted closure environment. Instead of closure environments, we use environmental extensions; apparently, the evaluation rules for `apply` and `letrec` expressions had to be adjusted to match this change. Another major change was the ordered evaluation of maps. Fortunately, the introduction of ordering on values and the *make_value_map* function in the map evaluation rule was sufficient to fulfil this goal.

In the following figures, the result *res* could be either a value or an exception, so its type is *Value* + *Exception*.

$$\frac{\langle \Gamma, e\rangle \Downarrow inl\ val' \quad \langle append\_vars\_to\_env\ [v]\ [val']\ \Gamma, e_1\rangle \Downarrow res}{\langle \Gamma, ETry\ e\ e_1\ e_2\ v\ vex_1\ vex_2\ vex_3\rangle \Downarrow res} \quad (\textsc{Try}^E)$$

$$\frac{\langle \Gamma, e\rangle \Downarrow inr\ (ex_1, ex_2, ex_3) \quad \langle append\_vars\_to\_env\ [vex_1; vex_2; vex_3]\ [exclass\_to\_value\ ex_1; ex_2; ex_3]\ \Gamma, e_2\rangle \Downarrow res}{\langle \Gamma, ETry\ e\ e_1\ e_2\ v\ vex_1\ vex_2\ vex_3\rangle \Downarrow res} \quad (\textsc{Catch}^E)$$

In the following rule *no_previous_match i* Γ *patts guards bodies v* states, the first *i* clause cannot be selected for the value *v* (either *v* does not match the pattern or the guard evaluation fails).

$$\frac{\langle \Gamma, e\rangle \Downarrow inl\ v \quad |patts| = |guards| \quad |patts| = |bodies| \quad no\_previous\_match\ |patts|\ \Gamma\ patts\ guards\ bodies\ v}{\langle \Gamma, ECase\ e\ patts\ guards\ bodies\rangle \Downarrow inr\ (if\_clause\ v)} \quad (\textsc{CaseExc}_1^E)$$

For the next rule, let us consider *nonclosure v* := ∀ Γ′, *ext*, *var_list*, *body*, *v* ≠ *VClos* Γ′ *ext var_list body*.

$$\frac{\langle \Gamma, exp\rangle \Downarrow inl\ v \quad eval\_all\ \Gamma\ params\ vals \quad nonclosure\ v}{\langle \Gamma, EApp\ exp\ params\rangle \Downarrow inr\ (badfun\ v)} \quad (\textsc{AppExc}_1^E)$$

$$\frac{eval\_all\ \Gamma\ params\ vals \quad |var\_list| \neq |vals| \quad \langle \Gamma, exp\rangle \Downarrow inl\ (VClos\ ref\ ext\ var\_list\ body)}{\langle \Gamma, EApp\ exp\ params\rangle \Downarrow inr\ (badarity\ v)} \quad (\textsc{AppExc}_2^E)$$

Figure 4. The big-step operational semantics of exception creation and *try* expressions

4 Exception Extension

In this section, the introduction of exceptions will be discussed. First, the syntax of expressions (Figure 2) is extended with an additional constructor for error handling statements:

$$|\ ETry\ (e\ e_1\ e_2 : Expression)\ (v\ vex_1\ vex_2\ vex_3 : Var)$$

This corresponds to the following concrete syntax:

$$\texttt{try}\ e\ \texttt{of}\ v \rightarrow e_1\ \texttt{catch}\ \langle vex_1, vex_2, vex_3\rangle \rightarrow e_2$$

In this form, `try` expressions handle one variable binding in their main clause, and three in the catch clause (exception class, reason, and additional data). This syntactic scheme (and the associated behaviour) is based on the language specification [5], which states that in Erlang exceptions are pairs of a reason term and an auxiliary data term. The latter contains additional information about the fault, as well as it encodes the class of the exception (*error*, *exit* or *throw*). In our formalisation, three variables are bound upon an exception caught, capturing separately the exception class (vex_1), the reason value (vex_2) and the additional information (vex_3).

In Coq, exceptions can be formalised as triplets of an exception class and two values, with the exception class being a value of a simple enumeration type. Also, exception classes can be converted back to a base values by using the *exclass_to_value* helper function.

```
Inductive ExceptionClass : Type := Error | Throw | Exit.
Definition Exception : Type :=
    ExceptionClass × Value × Value.
```

After introducing these abstractions, our former semantics [3] can be refined and extended to accommodate exceptional values and exception handling. In the refined semantics, the result of the evaluation of an expression is either a value or an exception. We denote the length of lists with the standard |.| notation for *length* and indexing operator (*list*[*i*]) for *nth* build-in functions in Coq for better readability.

Semantics of previously defined language features need to be tailored to handle exceptional values and propagate exceptions in a bottom-up manner. For this, we use auxiliary propositions to simplify the description of propagation. In particular, we introduce properties stating that a prefix of a list of expressions, or an entire expression list, can be evaluated to values (i.e. not raising any exceptions). In the following, (*eval_prefix* Γ *es vs i*) denotes ($i < |es|$) ⇒ ($|vs| = i$) ⇒ (∀$j < i$, ⟨Γ, *es*[*j*]⟩ ⇓ *inl vs*[*j*]). Similarly, (*eval_all* Γ *es vs*) denotes $|es| = |vs|$ ⇒ (∀$j < |es|$, ⟨Γ, *es*[*j*]⟩ ⇓ *inl vs*[*j*]).

Similarly to Erlang, we distinguish the common exception types and define shortcuts for creating instances of these:

- *badarith v*: arithmetic operation failed for the *v* value;
- *badarity v*: parameter count mismatch for *v*;
- *badfun v*: *v* is not a closure;
- *if_clause v*: no matching branch in a case expression.

Worth noting that Core Erlang specifies undefined behaviour if no clauses can be selected while evaluating a `case` expression; however, according to the behaviour of the Erlang/OTP compiler, an *if_clause* error is raised for such cases.

With these ideas, the semantics of exception creation and propagation rules are defined in Figures 4 and 5, respectively.

$$\frac{eval\_prefix\ \Gamma\ exps\ vals\ i \quad \langle \Gamma, exps[i] \rangle \Downarrow inr\ ex}{\langle \Gamma, ETuple\ exps \rangle \Downarrow inr\ ex}\ (\textsc{TupleExc}^E) \qquad \frac{\langle \Gamma, tl \rangle \Downarrow inr\ ex}{\langle \Gamma, ECons\ hd\ tl \rangle \Downarrow inr\ ex}\ (\textsc{ConsExc}_1^E)$$

$$\frac{\langle \Gamma, tl \rangle \Downarrow inl\ tlv \quad \langle \Gamma, hd \rangle \Downarrow inr\ ex}{\langle \Gamma, ECons\ hd\ tl \rangle \Downarrow inr\ ex}\ (\textsc{ConsExc}_2^E) \qquad \frac{eval\_prefix\ \Gamma\ params\ vals\ i \quad \langle \Gamma, params[i] \rangle \Downarrow inr\ ex}{\langle \Gamma, ECall\ fname\ params \rangle \Downarrow inr\ ex}\ (\textsc{CallExc}^E)$$

$$\frac{eval\_prefix\ \Gamma\ exps\ vals\ i \quad \langle \Gamma, exps[i] \rangle \Downarrow inr\ ex}{\langle \Gamma, ELet\ vars\ exps\ e \rangle \Downarrow inr\ ex}\ (\textsc{LetExc}^E)$$

$$\frac{|patts| = |guards| \quad |patts| = |bodies| \quad \langle \Gamma, e \rangle \Downarrow inr\ ex}{\langle \Gamma, ECase\ e\ patts\ guards\ bodies \rangle \Downarrow inr\ ex}\ (\textsc{CaseExc}_2^E) \qquad \frac{\langle \Gamma, exp \rangle \Downarrow inr\ ex}{\langle \Gamma, EApp\ exp\ params \rangle \Downarrow inr\ ex}\ (\textsc{AppExc}_3^E)$$

$$\frac{\langle \Gamma, exp \rangle \Downarrow inl\ v \quad eval\_prefix\ \Gamma\ params\ vals\ i \quad \langle \Gamma, params[i] \rangle \Downarrow inr\ ex}{\langle \Gamma, EApp\ exp\ params \rangle \Downarrow inr\ ex}\ (\textsc{AppExc}_4^E)$$

$$\frac{eval\_prefix\ \Gamma\ kl\ kvals\ i \quad eval\_prefix\ \Gamma\ vl\ vvals\ i \quad \langle \Gamma, kl[i] \rangle \Downarrow inr\ ex \quad |kl| = |vl|}{\langle \Gamma, EMap\ kl\ vl \rangle \Downarrow inr\ ex}\ (\textsc{MapExc}_1^E)$$

$$\frac{eval\_prefix\ \Gamma\ kl\ kvals\ i \quad eval\_prefix\ \Gamma\ vl\ vvals\ i \quad \langle \Gamma, kl[i] \rangle \Downarrow inl\ val \quad \langle \Gamma, vl[i] \rangle \Downarrow inr\ ex \quad |kl| = |vl|}{\langle \Gamma, EMap\ kl\ vl \rangle \Downarrow inr\ ex}\ (\textsc{MapExc}_2^E)$$

Figure 5. The big-step operational semantics of exception propagation

5 Side Effect Extension

In this section, the formalisation of side effects, such as standard input and output, is discussed. Unlike exceptions, other side effects are not captured in full detail in our semantics. We propose a hybrid approach where exceptions are faithfully modeled, whilst other side effects are only collected in an event log. This decision is based on the fact that the refactoring steps we aim to verify are not allowed to alter side effects, so the correctness in this respect is expressed as the order-preservation of effects.

We do the formalisation by further extending the semantics definition presented in the previous section; the adjustment of the already presented semantics rules is partially covered in this paper in Figure 6, but the rest of the rules can be constructed similarly — the entire formalisation is available on Github [23].

First, we need to introduce abstractions for representing and aggregating side effects. Currently, only simple reading and writing operations are supported, but this definition can be extended easily to support other kinds of effects. Each side effect is given a *SideEffectId* identifier, its type:

- Writing to standard output: *Output*;
- Reading from standard input: *Input*.

Thereafter, we introduce side effect logs in form of lists whose items couple side effect types with a list of values. The value list represents the parameters of the encoded side effect. Throughout this paper, side effect lists will be denoted by standard list notation (e.g. $[(Output, \dots), (Input, \dots)]$).

```
Definition SideEffectList : Type :=
  list (SideEffectId × list Value).
```

Next, we introduce a helper function to manage the concatenation of the first elements of a list containing side effect logs. We remind the reader that *app* (denoted by ++), *firstn* and *concat* are defined in Coq's standard library [6].

```
Definition concatn (def : SideEffectList)
    (l : list SideEffectList) (n : nat) : SideEffectList :=
  def ++ concat (firstn n l).
```

With the help of these abstractions, we can modify the *eval_all* and *eval_prefix* propositions, such that they additionally express how side effect lists are accumulated during subsequent evaluation of expressions. This ensures that the semantics defines leftmost-innermost evaluation order for expressions and their side effects, which is in line with the reference implementation (Neuhäußer and Noll [24]).

The definition of *eval_prefix* is shown in Figure 7 (note that *S i* denotes the successor of *i*). The refined version of *eval_all* can be obtained similarly from the side effect free variant. We provide a short description about these propositions:

1. eff_1 is used as an initial side effect log;
2. The evaluation of expressions contained in *exps* list all can have additional side effects which are stored in *eff* list of logs;
3. During the evaluation of the first expression, the initial side effect log is the eff_1 mentioned before, and the result will be the concatenation of this log with the first element of *eff*;
4. For the general *i*th case: the starting log of the evaluation is the concatenation of the initial eff_1 log and the first $(i-1)$ elements of *eff* whilst the result will append the *i*th element of *eff* to this log.

In the following rules, the result *res* could be either a value or an exception, so its type is *Value* + *Exception*.

$$\frac{}{\langle \Gamma, ELit\ l, eff_1\rangle \Downarrow \{inl\ (VLit\ l), eff_1\}} \quad (\textsc{Lit}^{SE}) \qquad \frac{}{\langle \Gamma, EVar\ s, eff_1\rangle \Downarrow \{\Gamma(inl\ s), eff_1\}} \quad (\textsc{Var}^{SE})$$

$$\frac{}{\langle \Gamma, EFunId\ fid, eff_1\rangle \Downarrow \{\Gamma(inr\ fid), eff_1\}} \quad (\textsc{FunId}^{SE}) \qquad \frac{}{\langle \Gamma, EFun\ vl\ e, eff_1\rangle \Downarrow \{inl\ (VClos\ \Gamma\ \langle\rangle\ vl\ e), eff_1\}} \quad (\textsc{Fun}^{SE})$$

$$\frac{eval\_all\ \Gamma\ params\ vals\ eff_1\ eff \qquad eval\ fname\ vals\ (concatn\ eff_1\ eff\ |params|) = (res, eff_2)}{\langle \Gamma, ECall\ fname\ params, eff_1\rangle \Downarrow \{res, eff_2\}} \quad (\textsc{Call}^{SE})$$

$$\frac{eval\_all\ \Gamma\ exps\ vals\ eff_1\ eff}{\langle \Gamma, ETuple\ exps, eff_1\rangle \Downarrow \{inl\ (VTuple\ vals), concatn\ eff_1\ eff\ |exps|\}} \quad (\textsc{Tuple}^{SE})$$

$$\frac{\begin{array}{c}\langle \Gamma, e, eff_1\rangle \Downarrow \{inl\ v, eff_1 +\!\!+ eff_2\} \qquad |patts| = |guards| \qquad |patts| = |bodies| \\ match\_clause\ v\ patts\ guards\ bodies\ i = Some\ (guard, exp, bindings) \\ \langle add\_bindings\ bindings\ \Gamma, eff_1 +\!\!+ eff_2, guard\rangle \Downarrow \{inl\ tt, eff_1 +\!\!+ eff_2\} \\ \langle add\_bindings\ bindings\ \Gamma, exp, eff_1 +\!\!+ eff_2\rangle \Downarrow \{res, eff_1 +\!\!+ eff_2 +\!\!+ eff_3\} \\ no\_previous\_match\ i\ \Gamma\ patts\ guards\ bodies\ v\ (eff_1 +\!\!+ eff_2)\end{array}}{\langle \Gamma, ECase\ e\ patts\ guards\ bodies, eff_1\rangle \Downarrow \{res, eff_1 +\!\!+ eff_2 +\!\!+ eff_3\}} \quad (\textsc{Case}^{SE})$$

For simplicity, log_1 will denote $concatn\ (eff_1 +\!\!+ eff_2)\ eff\ |params|$ and Γ' will denote $append\_vars\_to\_env\ var\_list\ vals\ (get\_env\ ref\ ext)$ in the following rule.

$$\frac{\begin{array}{c}\langle \Gamma, exp, eff_1\rangle \Downarrow \{inl\ (VClos\ ref\ ext\ var\_list\ body), eff_1 +\!\!+ eff_2\} \qquad |var\_list| = |vals| \\ eval\_all\ \Gamma\ params\ vals\ (eff_1 +\!\!+ eff_2)\ eff \qquad \langle \Gamma', body, log_1\rangle \Downarrow \{res, log_1 +\!\!+ eff_2\}\end{array}}{\langle \Gamma, EApp\ exp\ params, eff_1\rangle \Downarrow \{res, log_1 +\!\!+ eff_3\}} \quad (\textsc{Apply}^{SE})$$

For readability, log_2 will denote $concatn\ eff_1\ eff\ |exps|$ in the following rule.

$$\frac{eval\_all\ \Gamma\ exps\ vals\ eff_1\ eff \qquad \langle append\_vars\_to\_env\ vars\ vals\ \Gamma, e, log_2\rangle \Downarrow \{res, log_2 +\!\!+ eff_2\}}{\langle \Gamma, ELet\ vars\ exps\ e, eff_1\rangle \Downarrow \{res, log_2 +\!\!+ eff_2\}} \quad (\textsc{Let}^{SE})$$

$$\frac{|fids| = |bodies| \qquad |fids| = |parss| \qquad \langle append\_funs\_to\_env\ fids\ parss\ bodies\ \Gamma, e, eff_1\rangle \Downarrow \{res, eff_1 +\!\!+ eff_2\}}{\langle \Gamma, ELetRec\ fids\ parss\ bodies\ e, eff_1\rangle \Downarrow \{res, eff_1 +\!\!+ eff_2\}} \quad (\textsc{LetRec}^{SE})$$

$$\frac{eval\_map\_all\ \Gamma\ kl\ vl\ kvals\ vvals\ eff_1\ eff \qquad make\_value\_map\ kvals\ vvals = (kl', vl')}{\langle \Gamma, EMap\ kl\ vl, eff_1\rangle \Downarrow \{inl\ (VMap\ kl'\ vl'), concatn\ eff_1\ eff\ (|kl| * 2)\}} \quad (\textsc{Map}^{SE})$$

$$\frac{\langle \Gamma, tl, eff_1\rangle \Downarrow \{inl\ tlv, eff_1 +\!\!+ eff_2\} \qquad \langle \Gamma, hd, eff_1 +\!\!+ eff_2\rangle \Downarrow \{inr\ ex, eff_1 +\!\!+ eff_2 +\!\!+ eff_3\}}{\langle \Gamma, ECons\ hd\ tl, eff_1\rangle \Downarrow \{inr\ ex, eff_1 +\!\!+ eff_2 +\!\!+ eff_3\}} \quad (\textsc{ConsExc}_2^{SE})$$

Figure 6. The big-step operational semantics of a subset of Core Erlang expressions with side effects

Unfortunately, if leftmost innermost evaluation is used, maps evaluate in pairs of key and value expressions, so the property *eval_all* cannot be used anymore. A new attribute should be introduced, which uses a similar idea described above (the definition is presented in Figure 8).

Here, the collection of logs (*eff*) contains the caused side effects by keys and values too: the log in the $(2 * i)$th position is caused by the evaluation of the ith key expression while $(2 * i + 1)$th position contains the additional side effects of the evaluation of the ith value expression.

Obviously, the auxiliary *eval* function — which simulates built-in function calls — could result now a modified side effect log aside the value or exception of the represented call. In addition, the *no_previous_match* property was also extended with a *SideEffectList* parameter to evaluate guard expressions. Note, that according to the specification [5], guards cannot produce side effects, so during the evaluation of guards, the side effect log does not expand. With these modifications, we present the extended semantics of some key rules in Figure 6.

$$
\begin{aligned}
&\mathit{eval\_prefix}\ (\Gamma : \mathit{Environment})\ (\mathit{exps} : \mathit{list\ Expression})\\
&\quad (\mathit{vals} : \mathit{list\ Value})\ (\mathit{eff}_1 : \mathit{SideEffectList})\\
&\quad (\mathit{eff} : \mathit{list\ SideEffectList})\ (i : \mathit{nat}) :=\\
&\quad i < |\mathit{exps}| \Rightarrow |\mathit{vals}| = i \Rightarrow |\mathit{eff}| = i \Rightarrow\\
&\quad (\forall j < i, \langle \Gamma, \mathit{exps}[j], \mathit{concatn}\ \mathit{eff}_1\ \mathit{eff}\ j\rangle \Downarrow\\
&\quad\quad \{\mathit{inl}\ \mathit{vals}[j], \mathit{concatn}\ \mathit{eff}_1\ \mathit{eff}\ (S\ j)\})
\end{aligned}
$$

Figure 7. The definition of *eval_prefix* with side effects

$$
\begin{aligned}
&\mathit{eval\_map\_all}\ (\Gamma : \mathit{Environment})\ (\mathit{kl}, \mathit{vl} : \mathit{list\ Expression})\\
&\quad (\mathit{kvals}, \mathit{vvals} : \mathit{list\ Value})\ (\mathit{eff}_1 : \mathit{SideEffectList})\\
&\quad (\mathit{eff} : \mathit{list\ SideEffectList}) :=\\
&\quad |\mathit{kl}| = |\mathit{vl}| \Rightarrow |\mathit{kl}| = |\mathit{kvals}| \Rightarrow |\mathit{kl}| = |\mathit{vvals}| \Rightarrow\\
&\quad 2 * |\mathit{kl}| = |\mathit{eff}| \Rightarrow\\
&\quad (\forall i < |\mathit{kl}|, \langle \Gamma, \mathit{kl}[i], \mathit{concatn}\ \mathit{eff}_1\ \mathit{eff}\ i\rangle \Downarrow\\
&\quad\quad \{\mathit{inl}\ \mathit{kvals}[i], \mathit{concatn}\ \mathit{eff}_1\ \mathit{eff}\ (S\ i)\}) \Rightarrow\\
&\quad (\forall i < |\mathit{kl}|, \langle \Gamma, \mathit{vl}[i], \mathit{concatn}\ \mathit{eff}_1\ \mathit{eff}\ (S\ i)\rangle \Downarrow\\
&\quad\quad \{\mathit{inl}\ \mathit{vvals}[i], \mathit{concatn}\ \mathit{eff}_1\ \mathit{eff}\ (S\ (S\ i))\})
\end{aligned}
$$

Figure 8. Evaluation of expressions in maps

6 Application of the Semantics

In this section, first we present examples about the use of this semantics for formal program evaluation, followed by proofs about the properties of the semantics. Then we show some expression pattern equivalences in the new side effect semantics. Throughout this section, the definition of the auxiliary function *eval* will frequently be referred to, which implements calls to built-in functions. Due to space constraints, we omit the definition of this function, but it can be retrieved from the project's repository [23]. Moreover we introduce a notation for the addition inter-module call: $e_1 + e_2 := \mathit{ECall}\ \text{"plus"}\ [e_1, e_2]$.

6.1 Tests

The presented examples here also served as test cases: comparison between their formal evaluation result and the behaviour of the reference implementation has been carried out.

Even though we have formalised all the test cases in the extended semantics, for the sake of simplicity, we discuss the first examples (6.1-6.5) in the side effect free semantics. The simplified rules ($\textsc{Var}^E$, $\textsc{Lit}^E$, $\textsc{Fun}^E$, $\textsc{Funid}^E$, $\textsc{Tuple}^E$, $\textsc{Case}^E$, $\textsc{Call}^E$, $\textsc{App}^E$, $\textsc{Let}^E$ and $\textsc{LetRec}^E$) are not included in this paper, but they can be obtained from the side effect sensitive rules by omitting the irrelevant elements. For instance, we get $\textsc{Lit}^E$ from $\textsc{Lit}^{SE}$ by dropping the eff_1 variables and the containing configuration cell, and similarly, we get $\textsc{Call}^E$ from $\textsc{Call}^{SE}$ by omitting the side effect related parts and using the former version of *eval*:

$$\frac{\mathit{eval\_all}\ \Gamma\ \mathit{params}\ \mathit{vals} \qquad \mathit{eval}\ \mathit{fname}\ \mathit{vals} = \mathit{res}}{\langle \Gamma, \mathit{ECall}\ \mathit{fname}\ \mathit{params}\rangle \Downarrow \mathit{res}}\ (\textsc{Call}^E)$$

In general, the simplified rules can be obtained with carrying out the following steps:

1. Omit the side effect log cells in the semantics.
2. Replace *eval_all* with its side effect free version.
3. In the case of $\textsc{Case}^E$, use the *no_previous_match* property described in Figure 4.
4. In the case of $\textsc{Call}^E$, use the former version of the auxiliary function *eval*.

When presenting examples, we use concrete syntax and omit the *inl* and *inr* prefixes in the derivation trees for better readability. The first example shows the formal evaluation of the non-recursive example mentioned in Section 3.3.

Example 6.1 (Non-recursive application evaluation)**.** In this example the expression `let` Y `=` `fun()` $\rightarrow$ X `in let` X `=` `5 in apply` $Y()$ will be denoted by *exp*, and *cl* will be used to abbreviate the closure value $\mathit{VClos}\ \{X : 42\}\ \langle\rangle\ []\ X$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{X:5, Y:cl\}(Y) = cl}\ \textsc{Var}^E}{\langle\{X:5, Y:cl\}, Y\rangle \Downarrow cl} \qquad \dfrac{\dfrac{}{\{X:42\}(X) = 42}\ \textsc{Var}^E}{\langle\{X:42\}, X\rangle \Downarrow 42}}{\langle\{X:5, Y:cl\}, \texttt{apply}\ Y()\rangle \Downarrow 42}\ \textsc{App}^E}{\langle\{X:42, Y:cl\}, \texttt{let}\ X\ \texttt{=}\ 5\ \texttt{in apply}\ Y()\rangle \Downarrow 42}\ \textsc{Let}^E}{\langle\{X:42\}, \mathit{exp}\rangle \Downarrow 42}\ \textsc{Let}^E}{\langle\varnothing, \texttt{let}\ X = 42\ \texttt{in}\ \mathit{exp}\rangle \Downarrow 42}\ \textsc{Let}^E$$

Example 6.2 (Recursive application evaluation)**.** This example describes an endless recursion using the environmental extension. Similarly to the previous example, we introduce some notations to simplify the formalism:

$$
\begin{aligned}
\mathit{exp} &:= \texttt{letrec}\ \texttt{'}x\texttt{'/0} = \texttt{fun()} \rightarrow \texttt{apply}\ \texttt{'}x\texttt{'/0()}\ \texttt{in}\\
&\qquad \texttt{apply}\ \texttt{'}x\texttt{'/0()}\\
\mathit{cl} &:= \mathit{VClos}\ \varnothing\ \langle\texttt{'}x\texttt{'/0} : \mathit{fun}() \rightarrow \texttt{apply}\ \texttt{'}x\texttt{'/0()}\rangle\ []\\
&\qquad \texttt{apply}\ \texttt{'}x\texttt{'/0()}\\
\Gamma &:= \{\texttt{'}x\texttt{'/0} : \mathit{cl}\}
\end{aligned}
$$

Since the evaluation is divergent due to the infinitely recursive function, in the proof tree the subgoal *g* (subproof) is recurring and thus the proof search is divergent.

$$\dfrac{\dfrac{\dfrac{}{\langle\Gamma, \texttt{'}x\texttt{'/0}\rangle \Downarrow cl}\ \textsc{Funid}^E \qquad \dfrac{\dfrac{}{\langle\Gamma, \texttt{'}x\texttt{'/0}\rangle \Downarrow cl}\ \textsc{Funid}^E \qquad g}{g : \langle\Gamma, \texttt{apply}\ \texttt{'}x\texttt{'/0()}\rangle \Downarrow ??}\ \textsc{App}^E}{g : \langle\Gamma, \texttt{apply}\ \texttt{'}x\texttt{'/0()}\rangle \Downarrow ??}\ \textsc{App}^E}{\langle\varnothing, \mathit{exp}\rangle \Downarrow ??}\ \textsc{LetRec}^E$$

The following examples demonstrate the use of exception propagation semantics.

Example 6.3 (Exception occurs during calls)**.** This example shows how a faulty addition is evaluated to an exception. Also, to be able to reuse this example, it is generalised for any Γ environment.

$$\dfrac{\dfrac{}{\langle \Gamma, 5\rangle \Downarrow 5}\ \textsc{Lit}^E \qquad \dfrac{}{\langle \Gamma, \{\}\rangle \Downarrow \{\}}\ \textsc{Tuple}^E \qquad \begin{array}{c} \textit{eval def} \\ \textit{eval "plus"}\ [5,\{\}] = \textit{badarith}\ [5,\{\}] \end{array}}{\langle \Gamma, 5 + \{\}\rangle \Downarrow \textit{badarith}\ [5,\{\}]}\ \textsc{Call}^E$$

Example 6.4 (Application defining expression evaluates to an exception)**.** This example shows faulty evaluation of application, if its defining expression evaluated to an exception.

$$\dfrac{\dfrac{}{\langle \varnothing, 5 + \{\}\rangle \Downarrow \textit{badarith}\ [5,\{\}]}\ \text{Ex. 6.3}}{\langle \varnothing, \texttt{apply}\ (5 + \{\})(5,5)\rangle \Downarrow \textit{badarith}\ [5,\{\}]}\ \textsc{AppExc}_3^E$$

Example 6.5 (Application parameter mismatch)**.** In the case of this example, the applied function received more actual parameters than formal ones. We denote the function $\texttt{fun}() \rightarrow 4$ with *fun* and its closure value of *VClos* $\varnothing$ $\langle\rangle$ [] 4 with *cl*.

$$\dfrac{\begin{array}{c} \langle \varnothing, \texttt{fun}() \rightarrow 4\rangle \Downarrow cl \\ \langle \{X : cl\}, \texttt{apply}\ X(2)\rangle \Downarrow \textit{badarity}\ cl \end{array}}{\langle \varnothing, \texttt{let}\ X = \textit{fun}\ \texttt{in apply}\ X(2)\rangle \Downarrow \textit{badarity}\ cl}\ \textsc{LetExc}^E$$

The first statement can be proven by $\textsc{Fun}^E$ and the second one is detailed below:

$$\dfrac{\dfrac{\dfrac{}{\langle \{X : cl\}, X\rangle \Downarrow cl}\ \textsc{Var}^E, \textsc{Lit}^E \qquad \langle \{X : cl\}, 2\rangle \Downarrow 2 \qquad \dfrac{}{0 \neq 1}}{\langle \{X : cl\}, \texttt{apply}\ X(2)\rangle \Downarrow \textit{badarity}\ cl}\ \textsc{AppExc}_2^E$$

The following three examples explain the use of side effect semantics, along with the update evaluation of maps. First, we introduce some notations:

$$wr(se) : \textit{Expression} := \textit{ECall "fwrite"}\ [se]$$
$$out(sv) : \textit{SideEffectId} \times \textit{list Value} := (\textit{Output}, [sv])$$

First, a simple writing expression evaluation is formalised and proved to reuse it later.

Example 6.6 (Writing expression evaluation)**.** We assume here, that *se* evaluates to *sv* without producing any side effects. In addition, this example is generalised for any Γ environment and *log* side effect log.

$$\dfrac{\dfrac{}{\langle \Gamma, se, log\rangle \Downarrow \{sv, log\}}\ \textit{hypothesis} \qquad \begin{array}{c} \textit{eval def} \\ \textit{eval "fwrite"}\ [sv]\ log = ('ok', log +\!\!+ [out(sv)]) \end{array}}{\langle \Gamma, wr(se), log\rangle \Downarrow \{'ok', log +\!\!+ [out(sv)]\}}\ \textsc{Call}^{SE}$$

The next example shows the evaluation of applications, where potentially all steps create additional side effects.

Example 6.7 (Evaluation of applications with side effects)**.** It is important to note that we used an environment initially, where *Y* had been bound to *cl*, which is used to denote *VClos* $\varnothing$ $\langle\rangle$ [*Z*] $wr('c')$. In addition, log_2 denotes the $[out('a'), out('b')]$ side effect log while log_3 is used to abbreviate $[out('a'), out('b'), out('c')]$. The expression `let` X = $wr('a')$ `in` Y will be denoted by *exp*.

$$\dfrac{\begin{array}{c} \langle \{Y : cl\}, exp, []\rangle \Downarrow \{cl, [out('a')]\} \\ \langle \{Y : cl\}, wr('b'), [out('a')]\rangle \Downarrow \{'ok', log_2\} \\ \langle \{Z : 'ok'\}, wr('c'), log_2\rangle \Downarrow \{'ok', log_3\} \end{array}}{\langle \{Y : cl\}, \texttt{apply}\ exp(wr('b')), []\rangle \Downarrow \{'ok', log_3\}}\ \textsc{Apply}^{SE}$$

From the proof tree, the second and the third statements can be proved by the Example 6.6. Only the first statement is left to be discussed.

$$\dfrac{\dfrac{}{\langle \{Y : cl\}, wr('a'), []\rangle \Downarrow \{'ok', [out('a')]\}}\ \text{Ex. 6.6, } \textsc{Var}^{SE} \qquad \langle \{Y : cl\}, Y, [out('a')]\rangle \Downarrow \{cl, [out('a')]\}}{\langle \{Y : cl\}, exp, []\rangle \Downarrow \{cl, [out('a')]\}}\ \textsc{Let}^{SE}$$

The first statement can be proven with Example 6.6 and the variable evaluation with $\textsc{Var}^{SE}$.

Example 6.8 (Evaluation of maps with side effects)**.** The last test case shows the evaluation of maps. We denote an initial map expression $\sim\{wr('a') \Rightarrow wr('b'), wr('c') \Rightarrow 5\}\sim$ with *map*, and the caused side effects [*out*('a'), *out*('b'), *out*('c')] will be denoted with *log*. As mentioned before, the duplicate keys are replaced in the value map.

$$\dfrac{\begin{array}{c} \dfrac{}{\langle \Gamma, wr('a'), []\rangle \Downarrow \{'ok', [out('a')]\}}\ \textsc{Lit}^{SE}, \text{Ex. 6.6} \\ \langle \Gamma, wr('b'), [out('a')]\rangle \Downarrow \{'ok', [out('a'), out('b')]\} \\ \langle \Gamma, wr('c'), [out('a'), out('b')]\rangle \Downarrow \{'ok', log\} \\ \langle \varnothing, 5, log\rangle \Downarrow \{5, log\} \end{array}}{\langle \varnothing, map, []\rangle \Downarrow \{\sim\{'ok' \Rightarrow 5\}\sim, log\}}\ \textsc{Map}^{SE}$$

6.2 Proofs

In this section, we describe two properties of the semantics. The machine-checked proofs of these are available on Github [23], and only a short summary is provided here.

Theorem 6.9 (Extended commutativity of addition)**.**

$$\forall (v_1\ v_2 : \textit{Value}), (t : \textit{Value}), (\textit{eff}\ \textit{eff}_2 : \textit{SideEffectList}),$$
$$\textit{eval "plus"}\ [v_1; v_2]\ \textit{eff} = (\textit{inl}\ t, \textit{eff}) \Rightarrow$$
$$\textit{eval "plus"}\ [v_2; v_1]\ \textit{eff}_2 = (\textit{inl}\ t, \textit{eff}_2).$$

This theorem states that the addition is commutative using the auxiliary *eval* function, provided that it is applied on appropriate values (integer literals). Furthermore, it states that the built-in call to addition does not create any side effects (the logs are unmodified). Clearly, this theorem would not hold if the results were exceptions since the exceptional values would contain local information about the exception.

Theorem 6.10 (Determinism)**.**

$$\begin{aligned}&\forall(\Gamma : \mathit{Environment}), (e : \mathit{Expression}),\\ &\quad (v_1 : \mathit{Value} + \mathit{Exception}),\ (\mathit{eff}\ \mathit{eff}_1 : \mathit{SideEffectList}),\\ &\langle \Gamma, e, \mathit{eff}\rangle \Downarrow \{v_1, \mathit{eff} +\!\!+ \mathit{eff}_1\} \Rightarrow\\ &(\forall(v_2 : \mathit{Value} + \mathit{Exception}), (\mathit{eff}_2 : \mathit{SideEffectList}),\\ &\quad \langle \Gamma, e, \mathit{eff}\rangle \Downarrow \{v_2, \mathit{eff} +\!\!+ \mathit{eff}_2\} \Rightarrow v_1 = v_2 \wedge \mathit{eff}_1 = \mathit{eff}_2).\end{aligned}$$

Determinism is a very important attribute of the current formalisation. In theory, Core Erlang is not deterministic, however, the reference implementation also uses a leftmost-innermost evaluation strategy [24], and we followed the footsteps of the compiler. In our case, determinism states, that not only an expression can be evaluated to a single value (from an initial environment and side effect log), but also the created side effects are unique. We considered side effect logs as part of the environment, because some side effects could have effect on the evaluation.

6.3 Equivalences

In this section the equivalences shown in our previous work are discussed in the updated semantics. Exceptions and side effects make these somewhat more complex, but the proof ideas described in our former work [3] can be applied here. We define two types of equivalences:

- Weak equivalence: The evaluation result of the equivalent expressions is the same (either a value or an exception), but the side effects were different or were emitted in different order.
- Strong equivalence: The evaluation result of the equivalent expressions is the same and the same side effects have been caused in the same order.

In the first equivalence, we swap two expressions in two let bindings. It is important to note that after swapping the expressions, their side effects also swap in the result, thus this is considered a weak (conditional) equivalence. In addition, in this example, exceptional evaluation is not considered (the result is a *Value*), because the result exceptions are not necessarily the same after swapping the two expressions. The four assumptions capture that the evaluation of the expressions are independent of the X and Y variables and each other's side effects as well.

Equivalence 1 (Swapping variable expressions)**.** *If*

$$\langle \Gamma, e_1, \mathit{eff}_0\rangle \Downarrow \{\mathit{inl}\, v_1, \mathit{eff}_0 +\!\!+ \mathit{eff}_1\}$$
$$\langle \Gamma + \{X : v_2\}, e_1, \mathit{eff}_0 +\!\!+ \mathit{eff}_2\rangle \Downarrow \{\mathit{inl}\, v_1, \mathit{eff}_0 +\!\!+ \mathit{eff}_2 +\!\!+ \mathit{eff}_1\}$$
$$\langle \Gamma, e_2, \mathit{eff}_0\rangle \Downarrow \{\mathit{inl}\, v_2, \mathit{eff}_0 +\!\!+ \mathit{eff}_2\}$$
$$\langle \Gamma + \{X : v_1\}, e_2, \mathit{eff}_0 +\!\!+ \mathit{eff}_1\rangle \Downarrow \{\mathit{inl}\, v_2, \mathit{eff}_0 +\!\!+ \mathit{eff}_1 +\!\!+ \mathit{eff}_2\}$$

then

`let` X `=` e_1 `in let` Y `=` e_2 `in` X `+` Y

is equivalent to

`let` X `=` e_2 `in let` Y `=` e_1 `in` X `+` Y

With a similar chain of thought, we managed to formalise and prove another weak equivalence about swapping the variable binding order in a function application. However, currently we have not considered additional side effects in this conditional equivalence yet. Naturally, we also assumed, that the defining expression of the application evaluates to the same value regardless of the order of the elements in the environment (in the future we plan to prove this property as a generalised theorem). On the other hand, we only assumed here, that the parameters evaluate correctly (without causing exceptions), and the application can cause any errors, thus the result potentially can be an exception too.

Equivalence 2 (Swapping binding order in application)**.** *If*

$$\langle \Gamma, e_1, \mathit{eff}\rangle \Downarrow \{\mathit{inl}\, v_1, \mathit{eff}\}$$
$$\langle \Gamma + \{Y : v_2\}, e_1, \mathit{eff}\rangle \Downarrow \{\mathit{inl}\, v_1, \mathit{eff}\}$$
$$\langle \Gamma, e_2, \mathit{eff}\rangle \Downarrow \{\mathit{inl}\, v_2, \mathit{eff}\}$$
$$\langle \Gamma + \{X : v_1\}, e_2, \mathit{eff}\rangle \Downarrow \{\mathit{inl}\, v_2, \mathit{eff}\}$$
$$\langle \Gamma + \{X : v_1, Y : v_2\}, \mathit{exp}, \mathit{eff}\rangle \Downarrow \{v_0, \mathit{eff}\}$$
$$\langle \Gamma + \{Y : v_2, X : v_1\}, \mathit{exp}, \mathit{eff}\rangle \Downarrow \{v_0, \mathit{eff}\}$$

then

`let` X `=` e_1 `in let` Y `=` e_2 `in apply` exp`(`X`,` Y`)`

is equivalent to

`let` Y `=` e_2 `in let` X `=` e_1 `in apply` exp`(`X`,` Y`)`

The final equivalence is a strong one about expression extraction to a function. Here there is no need to assume anything; during the evaluation exceptions and side effects can be caused, but the result will remain the same after the extraction, vica versa.

Equivalence 3 (Extraction of an expression into a function)**.**

e

is equivalent to

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

We proved both directions in all of the mentioned examples using the same basic idea: first the given complex assumption has to be deconstructed that yields more information about the parts of it, then the deconstruction can be continued with these parts. We used determinism (Theorem 6.10) in some steps of this process, in order to fit the assumptions on the same expressions together. Thereafter, a proof tree can be built to prove the conclusion. The full machine-checked proofs are available on our Github repository [23] along with other quite similar equivalences and alternative proofs.

The proof of these expression pattern equivalences is an important result of our project, because these ones can be interpreted as simple local refactorings, and our ultimate goal is to argue about the behaviour-preservation of refactorings.

7 Discussion

In this section, we summarise the modifications and differences between the presented, the former and some other approaches. In terms of syntax, there are notable differences between our approach and related work (e.g. the Erlang formalisations [10, 11, 27]). In particular, we have removed the empty tuple and map literals, because they caused ambiguity and redundancy. They can be expressed using the empty list in the appropriate constructors. In the abovementioned sources the authors solved this problem by constraining the length of the mentioned lists (it should be over 0), but such constraints generate additional statements and cause undesired complexity in Coq proofs.

Moreover, we did not consider the ordering and equality of values in our former work [3]; however, these concepts were needed to formalise maps correctly, so it was necessary to introduce them in this semantics.

While formalising side effects we used an intermediate approach between modeling [8] or just logging everything [14], which models exceptions in a similar way as values during evaluation as mentioned before, but every other side effect is just logged. Compared to our former equivalence proofs, we had to introduce some extra assumptions in some cases. This is due to the fact that side effects are not interpreted, only logged, and there could be side effects that alter the evaluation of some expressions (i.e. the side effect log can be interpreted as an evaluation context too).

Closures of recursive functions

In this updated semantics, we have dropped the closure environment introduced in our previous work [3], because that concept is not always correct. For example, we can define a recursive function which takes three iterations to terminate:

$body_1$:
```
letrec 'f'/1 = fun(X) ->
  case X of
    <0> when 'true' -> 5
    <1> when 'true' -> apply 'f'/1(0)
    <A> when 'true' -> apply 'f'/1(1)
  end in
```
let_X:
```
  let X = fun(F) ->
```
$body_X$:
```
    letrec 'f'/1 = fun(X) -> 0
      in apply F(2)
  in
  apply X('f'/1)
```

In the body of `let`, the binding of `'f'/1` is overwritten locally, however, this action replaces the existing binding in the closure environment. We present the formal evaluation of this example in our former approach [3] in Figure 9. In order to enhance readability, we use the following notations:

- Beside the notations on the code snippet, *exp* will denote the whole `letrec` expression;

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{|\Gamma_F + \{X:1\}, \{'f'/1 : \Gamma_F\}, 0| \Downarrow 5\ \lightning}{|\Gamma_F + \{X:2, A:2\}, \{'f'/1 : \{\Gamma_F\}\}, \texttt{apply}\ 'f'/1(1)| \Downarrow 5}\,3.9}{|\Gamma_F + \{X:2\}, \{'f'/1 : \Gamma_F\}, body_1| \Downarrow 5}\,3.7}{|\Gamma_F, \{'f'/1 : \Gamma_F\}, \texttt{apply}\ F(2)| \Downarrow 5}\,3.9}{|\Gamma + \{F : cl_1\}, \{'f'/1 : \Gamma\}, body_X| \Downarrow 5}\,3.11}{|\Gamma + \{X : cl_X\}, \{'f'/1 : \Gamma\}, \texttt{apply}\ X('f'/1)| \Downarrow 5}\,3.9}{|\Gamma, \{'f'/1 : \Gamma\}, let_X| \Downarrow 5}\,3.10}{|\varnothing, \varnothing, exp| \Downarrow 5}\,3.11$$

Figure 9. Closures as parameters using closure environments

- cl_1 will denote the closure of the first recursive function: $VClos\ (inr\ 'f'/1)\ [X]\ body_1$
- cl_2 will denote the closure of the second recursive function: $VClos\ (inr\ 'f'/1)\ [X]\ 0$
- cl_X will denote the closure of the function bound to `X`: $VClos\ (inl\ \{'f'/1 : cl_1\})\ [F]\ body_X$
  At the point of application of Rule 3.10, it is omitted, that `fun()` $\rightarrow$ $body_X$ evaluates to this closure. This function is not recursive, so it will not be added to the closure environment.
- Γ will denote: $\{'f'/1 : cl_1\}$;
- Γ_F will denote $\{'f'/1 : cl_2, F : cl_1\}$;
- The operator + will be used to denote the addition of some bindings to some environment.

Now, we present the evaluation in the updated semantics. For simplicity, we use the side effect free semantics (see Figure 10). First, we modify the previous notations slightly:

- $cl_1 := VClos\ \varnothing\ \langle 'f'/1 : fun(X) \rightarrow body_1 \rangle\ [X]\ body_1$
- $cl_2 := VClos\ \{'f'/1 : cl_1, F : cl_1\}\ \langle 'f'/1 : fun(X) \rightarrow 0 \rangle\ [X]\ 0$. Apparently — because of the environmental extension, and the *insert_value*'s replacing behaviour — if this closure had been applied, then its body would have been evaluated inside the environment where $'f'/1$ is bound to cl_2.
- cl_X will denote the closure of the function bound to `X`: $VClos\ \{'f'/1 : cl_1\}\ \langle\rangle\ [F]\ body_X$. At the point of application of Let^E, it is omitted that `fun()` $\rightarrow$ $body_X$ evaluates to this closure. In this case, because this function is not recursive, no environmental extension is needed.

In the updated closure representation and semantics, the biggest changes have been made in App^E and LetRec^E. In App^E, the evaluation environment of the body had to be constructed from the local environment, the actual and formal parameter bindings, and the addition of function closures stored in the environmental extension. In LetRec^E the construction of the environmental extension had to be introduced for the defined recursive function closures.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle\{'f'/1 : cl_1, X : 0\}, 5\rangle \Downarrow 5}\,\textsc{Lit}^E}{\langle\{'f'/1 : cl_1, X : 0\}, body_1\rangle \Downarrow 5}\,\textsc{Case}^E}{\langle\{'f'/1 : cl_1, X : 1\}, \texttt{apply}\ 'f'/1(0)\rangle \Downarrow 5}\,\textsc{App}^E}{\langle\{'f'/1 : cl_1, X : 1\}, body_1\rangle \Downarrow 5}\,\textsc{Case}^E}{\langle\{'f'/1 : cl_1, X : 2, A : 2\}, \texttt{apply}\ 'f'/1(1)\rangle \Downarrow 5}\,\textsc{App}^E}{\langle\{'f'/1 : cl_1, X : 2\}, body_1\rangle \Downarrow 5}\,\textsc{Case}^E}{\langle\{'f'/1 : cl_2, F : cl_1\}, \texttt{apply}\ F(2)\rangle \Downarrow 5}\,\textsc{App}^E}{\langle\{'f'/1 : cl_1, F : cl_1\}, body_X\rangle \Downarrow 5}\,\textsc{LetRec}^E}{\langle\{'f'/1 : cl_1, X : cl_X\}, \texttt{apply}\ X('f'/1)\rangle \Downarrow 5}\,\textsc{App}^E}{\langle\{'f'/1 : cl_1\}, let_x\rangle \Downarrow 5}\,\textsc{Let}^E}{\langle\varnothing, exp\rangle \Downarrow 5}\,\textsc{LetRec}^E$$

Figure 10. Closures as parameters using environmental extension

8 Conclusion and Future Work

In this paper, we briefly explained why having a formal definition is important for a programming language, and presented a formalisation of a subset of sequential Core Erlang that can be used to argue about behaviour preservation of local refactorings on Core Erlang programs. Core Erlang is just a stepping stone to our ultimate goal which is arguing about refactorings on Erlang programs. This language is suitable for us, because it is not merely a subset of Erlang, but also Erlang (along with other languages, like Elixir [12]) translates to Core Erlang during compilation. Thereafter, we briefly discussed the related work and our previous results [3].

Thereafter, we extended this semantics with the concepts of exceptions and side effects. After finishing this extension, we updated the former examples, proofs and expression pattern equivalences and added new ones. These equivalences can be interpreted as simple local refactorings. Some of these fully preserve the behaviour (strong equivalences) while others evaluate to the same the results. All of our work has been formalised in the Coq Proof Assistant and can be accessed on Github [23]. Next, we compared the semantics to our former approach. In the future we plan to implement the advancement to Erlang, the addition of other expressions (e.g. binaries, bitstrings) and the simplification of current proofs with the help of Coq's tactic language.

Evaluation. The work has shown that the semantics is suitable for formalising proofs of various properties of Core Erlang, including reasoning about expression equivalence, which will support proofs of correctness for program refactorings. We have formalised a representative subset of sequential Core Erlang, but others (e.g. bitstrings, binaries) could be formalised using similar techniques to those used here.

Formal reasoning is a rigorous discipline, as we remarked in [3]; with the introduction of exceptions and side effects this has become no less true. On the other hand, additional tactics can be developed to reduce the size of such complex machine-checked proofs, and indeed include some level of automation.

Future Work. As noted before, there are various ways to enhance our formalisation. Our short term goals include:

- Proving the correctness of additional local refactorings (e.g. renaming variables, functions, expression extraction to top-level function, and so on).
- Shortening the proofs by means of custom tactics (e.g. unfolding tactics based on the length of the list in question).
- Extending the coverage of side effects and their semantics with more type of effects (e.g. global variable modifications);
- Simplifying some expressions (`case`, `let`, `letrec` and `map`) which contain several lists of the same length to one list of tuples;
- Investigation of delayed writing side effects during list evaluation: if list expressions contain nested write expressions (for example the call for the write function is inside a `let` expression) then the order of the output values is from front to back, however, according to the tests about evaluating lists to exceptions or read expressions, list evaluation order is from back to front;
- Formalising the module system and inter-module calls.

Our longer-term goals include formalising divergence, extending the work to Erlang itself (semantics and syntax), formalising primitive operations and inter-module calls, and formalising the concurrent semantics of Core Erlang.

When formalising concurrent parts of a language, big-step semantics is usually not expressive enough. To extend to concurrency we would expect to devise a small-step semantics that us compatible with the big-step one, so that results proved for the big-step version would carry over to the small step case.

Acknowledgments

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] Joe Armstrong. 2013. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf.

[2] Brian E. Aydemir et al. 2005. Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005 (LNCS, Vol. 3603).* Springer. https://doi.org/10.1007/11541868_4

[3] Péter Bereczky, Dániel Horpácsi, and Simon Thompson. 2020. A Proof Assistant Based Formalisation of Core Erlang. (2020). arXiv:2005.11821

[4] Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. 2012. A first step in the design of a formally verified constraint-based testing tool: FocalTest. In *International Conference on Tests and Proofs.* Springer, 35–50. https://doi.org/10.1007/978-3-642-30473-6_5

[5] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. 2004. *Core Erlang 1.0.3 language specification.* Technical Report. https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf

[6] Coq documentation 2020. *The Coq Proof Assistant Documentation.* Retrieved July 2st, 2020 from https://coq.inria.fr/documentation

[7] Emanuele De Angelis, Fabio Fioravanti, Adrián Palacios, Alberto Pettorossi, and Maurizio Proietti. 2018. Bounded Symbolic Execution for Runtime Error Detection of Erlang Programs. (2018). arXiv:1809.04770

[8] Chucky Ellison and Grigore Rosu. 2012. An Executable Formal Semantics of C with Applications. *ACM SIGPLAN Notices* 47, 1 (2012), 533–544. https://doi.org/10.1145/2103621.2103719

[9] Exceptions 2015. *Course Notes: Formalising exceptions.* Retrieved May 4th, 2020 from https://www.scss.tcd.ie/Matthew.Hennessy/splexternal2015/LectureNotes/Exceptionscopy.pdf

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

[11] Lars-Åke Fredlund, Dilian Gurov, Thomas Noll, Mads Dam, Thomas Arts, and Gennady Chugunov. 2003. A verification tool for Erlang. *International Journal on Software Tools for Technology Transfer* 4, 4 (2003), 405–420. https://doi.org/10.1007/s100090100071

[12] Kofi Gumbs. 2017. *The Core of Erlang.* Retrieved 11th May, 2020 from https://8thlight.com/blog/kofi-gumbs/2017/05/02/core-erlang.html

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

[14] Dániel Horpácsi, Judit Kőszegi, and Zoltán Horváth. 2017. Trustworthy Refactoring via Decomposition and Schemes: A Complex Case Study. (2017). arXiv:1708.07225

[15] Dániel Horpácsi, Judit Kőszegi, and Simon Thompson. 2016. Towards Trustworthy Refactoring in Erlang. (2016). arXiv:1607.02228

[16] Ruud Koot and Jurriaan Hage. 2015. Type-Based Exception Analysis for Non-Strict Higher-Order Functional Languages with Imprecise Exception Semantics. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation* (Mumbai, India) *(PEPM '15).* Association for Computing Machinery, New York, NY, USA, 127–138. https://doi.org/10.1145/2678015.2682542

[17] Judit Kőszegi. 2018. KErl: Executable semantics for Erlang. *CEUR Workshop Proceedings* 2046 (2018), 144–160. http://ceur-ws.org/Vol-2046/koszegi.pdf

[18] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL).* ACM Press. https://doi.org/10.1145/2535838.2535841

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

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

[21] Ivan Lanese, Davide Sangiorgi, and Gianluigi Zavattaro. 2019. Playing with Bisimulation in Erlang. In *Models, Languages, and Tools for Concurrent and Distributed Programming*, Michele Boreale, Flavio Corradini, Michele Loreti, and Rosario Pugliese (Eds.). Springer, Cham, 71–91. https://doi.org/10.1007/978-3-030-21485-2_6

[22] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52 (2009). Issue 7. https://doi.org/10.1145/1538788.1538814

[23] Natural Semantics for Core Erlang 2020. *Core Erlang Formalization.* Retrieved July 5th, 2020 from https://github.com/harp-project/Core-Erlang-Formalization

[24] Martin Neuhäußer and Thomas Noll. 2007. Abstraction and model checking of Core Erlang programs in Maude. *Electronic Notes in Theoretical Computer Science* 176, 4 (2007), 147–163. https://doi.org/10.1016/j.entcs.2007.06.013 Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006).

[25] Naoki Nishida, Adrián Palacios, and Germán Vidal. 2017. A reversible semantics for Erlang. In *International Symposium on Logic-Based Program Synthesis and Transformation*, Manuel V Hermenegildo and Pedro Lopez-Garcia (Eds.). Springer, Springer International Publishing, Cham, 259–274. https://doi.org/10.1007/978-3-319-63139-4_15

[26] John C Reynolds. 1998. Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation* 11, 4 (1998), 363–397. https://doi.org/10.1023/A:1010027404223

[27] Germán Vidal. 2015. Towards Symbolic Execution in Erlang. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, Andrei Voronkov and Irina Virbitskaite (Eds.). Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 351–360. https://doi.org/10.1007/978-3-662-46823-4_28

[28] Joel J Wright. 2005. *Compiling and reasoning about exceptions and interrupts.* Ph.D. Dissertation. University of Nottingham.