A Frame Stack Semantics for Sequential Core Erlang

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

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

ABSTRACT

We present a small-step, frame stack style, semantics for sequential Core Erlang, a dynamically typed, impure functional programming language. The semantics and the properties that we prove are machine-checked with the Coq proof assistant. We improve on previous work by including exceptions and exception handling, as well as built-in data types and functions. Based on the semantics, we define multiple concepts of program equivalence (contextual, CIU equivalence, and equivalence based on logical relations) and prove that the definitions all coincide. Using this we are able to give a correctness criterion for refactorings, which is one of the main motivations of this work, by means of contextually equivalent symbolic expression pairs.

CCS CONCEPTS

• **Theory of computation → Operational semantics**; **Program reasoning**; **Functional constructs**.

KEYWORDS

Formal semantics, Frame stack semantics, Coq, program equivalence, Erlang, CIU theorem

ACM Reference Format:
Péter Bereczky, Dániel Horpácsi, and Simon Thompson. 2023. A Frame Stack Semantics for Sequential Core Erlang. In *The 35th Symposium on Implementation and Application of Functional Languages (IFL 2023), August 29–31, 2023, Braga, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3652561.3652566

1 INTRODUCTION

Most language processors and refactoring tools lack a precise formal specification of how the code is affected by the changes they may make. In particular, using a refactoring tool to improve the quality of the code should not change the observable behaviour of the program; however, this property is validated only by testing in most cases. Higher assurance can be achieved by making formal arguments to verify behaviour preservation, which requires a formal description (e.g., a formal semantics) of the programs being refactored, a precise specification how the refactorings affect these programs, and a suitable definition of program equivalence.

The research presented in this paper is part of a wider project aiming to improve the trustworthiness of Erlang [10] refactorings via formal verification [17]. As a stepping stone, we formalise Core Erlang [9], which is a core and intermediate language of Erlang and its compilation process. Besides Erlang, other languages (e.g., Elixir [15]) can be translated to Core Erlang, therefore a formalisation of the core language may contribute to the studies of all of these languages.

In this paper, we are defining a small-step (frame stack style) semantics of Core Erlang extending our previous work [18] with most of the sequential features of Core Erlang. Based on the formal semantics, we define a number of expression equivalence concepts, which we use to prove the correctness of simple Erlang refactorings. All of the results presented here are also formalised with the Coq proof management system [32].

```
f(x) when length(x) == 0 -> e1;
f(_) -> e2.
```

$$\downarrow \underline{when}\ x \notin vars(e_1)$$

```
f([]) -> e1;
f(_) -> e2.
```

Figure 1: A simple function refactoring in Erlang

Running example. We present a simple refactoring in Erlang, which replaces a guard of a function clause with a more effective and readable pattern matching (Figure 1). Note that in the figure, f, e_1, e_2, and x are metavariables (f is used for atoms, e_1, e_2 for expressions, x for variables) and the side-condition of the refactoring is given as a logical constraint in the *when* clause. This example will serve as a running example throughout this paper.

To utilize the formal semantics of Core Erlang presented here, first both of these code chunks are translated to Core Erlang by the standard Erlang/OTP compiler (OTP version 24). Next, we encode the Core Erlang programs in the Coq formalisation, and prove their equivalence. In this process, we consider the compiler as trusted. This is a reasonable assumption for two reasons. The compiler produces human-readable Core Erlang, and so it can be inspected in any particular case. Also, the compiler is widely used, and so has been subject to substantial social scrutiny. In Figure 2 we show the result of the unoptimised translation of the first function in Figure 1, after clearing the annotations of Core Erlang [9].

Contributions. In this paper, we make the following contributions:

IFL 2023, August 29–31, 2023, Braga, Portugal

ACM ISBN 979-8-4007-1631-7/23/08
https://doi.org/10.1145/3652561.3652566

```
f/1 = fun (_0) ->  case _0 of
    <x> when try let <_1> = call 'erlang':'length'(x)
                    in call 'erlang':'=='(_1, 0)
              of <Try> -> Try
              catch <T,R> -> 'false'
       -> e1
    <_3> when 'true' -> e2
    <_2> when 'true' ->
      primop 'match_fail'({'function_clause',_2})
  end
```

Figure 2: Core Erlang code of the first function in Figure 1

- A (frame stack) semantics for the sequential subset of Core Erlang including exception handling, which extends and improves on our previous work [18].
- Definitions of termination-based program equivalence relations (namely: contextual equivalence, equivalence based on logical relations, and CIU equivalence).
- Properties of the program equivalence relations, and their coincidence.
- A number of simple (Erlang) expression equivalences (one presented as the running example above).
- A machine-checked implementation of the results in the Coq proof management system [32].

For the proofs of the theorems, lemmas, and examples presented here, we refer to the formalisation [32]. The rest of the paper is structured as follows. In Section 2 we summarise the concepts and our previous work which this paper builds on. Thereafter, Section 3 introduces the formal semantics of sequential Core Erlang. Section 4 discusses program equivalence definitions, followed by a short discussion in Section 5 on the Coq implementation details. Section 6 briefly discusses related work, while Section 7 concludes and points out areas for future work.

2 BACKGROUND

In this section, we briefly introduce the concepts of frame stack semantics and program equivalences, and also discuss our previous work which we evolve in this paper.

Frame stack semantics. A frame stack style semantics is a small-step [30] operational semantics. It is derived from the reduction-style semantics of Wright and Felleisen [37]. In the frame stack semantics, the reduction rules are applied in a special reduction context; it is constructed as a stack of basic evaluation frames. This stack can also be considered as the continuation of the evaluation. The advantage of this style of semantics is that it is simpler to use in a proof assistant since the frame stack is separated into a distinct configuration cell (hence it does not need to be inferred like the reduction context in reduction-style semantics).

Program equivalence concepts. In this paper, we investigate three definitions of program equivalence. *Contextual equivalence* is a syntactical notion of program equivalence: two expressions are equivalent if their behaviours cannot be distinguished in *any syntactical contexts*. Usually, it is burdensome to reason about contextual equivalence since it requires induction on the context; however, this notion is needed to express the correctness of local program transformations (i.e., equivalent programs can be replaced in arbitrary context without affecting the overall behaviour).

With *equivalence based on logical relations* [28] two expressions are equivalent when their behaviour cannot be distinguished in *equivalent reduction contexts* (i.e., frame stacks). In the frame stack semantics reasoning about this equivalence can be carried out by inspecting the semantics of the expressions instead of using induction on the contexts.

With *CIU equivalence* ("closed instances of use" equivalence) [22] two expressions are equivalent when their behaviour is indistinguishable in *any reduction context*. This notion is the most suitable to reason about expressions being equivalent, since it involves only one reduction context (frame stack).

Previous work. In earlier work [18] we have defined frame stack semantics for a limited variant of Core Erlang, and defined the program equivalence concepts mentioned above. Hereby we extend this limited language by adding further language elements of Core Erlang: function closures, tuples, maps, sequencing, value lists and value sequences, exceptions and exception handling. Moreover, we generalise pattern matching, `let`, `letrec` expressions, and built-in function calls. With this extension, our semantics covers most of the (sequential) language elements of Core Erlang [9] (except for the module system, bitstring and alias patterns, binaries, timeouts and floats). The syntax presented here also covers all the concurrent language elements of Core Erlang (apart from process identifiers), since they are expressed by built-in functions and primitive operations since OTP version 23 [16]. After extending the language and its semantics, we adjust and extend the equivalence relations, corresponding properties and proofs. Another improvement on previous work is handling expressions with an arbitrary number of subexpressions in a uniform way.

It is worth mentioning that in prior work we also investigated a big-step style semantics for sequential Core Erlang [4, 5], which included studying the semantics of various language elements presented here, allowing us to reuse and adjust some of the results about the syntax achieved there.

3 CORE ERLANG SEMANTICS

In this section, we discuss the syntax and frame stack semantics of sequential Core Erlang, evaluate an example expression, and show a number of semantic properties. For proofs we refer to the machine-checked Coq formalisation [32].

3.1 Syntax

First, we present the syntax of Core Erlang [9] in Figure 3. We use subscripts to denote multiplicities and superscripts to denote the roles of values and expressions. For simplicity, we denote lists from the metatheory with $e_1, \ldots, e_n$, and non-empty lists with $e_1, e_2, \ldots, e_n$. We use x to range over variables, i over integers, a, f denote atoms, and k, l, m, n are used to denote natural numbers. Compared to our previous work [18], here we separate values from expressions, but use similar notations for them (e.g., $\{e_1, \ldots, e_n\}$ is a tuple expression, while $\{v_1, \ldots, v_n\}$ is a tuple value).

The patterns of the language are integers (denoted with numbers), atoms (enclosed in single quotation marks), variables, lists,

$$
\begin{aligned}
p &\in \mathit{Pattern} ::= i \mid a \mid x \mid [p_1|p_2] \mid [] \mid \{p_1, \ldots, p_n\} \\
&\quad \mid \sim\{p_1^k \Rightarrow p_1^v, \ldots, p_n^k \Rightarrow p_n^v\}\sim \\
ps &\in \mathit{list(Pattern)} ::= <p_1, \ldots, p_n> \\
cli &\in \mathit{ClosItem} ::= f/k = \texttt{fun}(x_1, \ldots, x_k) \rightarrow e \\
ext &\in \mathit{list(ClosItem)} ::= cli_1, \ldots, cli_n \\
cl &\in \mathit{Clause} ::= ps \ \texttt{when}\ e^g \rightarrow e^b \\
v &\in \mathit{Val} ::= i \mid a \mid x \mid f/k \mid \mathit{clos}(ext, [x_1, \ldots, x_n], e) \\
&\quad \mid [v_1|v_2] \mid [] \mid \{v_1, \ldots, v_n\} \mid \sim\{v_1^k \Rightarrow v_1^v, \ldots, v_n^k \Rightarrow v_n^v\}\sim \\
nv &\in \mathit{NonVal} ::= \texttt{fun}(x_1, \ldots, x_n) \rightarrow e \mid <e_1, \ldots, e_n> \mid [e_1|e_2] \\
&\quad \mid \{e_1, \ldots, e_n\} \mid \sim\{e_1^k \Rightarrow e_1^v, \ldots, e_n^k \Rightarrow e_n^v\}\sim \\
&\quad \mid \texttt{call}\ e^m{:}e^f(e_1, \ldots, e_n) \mid \texttt{primop}\ a(e_1, \ldots, e_n) \\
&\quad \mid \texttt{apply}\ e(e_1, \ldots, e_n) \mid \texttt{case}\ e_1\ \texttt{of}\ cl_1; \ldots; cl_n\ \texttt{end} \\
&\quad \mid \texttt{let} <x_1, \ldots, x_n> = e_1\ \texttt{in}\ e_2 \mid \texttt{do}\ e_1\ e_2 \mid \texttt{letrec}\ ext\ \texttt{in}\ e \\
&\quad \mid \texttt{try}\ e_1\ \texttt{of} <x_1, \ldots, x_k> \rightarrow e_2\ \texttt{catch} <x_{k+1}, \ldots, x_{k+n}> \rightarrow e_3 \\
e &\in \mathit{Exp} ::= nv \mid v \\
vs &\in \mathit{ValSeq} ::= <v_1, \ldots, v_n> \\
c &\in \mathit{ExcClass} ::= \texttt{'throw'} \mid \texttt{'exit'} \mid \texttt{'error'} \\
exc &\in \mathit{Exception} := \{c, v^r, v^d\}^X \\
res &\in \mathit{Result} := exc \mid vs
\end{aligned}
$$

Figure 3: Syntax of Core Erlang

tuples and maps (tilde-enclosed tuples containing key-value pairs denoted with superscripts k and v). The set of values in the language essentially consists of the same constructs, extended with function identifiers (f/k, atom-arity pairs) and function closures. Note that we refer to variables and function identifiers as *names* in the rest of the paper.

Closures. Closures are the normal forms of functions (we reuse and adjust their definition from [5]). Besides a function's parameter list, body expression, they also include a list of function definitions (*ext*) that can be applied recursively by the body expression (this list is constructed while evaluating a `letrec` expression).

The expressions of the language are either values or non-values, which consist of uncurried function abstractions, value lists (denoted with $<e_1, \ldots, e_n>$, usually used in binding expressions), lists, tuples, maps, inter-module calls (we denote the module and function names with superscripts), primitive operations, function application, binding expressions (`let`, `letrec`, `case`, `try`), and sequencing (`do`).

Primitive operations, inter-module calls, and function application. The semantics of primitive operations (`primop`) is unspecified, thus in the formalisation we simulate the behaviour of these operations based on their implementation in the standard compiler [1]. Inter-module calls (`call`) are applications of top-level functions of a (Core) Erlang module (identified by module and function names). The current formalisation does not include the module system yet, thus for our semantics inter-module call expressions are mainly

$$
\begin{aligned}
r &\in \mathit{Redex} ::= vs \mid exc \mid e \mid \square \\
id &\in \mathit{FrameId} ::= tuple \mid values \mid call(v^m, v^f) \mid primop(a) \mid map \\
&\quad \mid app(v) \\
F &\in \mathit{Frame} ::= id(v_1, \ldots, v_{i-1}, \square, e_{i+1} \ldots, e_n) \mid [e_1|\square] \mid [\square|v_2] \\
&\quad \mid \texttt{call}\ \square{:}e^f(e_1, \ldots, e_n) \mid \texttt{call}\ v^m{:}\square(e_1, \ldots, e_n) \\
&\quad \mid \texttt{apply}\ \square(e_1, \ldots, e_n) \mid \texttt{case}\ \square\ \texttt{of}\ cl_1; \ldots; cl_n\ \texttt{end} \\
&\quad \mid \texttt{case}\ vs\ \texttt{of}\ ps\ \texttt{when}\ \square \rightarrow e^b; cl_2; \ldots; cl_n\ \texttt{end} \\
&\quad \mid \texttt{let} <x_1, \ldots, x_n> = \square\ \texttt{in}\ e_2 \mid \texttt{do}\ \square\ e_2 \\
&\quad \mid \texttt{try}\ \square\ \texttt{of} <x_1, \ldots, x_n> \rightarrow e_2\ \texttt{catch} <x_{k+1}, x_{k+2}, x_{k+3}> \rightarrow e_3 \\
K &\in \mathit{FrameStack} ::= \varepsilon \mid F :: K
\end{aligned}
$$

Figure 4: Syntax of redexes, frames, frame stacks

simulated built-in function (BIF) calls. BIFs include a collection of simple operations (e.g., addition, subtraction, type testing) and they live in the `'erlang'` module. In contrast, function application (`apply`) is used to apply any closure to the given parameters.

Evaluation and binding. In Core Erlang all expressions evaluate to either *value sequences* (denoted with $<v_1, \ldots, v_n>$ or vs) or exceptions (denoted with *exc*). Most expressions evaluate to a single value and hence yield singleton value sequences, but value lists ($<e_1, \ldots, e_n>$) evaluate to a value sequence of the same length.

Binding expressions are capable of binding any number of variables (or function identifiers in case of `letrec`). For example, if n variables are given in `try` or `let` expressions, and e_1 evaluates to a value sequence of n values, then these values will be bound to the n given variables in e_2. This is true for `case` expressions too, but in this case a list of n patterns has to be specified in each clause.

Exceptions. In Erlang implementations [9], exceptions (denoted with $\{c, v^r, v^d\}^X$, where we use the superscript X to distinguish exceptions from tuples) consist of an exception class (c) and two values describing the reason (v^r) and additional details (v^d) of the exception. These three values are bound inside the `catch` clause of a `try` expression.[1]

3.2 Frame Stacks

Next, we define the formal semantics for Core Erlang. We use $\langle K, r\rangle \longrightarrow \langle K', r'\rangle$ to denote reduction steps, where the initial configuration consists of the frame stack K and redex r, while the final configuration includes the stack K' and redex r'. Before discussing the rules of the semantics, we define the syntax of redexes, frame stacks, and a number of auxiliary definitions.

The syntax for frames, frame stacks, and redexes are presented in Figure 4. Frames are essentially non-values with one of their subexpression replaced by $\square$ (they resemble the reduction contexts of [37]). However, frames do not capture all syntactical contexts. Frames capture evaluation order by some of their parameters being values (that have already been evaluated) while others being (non-value) expressions.

[1] We note that Core Erlang implementations also allow binding only the first two values in `catch`.

Frame stacks are essentially lists: there is the empty stack ε, and the stack $F :: K$ which denotes the frame stack K with the frame F pushed onto it.

Frames for expression lists. In order to avoid duplication for multiple kinds of expressions containing parameter lists (e.g., tuples, maps, function applications)—which always need to be evaluated in the same way—we introduce frame identifiers, the parameter list frame $id(v_1, \ldots, v_{k-1}, \square, e_{k+1}, \ldots, e_n)$, and the $\square$ redex to handle empty parameter lists in a uniform way.

3.3 Auxiliary definitions

For the rest of the paper, we introduce the following concepts:

- Similarly to our previous work [18], we use σ to denote capture-avoiding, parallel substitutions. Substitutions map names to values. We use $\sigma(x)$ to denote the value that is mapped to the name x by the substitution σ.
- Applying a substitution to a redex (or single value) is denoted with $r[\sigma]$. If a concrete substitution is given, we use $r[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ which replaces the names x_i with values v_i in r.
- We also adapt the scoping rules and notations of [18] to the extended language. We use $\Gamma \vdash t$ to denote that the redex, single value (note that variables are values too), frame or frame stack t contains free names listed in the set Γ. A redex, single value, frame or frame stack t is closed, if $\emptyset \vdash t$. Moreover, $\Gamma \vdash \sigma \multimap \Delta$ denotes that the substitution σ maps names in Γ to values (v) such that $\Delta \vdash v$.
- Let $vars(p)$ denote the set of variables in pattern p.
- The function $is\_match(ps, vs)$ decides whether the list of patterns ps pairwise match to the given value sequence vs. The function $match(ps, vs)$ creates a substitution that includes the result variable-value bindings of the successful pattern matching.
- The function $names\_of(ext)$ returns the set of bound function identifiers in the list of function definitions ext.
- The function $mk\_closlist(ext)$ creates a substitution of closures based on the function definitions in ext by transforming all function definitions $f/k = \texttt{fun}(x_1, \ldots, x_n) \rightarrow e$ of ext into $f/k \mapsto clos(ext, [x_1, \ldots, x_n], e)$ (ext is used in all closures as the collection of recursive function). For further details we refer to the formalization [32].

3.4 Dynamic Semantics

In this subsection, we present the rules of the semantics. There are 4 rule categories:

(1) Rules that deconstruct an expression by extracting its first redex while putting the rest of the expression in the frame stack (Figure 5).
(2) Rules that modify the top frame of the stack by extracting the next redex and putting back the currently evaluated value into this top frame (Figure 6).
(3) Rules that remove the top frame of the stack and construct the next redex based on this removed frame (Figure 7). We also included rules here which immediately reduce an expression without modifying the stack (e.g., PFun).
(4) Rules that express concepts of exception creation, handling, or propagation (Figure 8).

Next, we discuss a number of the more complex rules, starting with rules of group 1 (Figure 5):

- SPrimOp, STuple, and SVals reduce expressions with parameter lists. They put a parameter list frame (with their corresponding frame identifier) on the top of the frame stack. To avoid handling empty parameter lists separately for each language element, $\square$ is put into the final configuration, which will be handled by PParams$_\square$ in case of an empty, or SParams$_\square$ in case of a non-empty parameter list.
- SMap starts the evaluation of a non-empty map expression by creating a parameter list frame. In this case, the use of $\square$ can be avoided, since there is at least one key expression. Note that empty maps are handled separately (in PMap$_0$) to satisfy that the number of subexpressions and values in parameter list frames for maps is always an odd number (we refer to the description of PParams on page 6 for more insights). If we used the same general rules for map evaluation as for other parameter lists, some theorems (e.g., 3.4 and 3.5) would become significantly harder to prove.
- The rest of the rules of Figure 5 extract the first redex of the given expression, and push the remaining parts onto the stack. We note that lists are evaluated in a right-to-left order in Core Erlang, this is why e_2 is extracted first in SConsTail.

Now we draw attention to the rules in groups 2 (Figure 6) and 3 (Figure 7). Observe that all of these restrict the redex in the initial configuration to be a singleton value sequence, except for the binding expressions (in line with what we said in Section 3.1), single-step reduction rules, and technical rules involving $\square$.

- SParams$_\square$ starts the evaluation of non-empty parameter lists. This is one of the two rules that expects $\square$ in the initial configuration. If there are some expressions in the parameter list frame on the top of the stack, this rule extracts the first one. Note that this rule cannot be used for map frames, since SMap handles non-empty, and PMap$_0$ handles empty maps.
- SParams extracts the next redex (e_{i+1}) from the parameter list frame on the top of the stack, if the ith expression has already been reduced to a singleton value sequence. The item in this singleton sequence is put back into the frame.
- SCallParam, SAppParam express reductions for frames with parameter lists, and behave the same way as described above for SPrimOp, STuple, and SVals.
- SCaseFail expresses the evaluation of a `case` expression if the pattern matching failed. In this case, the first clause can be removed, and the next clause needs to be checked.
- SCaseSuccess expresses the evaluation of a `case` expression if the pattern matching succeeds. In this case, the next redex to evaluate is the guard expression of the current clause, substituted by the result of the pattern matching. Note that the substitution is also applied to the body expression of the current clause.
- SCaseFalse expresses when a guard of a clause evaluates to `'false'`. In this case, the first clause can be removed, and the next clause needs to be checked.

$$\langle K, [e_1|e_2]\rangle \longrightarrow \langle [e_1|\square] :: K, e_2\rangle \quad \text{(SConsTail)}$$

$$\langle K, \texttt{let} <x_1, \ldots, x_n> = e_1 \texttt{ in } e_2\rangle \longrightarrow \langle \texttt{let} <x_1, \ldots, x_n> = \square \texttt{ in } e_2 :: K, e_1\rangle \quad \text{(SLet)}$$

$$\langle K, \texttt{do } e_1\ e_2\rangle \longrightarrow \langle \texttt{do } \square\ e_2 :: K, e_1\rangle \quad \text{(SSeq)}$$

$$\langle K, \texttt{apply } e(e_1, \ldots, e_n)\rangle \longrightarrow \langle \texttt{apply } \square(e_1, \ldots, e_n) :: K, e\rangle \quad \text{(SApp)}$$

$$\langle K, \texttt{call } e^m{:}e^f(e_1, \ldots, e_n)\rangle \longrightarrow \langle \texttt{call } \square{:}e^f(e_1, \ldots, e_n) :: K, e^m\rangle \quad \text{(SCallMod)}$$

$$\langle K, \texttt{primop } a(e_1, \ldots, e_n)\rangle \longrightarrow \langle primop(a)(\square, e_1, \ldots, e_n) :: K, \square\rangle \quad \text{(SPrimOp)}$$

$$\langle K, <e_1, \ldots, e_n>\rangle \longrightarrow \langle values(\square, e_1, \ldots, e_n) :: K, \square\rangle \quad \text{(SVals)}$$

$$\langle K, \{e_1, \ldots, e_n\}\rangle \longrightarrow \langle tuple(\square, e_1, \ldots, e_n) :: K, \square\rangle \quad \text{(STuple)}$$

$$\langle K, \sim\{e_1^k \Rightarrow e_1^v, e_2^k \Rightarrow e_2^v \ldots, e_n^k \Rightarrow e_n^v\}\sim\rangle \longrightarrow \langle map(\square, e_1^v, e_2^k, e_2^v, \ldots, e_n^k, e_n^v) :: K, e_1^k\rangle \quad \text{(SMap)}$$

$$\langle K, \texttt{case } e \texttt{ of } cl_1; \ldots; cl_n \texttt{ end}\rangle \longrightarrow \langle \texttt{case } \square \texttt{ of } cl_1; \ldots; cl_n \texttt{ end} :: K, e\rangle \quad \text{(SCase)}$$

Figure 5: Frame stack semantics rules (group 1)

$$\langle [e_1|\square] :: K, <v_2>\rangle \longrightarrow \langle [\square|v_2] :: K, e_1\rangle \quad \text{(SConsHead)}$$

$$\langle \texttt{call } \square{:}e^f(e_1, \ldots, e_n) :: K, <v^m>\rangle \longrightarrow \langle \texttt{call } v^m{:}\square(e_1, \ldots, e_n) :: K, e^f\rangle \quad \text{(SCallFun)}$$

$$\langle \texttt{call } v^m{:}\square(e_1, \ldots, e_n) :: K, <v^f>\rangle \longrightarrow \langle call(v^m, v^f)(\square, e_1, \ldots, e_n) :: K, \square\rangle \quad \text{(SCallParam)}$$

$$\langle \texttt{apply } \square(e_1, \ldots, e_n) :: K, <v>\rangle \longrightarrow \langle app(v)(\square, e_1, \ldots, e_n) :: K, \square\rangle \quad \text{(SAppParam)}$$

$$\langle \texttt{case } \square \texttt{ of } ps \texttt{ when } e^g \rightarrow e^b; cl_2; \ldots; cl_n \texttt{ end} :: K, vs\rangle \longrightarrow \langle \texttt{case } \square \texttt{ of } cl_2; \ldots; cl_n \texttt{ end} :: K, vs\rangle \quad (\text{if } \neg is\_match(ps, vs)) \quad \text{(SCaseFail)}$$

$$\langle \texttt{case } \square \texttt{ of } ps \texttt{ when } e^g \rightarrow e^b; cl_2; \ldots; cl_n \texttt{ end} :: K, vs\rangle \longrightarrow \langle \texttt{case } vs \texttt{ of } ps \texttt{ when } \square \rightarrow e^b[match(ps, vs)]; cl_2; \ldots; cl_n \texttt{ end} :: K, e^g[match(ps, vs)]\rangle \quad (\text{if } is\_match(ps, vs)) \quad \text{(SCaseSuccess)}$$

$$\langle \texttt{case } vs \texttt{ of } ps \texttt{ when } \square \rightarrow e^b; cl_2; \ldots; cl_n \texttt{ end} :: K, <\texttt{'false'}>\rangle \longrightarrow \langle \texttt{case } \square \texttt{ of } cl_2; \ldots; cl_n \texttt{ end} :: K, vs\rangle \quad \text{(SCaseFalse)}$$

$$\langle id(\square, e_1, e_2, \ldots, e_n) :: K, \square\rangle \longrightarrow \langle id(\square, e_2, \ldots, e_n) :: K, e_1\rangle \quad (\text{if } id \neq map) \quad \text{(SParams}_\square\text{)}$$

$$\langle id(v_1, \ldots, v_{i-1}, \square, e_{i+1}, e_{i+2}, \ldots, e_n) :: K, <v_i>\rangle \longrightarrow \langle id(v_1, \ldots, v_{i-1}, v_i, \square, e_{i+2}, \ldots, e_n) :: K, e_{i+1}\rangle \quad \text{(SParams)}$$

Figure 6: Frame stack semantics rules (group 2)

- The rest of the rules of Figure 6 extract the next redex from the top frame of the stack, and put back the result value (which is inside a singleton value sequence) into the frame.

We use the auxiliary function $eval(id, v_1, \ldots, v_n)$ that constructs a redex based on a frame identifier and a parameter list. We provide an informal overview of its definition here, and for the precise definition, we refer to the formalisation [32]. If

- $id = app(v)$ and $v = clos(ext, [x_1, \ldots, x_n], e)$, then $eval(app(v), v_1, \ldots, v_n) = e[mk\_closlist(ext), x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$.
- $id = app(v)$ and v is not a closure, or has an incorrect number of formal parameters, the result is an exception.
- $id = tuple$, then $eval(tuple, v_1, \ldots, v_n) = \{v_1, \ldots, v_n\}$.
- $id = values$, then $eval(values, v_1, \ldots, v_n) = <v_1, \ldots, v_n>$.
- $id = map$ and n is an even number, then

$$eval(map, v_1, \ldots, v_n) = \sim\{v_1 \Rightarrow v_2, \ldots, v_{k-1} \Rightarrow v_k\}\sim,$$

where the $k \leq n$ result values inside the map are obtained by eliminating duplicate keys and their associated values.

- $id = call(a^m, a^f)$, then $eval(call(a^m, a^f), v_1, \ldots, v_n)$ simulates the behaviour of the built-in functions of (Core) Erlang.
- $id = primop(a)$, then $eval(primop(a), v_1, \ldots, v_n)$ simulates the behaviour of primitive operations of Core Erlang.

Thereafter, we highlight some rules from group 3 (Figure 7):

- PMap$_0$, PFun, and PLetRec express single step reductions that do not include the manipulation of the frame stack.
- PValue reduces a value to a singleton value sequence. In most cases, this rule is used to evaluate atoms, integers, and empty lists, since these values do not have a corresponding expression, in contrast to tuples, maps, and non-empty lists.
- PParams$_\square$ handles the evaluation of empty parameter lists (note that maps are handled separately with PMap$_0$). This is the other rule (besides SParams$_\square$) that expects $\square$ in the initial configuration as the redex.

$$\langle K, \sim\{\}\sim\rangle \longrightarrow \langle K, <\sim\{\}\sim>\rangle \quad \text{(PMap}_0\text{)}$$

$$\langle K, \texttt{fun}(x_1, \ldots, x_n) \to e\rangle \longrightarrow \langle K, <clos(\emptyset, [x_1, \ldots, x_n], e)>\rangle \quad \text{(PFun)}$$

$$\langle K, \texttt{letrec } ext \texttt{ in } e\rangle \longrightarrow \langle K, e[mk\_closlist(ext)]\rangle \quad \text{(PLetRec)}$$

$$\langle K, v\rangle \longrightarrow \langle K, <v>\rangle \quad \text{(PValue)}$$

$$\langle id(v_1, \ldots, v_n, \Box) :: K, \Box\rangle \longrightarrow \langle K, eval(id, v_1, \ldots, v_n)\rangle \qquad (\text{if } id \neq map) \quad \text{(PParams}_\Box\text{)}$$

$$\langle id(v_1, \ldots, v_{n-1}, \Box) :: K, <v_n>\rangle \longrightarrow \langle K, eval(id, v_1, \ldots, v_n)\rangle \quad \text{(PParams)}$$

$$\langle [\Box|v_2] :: K, <v_1>\rangle \longrightarrow \langle K, <[v_1|v_2]>\rangle \quad \text{(PCons)}$$

$$\langle \texttt{case } vs \texttt{ of } ps \texttt{ when } \Box \to e^b; cl_2; \ldots; cl_n \texttt{ end} :: K, <\texttt{'true'}>\rangle \longrightarrow \langle K, e^b\rangle \quad \text{(PCaseTrue)}$$

$$\langle \texttt{let } <x_1, \ldots, x_n> = \Box \texttt{ in } e_2 :: K, <v_1, \ldots, v_n>\rangle \longrightarrow \langle K, e_2[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]\rangle \quad \text{(PLet)}$$

$$\langle \texttt{do } \Box\ e_2 :: K, <v_1>\rangle \longrightarrow \langle K, e_2\rangle \quad \text{(PSeq)}$$

Figure 7: Frame stack semantics rules (group 3)

$$\langle \texttt{case } \Box \texttt{ of } \emptyset \texttt{ end} :: K, vs\rangle \longrightarrow \langle K, \{error, if\_clause, \{\}\}^X\rangle \quad \text{(ExcCase)}$$

$$\langle K, \texttt{try } e_1 \texttt{ of } <x_1, \ldots, x_n> \to e_2 \texttt{ catch } <x_{k+1}, \ldots, x_{k+n}> \to e_3\rangle \longrightarrow \langle \texttt{try } \Box \texttt{ of } <x_1, \ldots, x_n> \to e_2 \texttt{ catch } <x_{k+1}, \ldots, x_{k+n}> \to e_3 :: K, e_1\rangle \quad \text{(STry)}$$

$$\langle \texttt{try } \Box \texttt{ of } <x_1, \ldots, x_n> \to e_2 \texttt{ catch } <x_{k+1}, \ldots, x_{k+n}> \to e_3 :: K, <v_1, \ldots, v_n>\rangle \longrightarrow \langle K, e_2[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]\rangle \quad \text{(PTry)}$$

$$\langle \texttt{try } \Box \texttt{ of } <x_1, \ldots, x_n> \to e_2 \texttt{ catch } <x_{k+1}, \ldots, x_{k+3}> \to e_3 :: K, \{c, v^r, v^d\}^X\rangle \longrightarrow \langle K, e_3[x_{k+1} \mapsto c, x_{k+2} \mapsto v^r, x_{k+3} \mapsto v^d]\rangle \quad \text{(ExcTry)}$$

$$\langle F :: K, \{c, v^r, v^d\}^X\rangle \longrightarrow \langle K, \{c, v^r, v^d\}^X\rangle \qquad (\text{if } F \neq \texttt{try } \Box \texttt{ of } <x_1, \ldots, x_n> \to e_2 \texttt{ catch } <x_{k+1}, \ldots, x_{k+n}> \to e_3) \quad \text{(ExcProp)}$$

Figure 8: Frame stack semantics rules (group 4)

- PParams handles parameter lists. If all of the expressions have been evaluated to values, then based on the frame identifier the next redex is constructed with $eval(id, v_1, \ldots, v_n)$. Note that if the frame identifier was *map*, then n is required to be an even number (i.e., there is an odd number of subvalues in the top frame, and the last value is in the second configuration cell).
- PCaseTrue is used when the guard expression of the clause is evaluated to `'true'`. The next redex is the body expression of the same clause. Note that the bindings obtained from the successful pattern matching are already substituted by SCaseSuccess.

Finally, we explain the rules for exception creation, handling and propagation (Figure 8):

- ExcCase is used when none of the clauses of a `case` expression matched, or all of the guards of the matching clauses evaluated to `'false'`. In these cases an exception is raised. Note that this is not the only option to raise exceptions: exceptions can be the result of computing $eval(id, v_1, \ldots, v_n)$.
- STry extracts the first redex from a `try` expression for evaluation. (This rule could also belong to group 1.)
- PTry is used when the first subexpression of a `try` expression evaluated to a value sequence. In this case (if the number of variables are correct) the execution continues with the expression of the first clause substituted with the resulting variable-value bindings.
- ExcTry is used when the first subexpression of a `try` expression evaluated to an exception. In this case, three variables are bound to the parts of the exceptions in the expression of the `catch` clause, and the evaluation continues with this redex.
- ExcProp describes exception propagation. If the first frame is not an exception handler, it is removed from the stack.

The evaluation relation. Now we can define the step-indexed, reflexive, transitive closure of the reductions as usual (denoted with $\langle K, r\rangle \longrightarrow^n \langle K', r'\rangle$ when the number of reduction steps is relevant, $\langle K, r\rangle \longrightarrow^* \langle K', r'\rangle$ when it is not).

3.5 Examples

Next, we show two examples on using the frame stack semantics.

Simple example. The first example involves a simple expression which demonstrates the use of the frame stack semantics' novel features:

```
try {1, call 'erlang':'div'(1,0)} of
      <X>       -> X
catch <C, R, V> -> R
```

For readability, we denote `call 'erlang':'div'(1,0)` with *div* and the frame `try □ of <X> → X catch <C, R, V> → R` for

the `try` expression with f_{try}. The first two reductions (with rules of group 1) deconstruct the expression in the configuration into frames. For tuple evaluation, parameter list frames are used. The next reduction is done by SPARAMS$_{\square}$ to extract the first expression of the tuple, which is reduced to itself inside a singleton value sequence in the next step (PVALUE), then the next parameter is extracted (SPARAMS).

$\langle \varepsilon, \texttt{try}\ \{1, div\}\ \texttt{of <X> -> X catch <C, R, V> -> R} \rangle \longrightarrow$
$\langle f_{try} :: \varepsilon, \{1, div\} \rangle \longrightarrow$
$\langle tuple(\square, 1, div) :: f_{try} :: \varepsilon, \square \rangle \longrightarrow$
$\langle tuple(\square, div) :: f_{try} :: \varepsilon, 1 \rangle \longrightarrow$
$\langle tuple(\square, div) :: f_{try} :: \varepsilon, \texttt{<1>} \rangle \longrightarrow$
$\langle tuple(1, \square) :: f_{try} :: \varepsilon, div \rangle$

In the next steps, the evaluation of the BIF `'div'` is explained. In the first steps, SCALLMOD and SCALLFUN are used to extract the module and name expressions of the inter-module call, which are reduced with PVALUE. Next, a parameter list frame is created for the evaluation of the call's parameter list (SCALLPARAM), which is evaluated similarly to the previous evaluation of the tuple's parameter list.

$\langle tuple(1, \square) :: f_{try} :: \varepsilon, div \rangle \longrightarrow$
$\langle \texttt{call}\ \square\texttt{:'div'}(1, \texttt{0}) :: tuple(1, \square) :: f_{try} :: \varepsilon, \texttt{'erlang'} \rangle \longrightarrow$
$\langle \texttt{call}\ \square\texttt{:'div'}(1, \texttt{0}) :: tuple(1, \square) :: f_{try} :: \varepsilon, \texttt{<'erlang'>} \rangle \longrightarrow$
$\langle \texttt{call 'erlang':}\square(1, \texttt{0}) :: tuple(1, \square) :: f_{try} :: \varepsilon, \texttt{'div'} \rangle \longrightarrow$
$\langle \texttt{call 'erlang':}\square(1, \texttt{0}) :: tuple(1, \square) :: f_{try} :: \varepsilon, \texttt{<'div'>} \rangle \longrightarrow$
$\langle call(\texttt{'erlang'}, \texttt{'div'})(\square, 1, \texttt{0}) :: tuple(1, \square) :: f_{try} :: \varepsilon, \square \rangle \longrightarrow$
$\langle call(\texttt{'erlang'}, \texttt{'div'})(\square, \texttt{0}) :: tuple(1, \square) :: f_{try} :: \varepsilon, 1 \rangle \longrightarrow$
$\langle call(\texttt{'erlang'}, \texttt{'div'})(\square, \texttt{0}) :: tuple(1, \square) :: f_{try} :: \varepsilon, \texttt{<1>} \rangle \longrightarrow$
$\langle call(\texttt{'erlang'}, \texttt{'div'})(1, \square) :: tuple(1, \square) :: f_{try} :: \varepsilon, \texttt{0} \rangle \longrightarrow$
$\langle call(\texttt{'erlang'}, \texttt{'div'})(1, \square) :: tuple(1, \square) :: f_{try} :: \varepsilon, \texttt{<0>} \rangle$

Next, the result of the BIF is simulated (PPARAMS), which is an exception of the `'error'` class, `'badarith'` reason, some additional details v which are irrelevant to this example. Since the parameter list frame is not an exception handler, it is removed from the stack (EXCPROP). Finally, the `try` frame (f_{try}) handles the exception (EXCTRY) and the result is the reason of the exception.

$\langle call(\texttt{'erlang'}, \texttt{'div'})(1, \square) :: tuple(1, \square) :: f_{try} :: \varepsilon, \texttt{<0>} \rangle \longrightarrow$
$\langle tuple(1, \square) :: f_{try} :: \varepsilon, \{\texttt{'error'}, \texttt{'badarith'}, v\}^X \rangle \longrightarrow$
$\langle f_{try} :: \varepsilon, \{\texttt{'error'}, \texttt{'badarith'}, v\}^X \rangle \longrightarrow$
$\langle \varepsilon, \texttt{'badarith'} \rangle \longrightarrow$
$\langle \varepsilon, \texttt{<'badarith'>} \rangle$

Running example. The second example shows the evaluation of our running example. We recall the expression from Figure 2 in Figure 9, and replace the metavariables (f, x, e_1, e_2) with concrete values (`'f'`, `L`, 1, 2, respectively).

We note that the semantics requires the `catch` clauses to bind three variables (which was based on the language specification [9]), while the compiler also works with only two, thus in the Coq

```
'f'/1 = fun (_0) ->
  case _0 of
    <L> when try let <_1> = call 'erlang':'length'(L)
                  in call 'erlang':'=='(_1, 0)
             of <Try> -> Try
             catch <T,R> -> 'false'
        -> 1
    <_3> when 'true' -> 2
    <_2> when 'true' ->
      primop 'match_fail'({'function_clause',_2})
  end
```

Figure 9: Running example from Figure 2 with substituted metavariables

formalisation, we have introduced a third variable to the `catch` clause above, which has never been used.

We denote the clauses of the `case` subexpression with cl_1, cl_2, cl_3, the `try` subexpression with *try*, and the `let` subexpression with *let*. Suppose that we apply this function to a value v. In the first three steps the singleton value v is evaluated from the head of the `case` expression. This involves transforming v into a singleton value sequence in the second step. In the next steps, rules from group 1 are used to deconstruct the complex expression until the `call` expression is reached.

$\langle \varepsilon, \texttt{case}\ v\ \texttt{of}\ cl_1; cl_2; cl_3\ \texttt{end} \rangle \longrightarrow$
$\langle \texttt{case}\ \square\ \texttt{of}\ cl_1; cl_2; cl_3\ \texttt{end} :: \varepsilon, v \rangle \longrightarrow$
$\langle \texttt{case}\ \square\ \texttt{of}\ cl_1; cl_2; cl_3\ \texttt{end} :: \varepsilon, \texttt{<}v\texttt{>} \rangle \longrightarrow$
$\langle \texttt{case <}v\texttt{> of <L> when}\ \square \longrightarrow 1; cl_2; cl_3\ \texttt{end} :: \varepsilon, try \rangle \longrightarrow$
$\langle \texttt{try}\ \square\ \texttt{of <Try> -> Try catch <T,R> -> 'false'} ::$
$\quad \texttt{case <}v\texttt{> of <L> when}\ \square \longrightarrow 1; cl_2; cl_3\ \texttt{end} :: \varepsilon, let \rangle \longrightarrow$
$\langle \texttt{let <\_1> =}\ \square\ \texttt{in call 'erlang':'=='(\_1, 0)} ::$
$\quad \texttt{try}\ \square\ \texttt{of <Try> -> Try catch <T,R> -> 'false'} ::$
$\quad \texttt{case <}v\texttt{> of <L> when}\ \square \longrightarrow 1; cl_2; cl_3\ \texttt{end} :: \varepsilon,$
$\quad \texttt{call 'erlang':'length'}(v) \rangle$

For readability, we show the evaluation of this `call` expression separately, and denote the current stack with K. Note that at this point the variable `L` has already been replaced by v. First, the module and then the function expression is turned into a singleton value sequence and put back into the frame stack (we merged these steps below). Then, the parameters are evaluated using a parameter list frame. In this case, there is one parameter, thus first SPARAMS$_{\square}$ is used, then v is reduced to a singleton value sequence, and finally, the use of PPARAMS concludes these reduction steps.

$\langle K, \texttt{call 'erlang':'length'}(v) \rangle \longrightarrow^*$
$\langle \texttt{call}\ \square\texttt{:'length'}(v) :: K, \texttt{'erlang'} \rangle \longrightarrow^*$
$\langle \texttt{call 'erlang':}\square(v) :: K, \texttt{'length'} \rangle \longrightarrow^*$
$\langle call(\texttt{'erlang'}, \texttt{'length'})(\square, v) :: K, \square \rangle \longrightarrow$
$\langle call(\texttt{'erlang'}, \texttt{'length'})(\square) :: K, v \rangle \longrightarrow$
$\langle call(\texttt{'erlang'}, \texttt{'length'})(\square) :: K, \texttt{<}v\texttt{>} \rangle \longrightarrow$
$\langle K, eval(call(\texttt{'erlang'}, \texttt{'length'}), v) \rangle$ (RESULT)

At this point, the result depends on the value v. First, let us suppose that $v = []$, and proceed with the evaluation. In this case, the

result of calling `'length'` is `0`. Let us denote the current frame stack without the first `let` frame with K_1, and the stack we get by removing the `try` frame from K_1 with K_2. The next step is to evaluate the equality check (`'=='`) expression inside `let`, which is done analogously to calling `length` above. The result is `'true'`, which is not an exception, thus it is propagated through the `try` expression. This means that the guard is true of the `case` expression, thus PCaseTrue is used followed by reducing 1 into a singleton value sequence.

$\langle$`let <_1> = □ in call 'erlang':'=='(_1, 0)` $:: K_1$, `<0>`$\rangle \longrightarrow$

$\langle K_1$, `call 'erlang':'=='(0, 0)`$\rangle \longrightarrow^*$

$\langle$`try □ of <Try> → Try`

`catch <T, R> → 'false'` $:: K_2$, `<'true'>`$\rangle \longrightarrow^*$

$\langle$`case <[]> of <L> when □ → 1;` cl_2; cl_3 `end` $:: \varepsilon$, `<'true'>`$\rangle \longrightarrow^*$

$\langle \varepsilon$, `<1>`$\rangle$

Next, we discuss the evaluation for another value. Suppose that v = `0` when the evaluation reached the point in equation Result. In this case, the result of calling `'length'` is a bad argument exception (we denote it with *badarg*). In this case, the next reduction with ExcProp removes the frame for `let`, then the exception is handled by the frame for `try` with ExcTry. The expression in the `catch` clause is `'false'`, thus the next clause of the `case` expression is checked (SCaseFalse). In this clause both the pattern matching succeeds, and the guard evaluates to `'true'`, thus the final result is `<2>` in this case.

$\langle$`let <_1> = □ in call 'erlang':'=='(_1, 0)` $:: K_1$, *badarg*$\rangle \longrightarrow$

$\langle$`try □ of <Try> → Try`

`catch <T, R> → 'false'` $:: K_2$, *badarg*$\rangle \longrightarrow$

$\langle$`case <0> of <L> when □ → 1;` cl_2; cl_3 `end` $:: \varepsilon$, `'false'`$\rangle \longrightarrow$

$\langle$`case <0> of <L> when □ → 1;` cl_2; cl_3 `end` $:: \varepsilon$, `<'false'>`$\rangle \longrightarrow$

$\langle$`case □ of <_3> when 'true' → 2;` cl_3 `end` $:: \varepsilon, 0\rangle \longrightarrow^*$

$\langle$`case <0> of <_3> when □ → 2;` cl_3 `end` $:: \varepsilon$, `'true'`$\rangle \longrightarrow^*$

$\langle \varepsilon$, `<2>`$\rangle$

For more details and examples, we refer to the formalisation [32].

3.6 Semantic Properties

According to [18, 28] (and Theorem 4.13) it is sufficient to reason about termination for programs to be equivalent, thus next we define termination. A redex terminates in frame stack K if it can be evaluated either to a value sequence or exception.

Definition 3.1 (Termination).

$$\langle K, r\rangle \Downarrow^n := \exists res : \langle K, r\rangle \longrightarrow^n \langle \varepsilon, res\rangle$$
$$\langle K, r\rangle \Downarrow := \exists n : \langle K, r\rangle \Downarrow^n$$

We note that in the Coq implementation termination is expressed as an inductive definition which is equivalent to the previous definition, because the inductive definition is simpler to use.

Finally, we highlight some properties of the semantics which were heavily used in the proofs on program equivalence. The first property expresses that adding frames to the bottom of the stack (denoted with $+\!\!+$) does not affect the behaviour.

Theorem 3.2 (Extend frame stack). *For all frame stacks K_1, K_2, K', redexes r_1, r_2, and step counters n, if $\langle K_1, r_1\rangle \longrightarrow^n \langle K_2, r_2\rangle$, then $\langle K_1 +\!\!+ K', r_1\rangle \longrightarrow^n \langle K_2 +\!\!+ K', r_2\rangle$.*

The next property expresses that whenever a redex terminates in a frame stack, the redex can be evaluated to a value sequence or exception in the empty frame stack.

Theorem 3.3 (Termination and reductions). *For all frame stacks K, redexes r, and step counters n, if $\langle K, r\rangle \Downarrow^n$ then $\exists res, k \leq n : \langle \varepsilon, r\rangle \longrightarrow^k \langle \varepsilon, res\rangle$.*

The next two properties show that the frame stack can be merged into the evaluable expression. We use $F[e]$ to substitute an expression e into the □ of frame F. While this operation is a syntactical replacement for most frames, there is one exception:

`case` *vs* `of` *ps* `when □ →` e^b; cl_2; ...; cl_n `end`

If this frame is on the top of the stack, the semantics has already substituted the pattern variables of *ps*, thus these variables should not be substituted again in the expression that replaces □ (neither in e^b). Thus for this case we define the substitution in the following way:

```
(case vs of ps when □ → e^b; cl_2; ...; cl_n end)[e^g] :=
  case <> of
    <> when e^g → e^b;
    <> when 'true' → case vs of cl_2; ...; cl_n end
  end
```

With this definition, we highlight the following two properties of the frame stack.

Theorem 3.4 (Remove frame). *For all closed frames F, closed expressions e, and all frame stacks K, if $\langle F :: K, e\rangle \Downarrow$ then $\langle K, F[e]\rangle \Downarrow$.*

The next theorem is the converse of the previous one, allowing a context frame to be pushed to the stack.

Theorem 3.5 (Add frame). *For all closed frames F, closed expressions e, and all frame stacks K, if $\langle K, F[e]\rangle \Downarrow$ then $\langle F :: K, e\rangle \Downarrow$.*

4 PROGRAM EQUIVALENCE

In this section, we show three concepts of program equivalence we investigated and formalised based on our previous [18] and related [36] work; program equivalence based on logical relations, CIU equivalence, and contextual equivalence, and show that these definitions coincide.

We express the correctness property of refactorings with contextual equivalence, but proving it for concrete expression pairs is a challenge (induction on the syntactical contexts is required in most cases). To prove equivalence of expressions, CIU equivalence is the most suitable, while for proving general properties of the equivalence relations, the logical relation-based approach is the most flexible.

4.1 Program Equivalence Based on Logical Relations

First, we define program equivalence with logical relations based on the techniques of Pitts [27, 28]. Since Core Erlang is a dynamically

typed language, we cannot rely on types (a typing judgement) to express the mutual definitions, so we formalise the relations using step-indexing [3] (following the approach of Wand et al. [36]). We start by defining the relations for closed expressions, values, frame stacks, and exceptions.

Definition 4.1 (Logical relations for closed expressions, values, exceptions and frame stacks). The following definitions are given simultaneously. First, we define the logical relation for expressions. We denote the set of related expressions with $\mathbb{E}_n$, where n is a step counter. Two expressions are related, when the first one terminates in at most n steps in a frame stack, the second also terminates (in any number of steps) in all frame stacks that are related to the stack in the first termination.

Note that this first definition is not (yet) about redexes, only about expressions. This decision is motivated by the definition of (syntactical) contextual equivalence (Definition 4.15) which coincides with the following definition.

$$(e_1, e_2) \in \mathbb{E}_n \stackrel{\text{def}}{=} (\forall m \leq n, K_1, K_2 : (K_1, K_2) \in \mathbb{K}_m \implies \langle K_1, e_1\rangle \Downarrow^m \implies \langle K_2, e_2\rangle \Downarrow)$$

We denote the set of related frame stacks with $\mathbb{K}_n$, where n is a step counter. Two stacks are related whenever the first one terminates in at most n steps in a configuration with a value sequence, exception, or $\square$, then the second stack also terminates (in any number of steps) in all configurations which contain value sequence, exception, or $\square$ that are related to the value sequence, exception, or $\square$ in the other configuration.

$$\begin{aligned}
&(K_1, K_2) \in \mathbb{K}_n \stackrel{\text{def}}{=} \\
&(\forall m \leq n, v_1, v_1', \ldots, v_l, v_l' : (v_1, v_1'), \ldots, (v_l, v_l') \in \mathbb{V}_m \implies \\
&\quad \langle K_1, <v_1, \ldots, v_l>\rangle \Downarrow^m \implies \langle K_2, <v_1', \ldots, v_l'>\rangle \Downarrow) \wedge \\
&(\forall m \leq n, exc_1, exc_2 : (exc_1, exc_2) \in \mathbb{X}_m \implies \\
&\quad \langle K_1, exc_1\rangle \Downarrow^m \implies \langle K_2, exc_2\rangle \Downarrow) \wedge \\
&(\forall m \leq n : \langle K_1, \square\rangle \Downarrow^m \implies \langle K_2, \square\rangle \Downarrow)
\end{aligned}$$

Next, we define the concept of related values (their set is denoted with $\mathbb{V}_n$, where n is a step counter). This relation defines the base cases of the mutual definitions. Two atoms, integers are related when they are equal. Two empty lists are always related, while non-empty value lists are related when their subvalues are related. Similarly, tuples and maps are related when they are related elementwise. Two closures are related, if their bodies—substituted with their recursive function definitions (*ext*, *ext'*) and pairwise-related actual parameters—are related expressions. In this case, we do not require the recursive definitions to be related, only the termination of the body expression matters. We also highlight the $<$ relation on the step counters in this relation to ensure well-founded recursion.

$$(i_1, i_2) \in \mathbb{V}_n \stackrel{\text{def}}{=} i_1 = i_2 \qquad (a_1, a_2) \in \mathbb{V}_n \stackrel{\text{def}}{=} a_1 = a_2$$

$$([], []) \in \mathbb{V}_n \stackrel{\text{def}}{=} true$$

$$([v_1|v_2], [v_1'|v_2']) \in \mathbb{V}_n \stackrel{\text{def}}{=} (v_1, v_1'), (v_2, v_2') \in \mathbb{V}_n$$

$$(\{v_1, \ldots, v_l\}, \{v_1', \ldots, v_l'\}) \in \mathbb{V}_n \stackrel{\text{def}}{=} (v_1, v_1'), \ldots, (v_l, v_l') \in \mathbb{V}_n$$

$$\begin{aligned}
&(\sim\{v_1^k \Rightarrow v_1^v, \ldots, v_l^k \Rightarrow v_l^v\}\sim, \sim\{v_1^{k'} \Rightarrow v_1^{v'}, \ldots, v_l^{k'} \Rightarrow v_l^{v'}\}\sim) \in \mathbb{V}_n \\
&\quad \stackrel{\text{def}}{=} (v_1^k, v_1^{k'}), (v_1^v, v_1^{v'}), \ldots, (v_l^k, v_l^{k'}), (v_l^v, v_l^{v'}) \in \mathbb{V}_n
\end{aligned}$$

$$\begin{aligned}
&(clos(ext, [x_1, \ldots, x_l], e), clos(ext', [x_1, \ldots, x_l], e')) \in \mathbb{V}_n \stackrel{\text{def}}{=} \\
&\quad (\forall m < n : \forall v_1, v_1', \ldots, v_l, v_l' : (v_1, v_1'), \ldots, (v_l, v_l') \in \mathbb{V}_m \implies \\
&\qquad (e[mk_closlist(ext), x_1 \mapsto v_1, \ldots, x_l \mapsto v_l], \\
&\qquad e'[mk_closlist(ext'), x_1 \mapsto v_1', \ldots, x_l \mapsto v_l']) \in \mathbb{E}_m)
\end{aligned}$$

Finally, we define the logical relation for exceptions (denoted with $\mathbb{X}_n$, where n is a step counter). Two exceptions are related, when their three subvalues are pairwise related (note that the exception classes are always atoms, thus they are related if they are equal).

$$\begin{aligned}
&(\{c, v^r, v^d\}^X, \{c', v^{r'}, v^{d'}\}^X) \in \mathbb{X}_n \stackrel{\text{def}}{=} \\
&\quad c = c' \wedge (v^r, v^{r'}) \in \mathbb{V}_n \wedge (v^d, v^{d'}) \in \mathbb{V}_n
\end{aligned}$$

Next, we also define logical relations for redexes (i.e., not only for expressions but also for values, exceptions and holes). This concept (generalised to open redexes) coincides with CIU equivalence (Theorem 4.11).

Definition 4.2 (Logical relation for redexes).

$$(r_1, r_2) \in \mathbb{R}_n \stackrel{\text{def}}{=} (\forall m \leq n, K_1, K_2 : (K_1, K_2) \in \mathbb{K}_m \implies \langle K_1, r_1\rangle \Downarrow^m \implies \langle K_2, r_2\rangle \Downarrow)$$

Similarly to the related work, relations with higher indices can distinguish more expressions, frame stacks, values, exceptions, and redexes.

Theorem 4.3 (Monotonicity of the logical relations). *For all step counters n, m, if $m \leq n$, then $R_n \subseteq R_m$ for $R \in \{\mathbb{E}, \mathbb{K}, \mathbb{V}, \mathbb{X}, \mathbb{R}\}$.*

Next, we generalise the relations for closed elements of the syntax to open elements too. For this, first we define related substitutions.

Definition 4.4 (Logical relations with closing substitutions). We denote the set of related substitutions with $\mathbb{G}_n^\Gamma$, where n is the usual step counter, and Γ is the set of free variables that are substituted with closed values by the substitutions.

$$(\sigma_1, \sigma_2) \in \mathbb{G}_n^\Gamma \stackrel{\text{def}}{=} \Gamma \vdash \sigma_1 \multimap \emptyset \wedge \Gamma \vdash \sigma_2 \multimap \emptyset \wedge (\forall x \in \Gamma : (\sigma_1(x), \sigma_2(x)) \in \mathbb{V}_n)$$

With the concept of related closing substitutions, we can define the logical relations for open expressions, values, exceptions, redexes.

$$(v_1, v_2) \in \mathbb{V}^\Gamma \stackrel{\text{def}}{=} (\forall n, \sigma_1, \sigma_2 : (\sigma_1, \sigma_2) \in \mathbb{G}_n^\Gamma \Rightarrow (v_1[\sigma_1], v_2[\sigma_2]) \in \mathbb{V}_n)$$

$$(e_1, e_2) \in \mathbb{E}^\Gamma \stackrel{\text{def}}{=} (\forall n, \sigma_1, \sigma_2 : (\sigma_1, \sigma_2) \in \mathbb{G}_n^\Gamma \implies (e_1[\sigma_1], e_2[\sigma_2]) \in \mathbb{E}_n)$$

$$(exc_1, exc_2) \in \mathbb{X}^\Gamma \stackrel{\text{def}}{=} (\forall n, \sigma_1, \sigma_2 : (\sigma_1, \sigma_2) \in \mathbb{G}_n^\Gamma \implies (exc_1[\sigma_1], exc_2[\sigma_2]) \in \mathbb{X}_n)$$

$$(r_1, r_2) \in \mathbb{R}^\Gamma \stackrel{\text{def}}{=} (\forall n, \sigma_1, \sigma_2 : (\sigma_1, \sigma_2) \in \mathbb{G}_n^\Gamma \implies (r_1[\sigma_1], r_2[\sigma_2]) \in \mathbb{R}_n)$$

$$\frac{x \in \Gamma}{(x,x) \in \mathbb{V}^\Gamma} \qquad \frac{f/k \in \Gamma}{(f/k, f/k) \in \mathbb{V}^\Gamma} \qquad \frac{}{(a,a) \in \mathbb{V}^\Gamma} \qquad \frac{}{(i,i) \in \mathbb{V}^\Gamma} \qquad \frac{}{([],[]) \in \mathbb{V}^\Gamma} \qquad \frac{(v_1,v_1'),\ldots,(v_n,v_n') \in \mathbb{V}^\Gamma}{(<v_1,\ldots,v_n>,<v_1',\ldots,v_n'>) \in \mathbb{R}^\Gamma}$$

$$\frac{(e_1,e_2) \in \mathbb{E}^{\Gamma \cup \{x_1,\ldots,x_n\}}}{(\texttt{fun}(x_1,\ldots,x_n) \to e, \texttt{fun}(x_1,\ldots,x_n) \to e_2) \in \mathbb{E}^\Gamma} \qquad \frac{(e_1,e_1') \in X^\Gamma \quad (e_2,e_2') \in X^\Gamma \quad X \in \{\mathbb{E},\mathbb{V}\}}{([e_1|e_2],[e_1'|e_2']) \in \mathbb{E}^\Gamma} \qquad \frac{(e,e'),(e_1,e_1'),\ldots,(e_n,e_n') \in \mathbb{E}^\Gamma}{(\texttt{apply}\ e(e_1,\ldots,e_n), \texttt{apply}\ e'(e_1',\ldots,e_n')) \in \mathbb{E}^\Gamma}$$

$$\frac{(e^m,e^{m'}),(e^f,e^{f'}),(e_1,e_1'),\ldots,(e_n,e_n') \in \mathbb{E}^\Gamma}{(\texttt{call}\ e^m{:}e^f(e_1,\ldots,e_n), \texttt{call}\ e^{m'}{:}e^{f'}(e_1',\ldots,e_n')) \in \mathbb{E}^\Gamma} \qquad \frac{(e_1^k,e_1^{k'}),(e_1^v,e_1^{v'}),\ldots,(e_n^k,e_n^{k'}),(e_n^v,e_n^{v'}) \in X^\Gamma \quad X \in \{\mathbb{E},\mathbb{V}\}}{(\sim\{e_1^k \Rightarrow e_1^v,\ldots,e_n^k \Rightarrow e_n^v\}\sim, \sim\{e_1^{k'} \Rightarrow e_1^{v'},\ldots,e_n^{k'} \Rightarrow e_n^{v'}\}\sim) \in X^\Gamma}$$

$$\frac{(e_1,e_1'),\ldots,(e_n,e_n') \in X^\Gamma \quad X \in \{\mathbb{E},\mathbb{V}\}}{(\{e_1,\ldots,e_n\},\{e_1',\ldots,e_n'\}) \in X^\Gamma} \qquad \frac{(e_1,e_1'),\ldots,(e_n,e_n') \in \mathbb{E}^\Gamma \quad a = a'}{(\texttt{primop}\ a(e_1,\ldots,e_n), \texttt{primop}\ a'(e_1',\ldots,e_n')) \in \mathbb{E}^\Gamma} \qquad \frac{(e,e') \in \mathbb{E}^{\Gamma \cup names\_of(ext)} \quad equiv\_ext(\Gamma, ext, ext')}{(\texttt{letrec}\ ext\ \texttt{in}\ e, \texttt{letrec}\ ext'\ \texttt{in}\ e') \in \mathbb{E}^\Gamma}$$

$$\frac{(e_1,e_1'),\ldots,(e_n,e_n') \in \mathbb{E}^\Gamma}{(<e_1,\ldots,e_n>,<e_1',\ldots,e_n'>) \in \mathbb{E}^\Gamma} \qquad \frac{(e_1,e_1') \in \mathbb{E}^\Gamma \quad (e_2,e_2') \in \mathbb{E}^{\Gamma \cup \{x_1,\ldots,x_n\}}}{(\texttt{let}\ <x_1,\ldots,x_n> = e_1\ \texttt{in}\ e_2, \texttt{let}\ <x_1,\ldots,x_n> = e_1'\ \texttt{in}\ e_2') \in \mathbb{E}^\Gamma} \qquad \frac{(e_1,e_1') \in \mathbb{E}^\Gamma \quad (e_2,e_2') \in \mathbb{E}^\Gamma}{(\texttt{do}\ e_1\ e_2, \texttt{do}\ e_1'\ e_2') \in \mathbb{E}^\Gamma}$$

$$\frac{(e,e') \in \mathbb{E}^\Gamma \quad \forall i \le n : (e_i^g,e_i^{g'}),(e_i^b,e_i^{b'}) \in \mathbb{E}^{\Gamma \cup vars(ps_i)}}{(\texttt{case}\ e\ \texttt{of}\ ps_1\ \texttt{when}\ e_1^g \to e_1^b; \ldots; ps_n\ \texttt{when}\ e_n^g \to e_n^b\ \texttt{end}, \texttt{case}\ e'\ \texttt{of}\ ps_1\ \texttt{when}\ e_1^{g'} \to e_1^{b'}; \ldots; ps_n\ \texttt{when}\ e_n^{g'} \to e_n^{b'}\ \texttt{end}) \in \mathbb{E}^\Gamma}$$

$$\frac{(e_1,e_1') \in \mathbb{E}^\Gamma \quad (e_2,e_2') \in \mathbb{E}^{\Gamma \cup \{x_1,\ldots,x_k\}} \quad (e_3,e_3') \in \mathbb{E}^{\Gamma \cup \{x_{k+1},\ldots,x_{k+n}\}}}{(\texttt{try}\ e_1\ \texttt{of}\ <x_1,\ldots,x_k> \to e_2\ \texttt{catch}\ <x_{k+1},\ldots,x_{k+n}> \to e_3, \texttt{try}\ e_1'\ \texttt{of}\ <x_1,\ldots,x_k> \to e_2'\ \texttt{catch}\ <x_{k+1},\ldots,x_{k+n}> \to e_3') \in \mathbb{E}^\Gamma}$$

$$\frac{(e,e') \in \mathbb{E}^{\Gamma \cup names\_of(ext) \cup \{x_1,\ldots,x_n\}} \quad equiv\_ext(\Gamma, ext, ext')}{(clos(ext,[x_1,\ldots,x_n],e), clos(ext',[x_1,\ldots,x_n],e')) \in \mathbb{V}^\Gamma} \qquad \frac{(v,v') \in \mathbb{V}^\Gamma}{(v,v') \in \mathbb{E}^\Gamma} \qquad \frac{(v^r,v^{r'}) \in \mathbb{V}^\Gamma \quad (v^d,v^{d'}) \in \mathbb{V}^\Gamma}{(\{c,v^r,v^d\}^X,\{c,v^{r'},v^{d'}\}^X) \in \mathbb{R}^\Gamma} \qquad \frac{(e,e') \in \mathbb{R}^\Gamma}{(e,e') \in \mathbb{E}^\Gamma} \qquad \frac{(e,e') \in \mathbb{E}^\Gamma}{(e,e') \in \mathbb{R}^\Gamma}$$

Figure 10: Compatibility properties

Next, we show the most important properties of the logical relations [11, 29, 36]. The first one is the compatibility property, which is a form of congruence. For readability, we introduce the following definition.

Definition 4.5 (Equivalence of function collections). Two function collections (*ext* and *ext'*) are related, if they bind the same names, and they are related element-wise. Function definitions (denoted with $f/k = \texttt{fun}(x_1,\ldots,x_n) \to e$ and $f/k = \texttt{fun}(x_1,\ldots,x_n) \to e'$) of the collections *ext* and *ext'* are related, if their bodies are related expressions, i.e. $(e,e') \in \mathbb{E}^{\Gamma \cup names\_of(ext) \cup \{x_1,\ldots,x_n\}}$. We use $equiv\_ext(\Gamma, ext, ext')$ to denote this property.

Theorem 4.6 (Expression Compatibility). *The logical relations satisfy the syntactical compatibility properties listed in Figure 10.*

The compatibility theorem basically defines a way to construct complex equivalent expressions by combining smaller equivalent expressions. Based on the previous theorem, we can prove that the logical relation for redexes and expressions coincide for related expressions.

Corollary 4.7 (Equivalence of logical relations). *For all expressions e_1, e_2 and scopes Γ, $(e_1,e_2) \in \mathbb{E}^\Gamma \iff (e_1,e_2) \in \mathbb{R}^\Gamma$.*

Another consequence of the compatibility theorem is the fundamental property of the relations, a form of reflexivity expressing that any expression (and similarly, any value, exception, redex or closing substitution) is indistinguishable from itself.

Theorem 4.8 (Fundamental property). *For all scopes Γ the following properties hold:*

- *For all expressions e, if $\Gamma \vdash e$ then $(e,e) \in \mathbb{E}^\Gamma$;*
- *For all values v, if $\Gamma \vdash v$ then $(v,v) \in \mathbb{V}^\Gamma$;*
- *For all exceptions exc, if $\Gamma \vdash exc$ then $(exc,exc) \in \mathbb{X}^\Gamma$;*
- *For all redexes r, if $\Gamma \vdash r$ then $(r,r) \in \mathbb{R}^\Gamma$;*
- *For all closing substitutions σ, if $\Gamma \vdash \sigma \multimap \emptyset$ then for all step counters n, $(\sigma,\sigma) \in \mathbb{G}_n^\Gamma$ holds.*

Last but not least, another important property of the logical relations for values is that all related values should be equal by the built-in equality of (Core) Erlang (simulated by the auxiliary function $eval(call(\texttt{'erlang'},\texttt{'=='}),v_1,v_2)$) with the limitation that two closures are always considered as equal values[2]. If this consideration is not made, and only syntactically equal functions are considered as equal (although, this still does not reflect the behaviour of the compiler), the converse of the following theorem would be true.

Theorem 4.9 (Equivalent values are equal). *For all values v_1, v_2, step counters m,*

$$(v_1,v_2) \in \mathbb{V}_m \implies eval(call(\texttt{'erlang'},\texttt{'=='}),v_1,v_2) = \texttt{'true'}.$$

[2]Closure comparison is undocumented in the reference manual [1].

4.2 CIU Equivalence

Next, we introduce CIU ("closed instances of use") preorder and equivalence.

Definition 4.10 (CIU preorder). Two redexes are CIU equivalent if they both terminate or diverge when placed in arbitrary frame stacks.

$$r_1 \leq_{ciu} r_2 \stackrel{\text{def}}{=} \emptyset \vdash r_1 \wedge \emptyset \vdash r_2 \wedge (\forall K : \emptyset \vdash K \wedge \langle K, r_1\rangle \Downarrow \implies \langle K, r_2\rangle \Downarrow)$$

$$r_1 \equiv_{ciu} r_2 \stackrel{\text{def}}{=} r_1 \leq_{ciu} r_2 \wedge r_2 \leq_{ciu} r_1$$

We extend these concepts to open redexes with closing substitutions.

$$r_1 \leq^{\Gamma}_{ciu} r_2 \stackrel{\text{def}}{=} \forall \sigma : \Gamma \vdash \sigma \multimap \emptyset \implies r_1[\sigma] \leq_{ciu} r_2[\sigma]$$

$$r_1 \equiv^{\Gamma}_{ciu} r_2 \stackrel{\text{def}}{=} r_1 \leq^{\Gamma}_{ciu} r_2 \wedge r_2 \leq^{\Gamma}_{ciu} r_1$$

In most cases, it is simpler to prove redexes CIU equivalent, than using logical relations or contextual equivalence, because CIU equivalence involves reasoning with respect to a single frame stack instead of two related ones, or one syntactical context. One of the most important properties of CIU equivalence is that it coincides with logical relations on redexes.

Theorem 4.11 (CIU coincides with the logical relations). *For all redexes r_1, r_2, and scopes Γ, $r_1 \leq^{\Gamma}_{ciu} r_2$ if and only if $(r_1, r_2) \in \mathbb{R}^{\Gamma}$.*

Another major property of CIU equivalence is that evaluating a redex results in an equivalent value sequence or exception.

Theorem 4.12 (Redexes are equivalent to their results). *For all closed redexes r, and results res, if $\langle \varepsilon, r\rangle \longrightarrow^* \langle \varepsilon, res\rangle$, then $r \equiv_{ciu} res$.*

Finally, we highlight one last property which expresses the fact that reasoning about termination of programs is sufficient for the final results to be equivalent.

Theorem 4.13 (Termination is sufficient). *For all closed values v_1, v_2, if $v_1 \leq_{ciu} v_2$ then for all step indices n, $(v_1, v_2) \in \mathbb{V}_n$.*

The idea to prove this theorem is (also described in [18, 27]) that whenever two values are not related by $\mathbb{V}$, then it is possible to construct an evaluation context in which one of the values terminates while the other one diverges.

This theorem together with Theorem 4.12 and the transitivity of the equivalence relations (Section 4.3) means that whenever two expressions are CIU equivalent, their values will be related by the logical relation for values, which expresses exactly what we expect from the behaviour of equivalent values.

4.3 Contextual Equivalence

Finally, we define contextual preorder and equivalence following the techniques of Wand et al. [36].

Definition 4.14 (Contextual preorder). We define the contextual preorder to be the largest family of relations R^{Γ} that satisfy the following properties:

- Adequacy: $(e_1, e_2) \in R^{\emptyset} \implies \langle \varepsilon, e_1\rangle \Downarrow \implies \langle \varepsilon, e_2\rangle \Downarrow$.
- Reflexivity: $(e, e) \in R^{\Gamma}$.
- Transitivity: $(e_1, e_2) \in R^{\Gamma} \wedge (e_2, e_3) \in R^{\Gamma} \implies (e_1, e_3) \in R^{\Gamma}$.
- Compatibility: R^{Γ} satisfies the compatibility rules for every expression from Figure 10 (these are the rules that involve $\mathbb{E}$ both in the premises and conclusion).

This definition is equivalent to the usual, syntax-based definition of contextual equivalence. We denote syntactical expression contexts with C (where one of the subexpressions is replaced by a unique variable $\square$), and use $C[e]$ to denote the substitution of $\square$ with expression e in context C.

Definition 4.15 (Syntax-based contextual preorder and equivalence).

$$e_1 \leq^{\Gamma}_{ctx} e_2 \stackrel{\text{def}}{=} \Gamma \vdash e_1 \wedge \Gamma \vdash e_2 \wedge (\forall (C : \mathit{Context}) : \emptyset \vdash C[e_1] \wedge \emptyset \vdash C[e_2] \implies \langle \varepsilon, C[e_1]\rangle \Downarrow \implies \langle \varepsilon, C[e_2]\rangle \Downarrow)$$

$$e_1 \equiv^{\Gamma}_{ctx} e_2 \stackrel{\text{def}}{=} e_1 \leq^{\Gamma}_{ctx} e_2 \wedge e_2 \leq^{\Gamma}_{ctx} e_1$$

The concepts above are all defined only for expressions, and not redexes. The reason for this is only expressions are syntactically valid Core Erlang expressions. There is no way to include an exception as a syntactical subexpression, because it is a semantical concept.

On the one hand, the previous definition of contextual equivalence expresses the correctness property of refactorings, that is replacing two equivalent expressions in any syntactical context preserves the behaviour, On the other hand, reasoning about contextual equivalence naively would require induction on the structure of the context. To tackle this issue, we proved that contextual equivalence coincides with CIU equivalence for expressions.

Theorem 4.16 (CIU theorem). *For all expressions e_1, e_2, and scopes Γ, $e_1 \leq^{\Gamma}_{ctx} e_2$ if and only if $e_1 \leq^{\Gamma}_{ciu} e_2$.*

As a consequence, we can state the following corollary on the connections between the equivalence concepts.

Corollary 4.17 (Coincidence of equivalences). *For all expressions e_1, e_2, and scopes Γ, the following equivalences hold:*

$$(e_1, e_2) \in \mathbb{E}^{\Gamma} \iff (e_1, e_2) \in \mathbb{R}^{\Gamma} \iff e_1 \leq^{\Gamma}_{ciu} e_2 \iff e_1 \leq^{\Gamma}_{ctx} e_2$$

4.4 Refactoring Correctness

With contextual equivalence, we can express the correctness property of refactorings. A local refactoring which replaces a subexpression e with e' is correct, if $e \equiv^{\Gamma}_{ctx} e'$ (supposing that e and e' contain the free variables in Γ). Proving the contextual equivalence implies that the behaviour of the entire context (i.e., a whole program) does not change when the two equivalent expressions are replaced.

As mentioned previously, reasoning about contextual equivalence is not simple in most cases, thus we prove CIU equivalence of expressions instead, and use Theorem 4.16 to establish the contextual equivalence. We proved the correctness of simple Erlang refactorings (two examples are Figure 1 and 11). We note that these two refactorings were motivated by the work of Poór et al. [31]. Moreover, a version of the example presented in Figure 1 is actually implemented in a refactoring tool [31].

To argue about these transformations, first we translated these programs to Core Erlang with the standard Erlang/OTP compiler

```
case e1 of true -> e2;                 if e1 -> e2;
           _ -> e3                        true -> e3
end                          ⟶        end
```

Figure 11: Expression refactoring example

(which we handled as trusted component for the proving process). Next, we encoded the Core Erlang programs in the Coq formalisation and proved their equivalence. We based the equivalence proofs on the termination of the expressions, that is we inspected all possible termination paths for one and proved the termination of the other expression, based on the properties we obtained from the first evaluation. For this step, the inductive definition of the frame stack termination (remark after Definition 3.1) proved to be extremely useful as its rules are (mostly) syntax-driven and most of them are free of side conditions.

5 DISCUSSION

All results presented here are formalised in the Coq proof management system [32]. In this section, we highlight a number of challenges we faced during the implementation.

Syntax. Initially, we considered two other approaches to formalise the syntax: (1) values are completely separated from expressions and (2) there are only expressions and a judgement which determines whether an expression is a value (i.e., is in normal form). The issue with (1) is applying substitutions requiring the substituted values to be transformed to expressions, leading to loss of information in case of closures (the list of recursive definitions would simply be lost). The disadvantage of (2) is that the value judgement relation needs to be used in most of the rules of the semantics to ensure determinism, which on the other hand leads to more proof steps about the evaluation. However, the approach presented in Section 3.1 does not come without drawbacks either: using mutually inductive types in Coq leads to more complicated induction principles (which had to be defined manually) and theorem statements about the syntax.

Semantics. The main advantage of the frame stack semantics is that most of the rules can be applied in a syntax-directed way, which significantly simplifies proving evaluation. The notion of parameter list frames was motivated by the implementation to avoid the duplication of frames, reduction rules, theorems for similar language elements: tuples, maps, inter-module calls, primitive operations, and function applications. With parameter list frames, these features can be handled in a unified way. This notion was also used for reduction contexts by Fredlund [13].

Logical relations. We formalised logical relations with definitions that are parametrised by the step-indexed value relation ($\mathbb{V}_n$) instead of mutually inductive types, following the footsteps of Wand et al. [36]. This way, we also avoided the strict positivity checks of Coq for inductive types.

Induction principles. Further interesting points in the formalisation are induction principles. We highlight the induction principle for the logical relation on values ($\mathbb{V}_n$). While using induction on the logical relation, only relevant cases that contain related values need to be proved, this means the cases for equal literal pairs, related tuple pairs, related list pairs etc., supposing that the subvalues are related and satisfy the induction hypothesis. This way we do not need to derive a contradiction from premises such as $(i, []) \in \mathbb{V}_n$ which would be needed if we used case separation on the structure of the related value pairs.

Further examples of program equivalence. Besides the refactorings described in Section 4.4, we formalised a number of other simple expression equivalences specifically for Core Erlang (namely, different versions of beta-reduction, list folding and mapping). For more details, we refer to the formalisation [32].

6 RELATED WORK

Our previous work and the result here on Core Erlang is based on the language specification [9] and related research. The most influential ones are reversible semantics for Erlang [19, 20, 25], a framework for reasoning about Erlang [13], symbolic execution [35], and abstraction and model checking [24].

In related work, CIU equivalence [3, 8, 11, 14, 22, 23, 36] and logical relations (either type-indexed [28, 34] or step-(and type-) indexed [3, 11, 29, 36]) were successfully applied for a wide variety of languages (e.g., different variants of lambda calculi, imperative languages). Most of the related works—that define CIU equivalence—use a continuation-style semantics, similarly to our case where the frame stack can be seen as the continuation. The novelty of our work lies with the choice of the language, the extent of the language elements formalised, and the machine-checked implementation.

In the related literature, there are other options to formalise program equivalence. The most simple notion is behavioural equivalence [26] which is based on syntactical equality of values. Another approach is using bisimulations [2, 21, 27, 33] which are relations between programs preserved by the reduction steps; however, using coinductively defined bisimulations in Coq is challenging [12] thus we avoided this approach for the sequential sublanguage.

7 CONCLUSION AND FUTURE WORK

In this paper, we defined a formal syntax and a frame stack semantics for sequential Core Erlang. Thereafter, we presented a number of properties of this semantics, and defined three expression equivalence concepts (based on logical relations, CIU equivalence, and contextual equivalence). We showed that these termination-based equivalences are sufficient to ensure the final results of equivalent programs to be behaviourally indistinguishable. Moreover, we also showed that these three equivalence concepts coincide for (Core Erlang) expressions.

In the short term future, we are going to extensively validate the frame stack semantics presented here by showing its equivalence with the validated big-step semantics in our previous work [6]. In the longer term, we are going to combine this work with related research on the concurrent subset of Core Erlang [7, 21], first focusing on the single-node semantics.

We also plan to investigate more complex (non-local) refactorings based on the semantics and equivalence concepts defined here. We plan to base the correctness of refactoring concurrent programs on bisimulation-based program equivalence relations, and expect challenges in reasoning about concurrent errors (e.g., exit signals) and process termination.

ACKNOWLEDGMENTS

Supported by the ÚNKP-22-3 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

REFERENCES

[1] 2021. *Erlang/OTP compiler, version 24.0.* https://www.erlang.org/patches/otp-24.0 Accessed on 8th March, 2024.

[2] S. Abramsky and C.H.L. Ong. 1993. Full Abstraction in the lazy lambda calculus. *Information and Computation* 105, 2 (1993), 159–267. https://doi.org/10.1006/inco.1993.1044

[3] Amal Ahmed. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *Programming Languages and Systems*, Peter Sestoft (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 69–83. https://doi.org/10.1007/11693024_6

[4] Péter Bereczky, Dániel Horpácsi, and Simon Thompson. 2020. A proof assistant based formalisation of a subset of sequential Core Erlang. In *Trends in Functional Programming*, Aleksander Byrski and John Hughes (Eds.). Springer International Publishing, Cham, 139–158. https://doi.org/10.1007/978-3-030-57761-2_7

[5] Péter Bereczky, Dániel Horpácsi, and Simon J. Thompson. 2020. Machine-checked natural semantics for Core Erlang: exceptions and side effects. In *Proceedings of Erlang 2020*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3406085.3409008

[6] Péter Bereczky, Dániel Horpácsi, Judit Kőszegi, Soma Szeier, and Simon Thompson. 2021. Validating formal semantics by property-based cross-testing. In *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20)*. Association for Computing Machinery, New York, NY, USA, 150–161. https://doi.org/10.1145/3462172.3462200

[7] Péter Bereczky, Dániel Horpácsi, and Simon Thompson. 2024. A Formalisation of Core Erlang, a concurrent actor language. *Acta Cybernetica* (2024). https://doi.org/10.14232/actacyb.298977

[8] Lars Birkedal, Aleš Bizjak, and Jan Schwinghammer. 2013. Step-indexed relational reasoning for countable nondeterminism. *Logical Methods in Computer Science* Volume 9, Issue 4 (Oct. 2013), 22 pages. https://doi.org/10.2168/LMCS-9(4:4)2013

[9] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. 2004. Core Erlang 1.0.3 language specification. https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf Accessed on 8th December, 2023.

[10] Francesco Cesarini and Simon Thompson. 2009. *Erlang programming* (1st ed.). O'Reilly Media, Inc., Sebastopol, California, USA.

[11] Ryan Culpepper and Andrew Cobb. 2017. Contextual equivalence for probabilistic programs with continuous random variables and scoring. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 368–392. https://doi.org/10.1007/978-3-662-54434-1_14

[12] Jörg Endrullis, Dimitri Hendriks, and Martin Bodin. 2013. Circular coinduction in Coq using bisimulation-up-to techniques. In *Interactive Theorem Proving*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 354–369. https://doi.org/10.1007/978-3-642-39634-2_26

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

[14] Andrew D Gordon, Paul D Hankin, and Søren B Lassen. 1999. Compilation and equivalence of imperative objects. *Journal of Functional Programming* 9, 4 (1999), 373–426. https://doi.org/10.1007/BFb0058024

[15] Kofi Gumbs. 2017. The core of Erlang. https://8thlight.com/blog/kofi-gumbs/2017/05/02/core-erlang.html Accessed on 8th December, 2023.

[16] Björn Gustavsson. 2020. *EEP 52: Allow key and size expressions in map and binary matching.* https://www.erlang.org/eeps/eep-0052

[17] Dániel Horpácsi, Judit Köszegi, and Simon Thompson. 2016. Towards trustworthy refactoring in Erlang. *Electronic Proceedings in Theoretical Computer Science* 216 (2016), 83–103. https://doi.org/10.4204/EPTCS.216.5 arXiv:1607.02228

[18] Dániel Horpácsi, Péter Bereczky, and Simon Thompson. 2023. Program equivalence in an untyped, call-by-value functional language with uncurried functions. *Journal of Logical and Algebraic Methods in Programming* 132 (2023), 100857. https://doi.org/10.1016/j.jlamp.2023.100857

[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] Ian Mason and Carolyn Talcott. 1991. Equivalence in functional languages with effects. *Journal of Functional Programming* 1, 3 (1991), 287–327. https://doi.org/10.1017/S0956796800000125

[23] Craig McLaughlin, James McKinna, and Ian Stark. 2018. Triangulating Context Lemmas. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA). Association for Computing Machinery, New York, NY, USA, 102–114. https://doi.org/10.1145/3167081

[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] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2022. Software foundations. https://softwarefoundations.cis.upenn.edu/ Accessed on 8th December, 2023.

[27] Andrew Pitts. 1997. *Operationally-based theories of program equivalence.* Cambridge University Press, Cambridge, UK, 241–298. https://doi.org/10.1017/CBO9780511526619.007

[28] Andrew M. Pitts. 2002. Operational semantics and program equivalence. In *Applied Semantics*, Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 378–412. https://doi.org/10.1007/3-540-45699-6_8

[29] Andrew M. Pitts. 2010. Step-indexed biorthogonality: a tutorial example. In *Modelling, Controlling and Reasoning About State (Dagstuhl Seminar Proceedings (DagSemProc), Vol. 10351)*, Amal Ahmed, Nick Benton, Lars Birkedal, and Martin Hofmann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1–10. https://doi.org/10.4230/DagSemProc.10351.6

[30] Gordon D Plotkin. 1981. *A structural approach to operational semantics.* Aarhus university, Aarhus, Denmark.

[31] Boldizsár Poór, Melinda Toth, and István Bozó. 2020. Transformations towards clean functional code. In *Proceedings of the 19th ACM SIGPLAN International Workshop on Erlang* (Virtual Event, USA) *(Erlang 2020)*. Association for Computing Machinery, New York, NY, USA, 24–30. https://doi.org/10.1145/3406085.3409010

[32] High-Assurance Refactoring Project. 2024. Core Erlang formalization. https://github.com/harp-project/Core-Erlang-Formalization/releases/tag/v1.0.4 Accessed on 11th March, 2024.

[33] Alex Simpson and Niels Voorneveld. 2019. Behavioural equivalence via modalities for algebraic effects. *ACM Trans. Program. Lang. Syst.* 42, 1, Article 4 (Nov. 2019), 45 pages. https://doi.org/10.1145/3363518

[34] Benjamin C. Sumii, Eijiro; Pierce. 2003. Logical relations for encryption. *Journal of Computer Security* 11 (2003), 521–554. https://doi.org/10.3233/JCS-2003-11403

[35] 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

[36] Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. 2018. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *Proc. ACM Program. Lang.* 2, ICFP, Article 87 (July 2018), 30 pages. https://doi.org/10.1145/3236782

[37] A.K. Wright and M. Felleisen. 1994. A syntactic approach to type soundness. *Information and Computation* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093