Program Equivalence in Sequential Core Erlang

Dániel Horpácsi
ELTE, Eötvös Loránd University, Hungary
daniel-h@elte.hu

Péter Bereczky
ELTE, Eötvös Loránd University, Hungary
berpeti@inf.elte.hu

Simon Thompson
ELTE, Eötvös Loránd University, Hungary
University of Kent, United Kingdom
s.j.thompson@kent.ac.uk

Our research aims to reason about the correctness of refactoring transformations of Erlang programs. As a stepping stone, we have developed formal semantics for sequential Core Erlang in the Coq proof assistant and we investigate reasoning about program transformations in Core Erlang. By refactoring, we mean program transformations that are expected to preserve the observable behaviour of programs. This property is characterised by a behavioural equivalence relation defined over the formal definition of semantics.

In this paper, we adapt fundamental formalisations of expression equivalence to a Core Erlang subset, ranging from the simplest behavioural equivalence to logical relations and contextual equivalence. We examine the properties of these equivalence definitions and formally establish connections among them. The results are implemented in the Coq proof assistant.

1 Introduction

Most language processors and refactoring tools lack a formal, precise specification of how the code is affected by the changes they may make. In particular, refactoring tools are expected not to change the behaviour of any program, but in practice, this property is only validated by testing. This form of verification may or may not provide trust in users willing to refactor industrial-scale code. Higher assurance can be achieved by making formal arguments to verify semantics-preservation (behaviour-preservation). This requires a rigorous, formal definition of the programming language under refactoring, a precise description of the refactoring's effect on the program, and a suitable definition of program equivalence.

The research presented in this paper is part of a wider project dedicated to improving the trustworthiness of Erlang refactorings [12]. As a stepping stone, we focus on Erlang's intermediate language, Core Erlang, and we investigate refactoring correctness and program equivalence on this lower level language first. In order to make the verification machine-checkable, we have developed natural semantics for Core Erlang [1, 2] in Coq. Although our ultimate goal is to prove Erlang refactorings correct, results on Core Erlang may contribute to other research projects that target languages in the BEAM family, that is languages that translate to Core Erlang during the compilation process, e.g. Elixir or Erlang [10]). In particular, given a trusted translation to Core Erlang, we can reason about semantic equivalence, and therefore refactoring step correctness, for such a language.

In this paper we investigate a number of approaches to defining semantic equivalence of Core Erlang programs and compare them with respect to their appropriateness for reasoning about refactoring correctness. In the remainder of the paper, Section 2 presents the refactoring definition and verification approach that motivates the fine-grained definition of semantic equivalence, and in Section 3 we

Submitted to:
WPTE 2021

briefly survey related work. Then in Section 4 we overview the various equivalence definitions for Core Erlang, including simple behavioural equivalence [22] and contextual equivalence based on logical relations [23, 29], pointing out their advantages and disadvantages. Finally, Section 5 concludes.

2 Motivation

The concept of semantic equivalence between programs (or patterns of programs in general) plays a key role in the verification of refactoring. A refactoring is a program transformation that is expected to preserve behaviour [8]; that is, the program's observable behaviour should not be affected by the various structural changes made by the refactoring. The behaviour-preservation is typically characterised with semantics preservation, or slightly relaxed, semantic equivalence. The latter has to be defined carefully: the internal workings of the program may be altered, but the semantic equivalence has to imply that the transformed program is observationally indistinguishable from the original when run in an arbitrary environment.

Local transformations. The typical refactoring process, i.e. reworking a piece of code to increase its quality, involves multiple, smaller refactoring steps, called *prime* or *micro-refactorings*. These smaller steps may vary from reordering arguments or eliminating variables to extracting code portions to function abstractions. Even though these modifications are likely to be local to syntactic segments of code (e.g. expressions, functions or modules), an entire software project has to be taken into account when reasoning about the correctness of the whole refactoring step; to take a concrete example, the renaming of a function will affect not only the module in which the function is defined, but also every module in which the function is called.

To avoid reasoning about equivalence of extensive code bases several times, we can outline the scope of each smaller transformation and reason about the correctness in two steps: whether the transformation is locally correct, and whether the local equivalence implies semantic equivalence in the scope of the entire code base. The syntactically local changes may be seen as pairs of concrete expressions: the original and transformed versions of the code fragments. Reasoning about correctness in this case is done by establishing *contextual equivalence* between the original and result expressions.

In practice, many of the small changes are instances of a similar transformation logic, implementing a form of parametric expression rephrasing. Furthermore, these typical local transformations can easily be defined by using conditional term rewrite rules — the verification of such refactoring steps may be carried out by checking a *conditional contextual equivalence* between the matching and replacement patterns in the rewrite rule.

Refactoring schemes. On the other hand, many micro-refactoring steps are not syntactically local, they can span across compilation units via semantic dependencies such as data-flow and control-flow via inter-module calls. In our terminology we say that such transformations are *extensive* and they are carried out in a semantic context, e.g. along the expression of a data-flow chain.

Basically, *extensive refactoring* is a special composition of local changes, following the semantic dependencies in the program (e.g. data-flow or binding dependencies). Such transformations are not local to expressions nor to modules (consider, for instance, renaming of functions or altering of data type definitions), but they are likely to be local to *a program slice*. In our previous work, we argued that the general definition of extensive refactoring can be given with so-called *refactoring schemes* [11, 12], which define the above-mentioned special combination of local rewrite rules using semantic rewrite

strategies. The verification of extensive steps involves reasoning about the locality to a *program slice*, the local correctness within the slice, and about the local consistency implying program-wide equivalence. Again, the local correctness is expressed in terms of a set of conditional contextual equivalence statements, while the argument of the local correctness implying equivalence on the code base level needs inductive reasoning with the semantics in general.

Nevertheless, the verification of both structurally local and (semantics-driven) extensive code transformations involves checking equivalence between first-order terms or expressions. It is therefore the main motivation of this work to provide basis for reasoning about refactorings by establishing the appropriate semantics and equivalence definitions.

3 Related Work

Formal semantics for Core Erlang. In the early stages of our project, we developed an inductive big-step semantics for sequential Core Erlang [1, 2] considering exceptions and simple side effects, which has also been implemented in Coq. This semantics is based on related research on Erlang and Core Erlang, e.g. reversible semantics and debugging [15, 14, 20], a framework for reasoning about Erlang [9] and symbolic execution [28]. We have also investigated different big-step definition styles [4], and recently we also implemented an equivalent functional big-step semantics [21] which enabled extensive validation [3].

Program equivalence. There are different approaches to determining semantic equivalence, including simple behavioural equivalence [22], bisimulation [13, 18], behavioural equivalence by means of a dedicated logic [27], and contextual equivalence [23, 29], in which logical relations are used to prove the equivalence between programs.

With the functional big-step semantics we have developed, we can "step-index" the equivalence relations using the recursion depth limit, moreover, we can also prove properties of logical relations, as Owens et al. point out [21]. Another suitable way to work with logical relations and contextual equivalence is using a "frame stack" semantics [25] which enables the evaluation of expressions in arbitrary reduction contexts (or evaluation frames). We also investigated this semantics definition style.

Another approach is to use algorithms that can find proofs for program equivalence; however, for these to work we need either an operational semantics based on term rewriting [16], or reasoning in matching logic [5]; there is ongoing work of ours to formalise matching logic in Coq [17].

The results presented in this paper are based on the work by Pitts [23, 24] and Culpepper et al. [7, 29]; we investigate how to adapt to Core Erlang the methods they used to establish equivalence definitions using logical relations and CIU and contextual pre-orders, and related theorems, prove the equality of these relations.[1] All of our results are formalised in the Coq proof assistant [6].

4 Program equivalence definitions

In order to be able to reason about the correctness of refactoring, a suitable program equivalence definition is needed for the object language. This section summarises two approaches: simple behavioural equivalence, and contextual equivalence based on step-indexed logical relations.

[1] We have proved the logical relations and the CIU-preoder equal, and now we investigate the equality of contextual preorder and CIU-preoder

Side effects. Both Erlang and Core Erlang are *impure* languages, that is, expressions can have side effects. Our formalisation of Core Erlang is able to interpret simple forms of side effects and accumulate them in a log, but it needed to be investigated whether side effects should count toward equivalence (regardless of the definition style).

- **Complete behavioural equivalence.** Strict equivalence is when two evaluable expressions produce the same results and the same side effects in the same order. A simple Core Erlang example for complete behavioural equivalence can be the following:

  $e \equiv$ `do call 'erlang':'+'(2, 2)` e

  This equivalence expresses that any expression e fully preserves its semantics when preceded by a pure expression $(2+2)$ in a sequential composition.

- **Weak behavioural equivalence.** Depending on the context, some input or output side effects may be reordered without changing the observable behaviour. Therefore, we defined a weaker equivalence as well, which still requires the expressions to produce the same results and the same set of side effects, but allows these effects to resolve in a different order. For instance, with this we can prove the following two expressions (weakly) equivalent, as the only difference between them is that the first two side effects (produced by the two *fwrite* applications) are swapped:

  `do (do call 'io':'fwrite'(1) call 'io':'fwrite'(2))` $e \equiv_{weak}$
  `do (do call 'io':'fwrite'(2) call 'io':'fwrite'(1))` e

4.1 Behavioural Equivalence

On our first attempt, we investigated program equivalence by the textbook definition, i.e. simple behavioural equivalence [22]: two expressions are said to be equivalent if and only if in every starting configuration they evaluate to the "same result", or they both diverge. We defined both complete and weak equivalence that treat side effects slightly differently, using a functional big-step semantics of Core Erlang [19]. Note that in the following formulas *eval* is the semantic function, d is the recursion depth limit, *eff*, *eff'* denote the initial and final side effect logs, and *res* is the final result (a value or an exception).

- **Complete behavioural equivalence.**

$$
\begin{aligned}
&e_1 \equiv e_2 := \\
&\quad \forall \mathit{eff}, \mathit{res}, \mathit{eff'} : \\
&\qquad (\exists d : \mathit{eval}\ d\ e_1\ \mathit{eff} = (\mathit{res}, \mathit{eff'})) \iff \\
&\qquad (\exists d : \mathit{eval}\ d\ e_2\ \mathit{eff} = (\mathit{res}, \mathit{eff'}))
\end{aligned}
$$

- **Weak behavioural equivalence.**

$$
\begin{aligned}
&e_1 \equiv_{\mathit{weak}} e_2 := \\
&\quad (\forall \mathit{eff}, \mathit{res}, \mathit{eff'} : (\exists d : \mathit{eval}\ d\ e_1\ \mathit{eff} = (\mathit{res}, \mathit{eff'})) \Rightarrow \\
&\qquad (\exists \mathit{eff''} : (\exists d : \mathit{eval}\ d\ e_2\ \mathit{eff} = (\mathit{res}, \mathit{eff''})) \wedge \mathit{Permutation\ eff'\ eff''})) \wedge \\
&\quad (\forall \mathit{eff}, \mathit{res}, \mathit{eff'} : (\exists d : \mathit{eval}\ d\ e_2\ \mathit{eff} = (\mathit{res}, \mathit{eff'})) \Rightarrow \\
&\qquad (\exists \mathit{eff''} : (\exists d : \mathit{eval}\ d\ e_1\ \mathit{eff} = (\mathit{res}, \mathit{eff''})) \wedge \mathit{Permutation\ eff'\ eff''}))
\end{aligned}
$$

After defining our equivalence relations, we needed to prove that they are reflexive, symmetric and transitive. Another important aspect of behavioural equivalence is that it should be a congruence [22]; this property helps proving compound expressions equivalent, and took significant effort to formalise and prove in Coq [19].

It is worth mentioning that to prove weak equivalence congruent, we needed to prove that the log of (the currently formalised) side effects does not affect the evaluation. This may change if the side effects are formalised at a different level of granularity. We also proved the natural property of complete equivalence implying weak equivalence.

Equivalence of function expressions. This definition of behavioural equivalence deals with the equivalence of values and side effects. Although values of most types can be checked for equality trivially, there is a special case: functions. For example:

```
fun(X) -> X + 2 ≡ fun(X) -> (X + 1) + 1
```

In Erlang and in Core Erlang, function expressions evaluate to closures (the function normal form, with all of its free variables substituted in its body expression). In this particular case, we would need to prove the following closure equality: $Clos\ [X]\ (X+2) = Clos\ [X]\ ((X+1)+1)$. Obviously, this structural equality cannot be proved, which means if we used simple behavioural equivalence, we can only prove structurally equal (identical) functions equivalent.

4.2 Contextual Equivalence

Instead of proving the above-mentioned closure equality, we could prove that these closures behave the same way in the same *expression contexts*. That is, we can consider using contextual equivalence instead of the behavioural one. Apparently, to be able to use this equivalence, we need to define expression contexts (type *Context*), which are basically expressions with one of their subexpressions replaced by a *hole*. This hole can be substituted by any expression (e.g. $\texttt{apply}\ f(1,\Box,3)[2] = \texttt{apply}\ f(1,2,3)$) to obtain valid expressions.

To investigate this equivalence, we chose a simple subset of sequential Core Erlang without considering side effect first [6]. Note that we plan to reinject side effects in the near future which would also count towards the equivalence, just as in Section 4.1.

$$
\begin{aligned}
v &::= l \mid x \mid f/k \mid \texttt{fun}(x_1,\ldots,x_k) \to e \mid \texttt{fun}\ f/k(x_1,\ldots,x_k) \to e \\
e &::= v \mid \texttt{apply}\ e(e_1,\ldots,e_k) \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{letrec}\ f/k = \texttt{fun}(x_1,\ldots,x_k) \to e_0\ \texttt{in}\ e \\
&\quad \mid e_1 + e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3
\end{aligned}
$$

This small language supports integer literals, variable names, function identifiers (function name, arity pairs), simple and recursive functions as syntactical values. Compound expressions are function applications, `let` and `letrec` expressions, while we also support simple conditional and addition expressions to be able to write meaningful examples. Now we define expression contexts the following way (note that there is distinct recursive and simple function contexts):

$$
\begin{aligned}
C ::= {} & \Box \mid \texttt{fun}(x_1,\ldots,x_k) \to C \mid \texttt{fun}\ f/k(x_1,\ldots,x_k) \to C \\
& \mid \texttt{apply}\ C(e_1,\ldots,e_k) \mid \texttt{apply}\ e(C,e_2,\ldots,e_k) \mid \cdots \mid \texttt{apply}\ e(e_1,\ldots,e_{k-1},C) \\
& \mid \texttt{let}\ x = C\ \texttt{in}\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ C \\
& \mid \texttt{letrec}\ f/k = \texttt{fun}(x_1,\ldots,x_k) \to C\ \texttt{in}\ e \mid \texttt{letrec}\ f/k = \texttt{fun}(x_1,\ldots,x_k) \to e_0\ \texttt{in}\ C \\
& \mid C + e_2 \mid e_1 + C \mid \texttt{if}\ C\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{if}\ e_1\ \texttt{then}\ C\ \texttt{else}\ e_3 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ C
\end{aligned}
$$

We define the substitution of the $\square$ by an expression in the usual way (see [23] for example), and we will denote it by $C[e]$. Next, we define the contextual equivalence (similarly to Pitts [23]) using contextual preorders:

$$e_1 \equiv_{ctx} e_2 := e_1 \leq_{ctx} e_2 \wedge e_2 \leq_{ctx} e_1$$
$$e_1 \leq_{ctx} e_2 := \forall C, res : (\exists d : eval\ d\ C[e_1] = res) \Rightarrow (\exists d : eval\ d\ C[e_2] = res)$$

While it is straightforward to disprove expressions contextually equivalent, the proof of equivalence is significantly more complex as it requires induction over contexts. Instead of proving expressions contextually equivalent, one can define other equivalence (and preorder) relations which are simpler to prove for concrete expressions, and prove these relations equal to the contextual equivalence.

Moreover, there is another problem with this definition: this relation is equal to syntactical equality (because of $C[e_1]$ and $C[e_2]$ must evaluate to the same value, considering a function context as C, only equal function expression bodies can be evaluated to equal closures), thus we need to define the equivalence of values too. One of the most common ways to overcome these limitations is to define CIU-equivalence and logical relations and prove them equal with the contextual equivalence [23, 29, 24]. We also note that according to Pitts [25] (Appendix B.3) it is sufficient to prove termination of equivalent expressions instead of the equivalence of their observable results (i.e. their values); however, in our case, simple side effects should be explicitly considered at this point, because they do not affect the termination of programs.

4.3 The logical relation

First, we followed the footsteps of Pitts [23], and adapted his "logical simulation relation". Unfortunately, their mathematical definitions cannot be directly formalised in Coq as they use statements that are not strictly positive and therefore do not pass Coq's positivity checker. Neither could we use *types* as others did [7] while formalising such logical relations, because both Core Erlang and Erlang are dynamically typed. These are the reasons why we chose to follow the idea of using step-indexed relations [29, 24]. First, we needed to formalise the termination relation inductively, in frame stack semantics ($\langle Fs, e\rangle \Downarrow^k$ if e terminates in exactly in k steps, and $\langle Fs, e\rangle \Downarrow$ if e terminates in Fs reduction context, for the details we refer to Appendix B and [6]). This semantics has the advantage of evaluating expressions in arbitrary reduction contexts (i.e. frame stacks) over our current functional big-step semantics [19]. Concrete frame stacks will be denoted by standard list notation (i.e. [] denotes the empty list and $[F_1, \ldots, F_n]$ frame stack contains frames $F_1, \ldots, F_n$, while $F :: Fs$ prepends F to the list of frames Fs).

Now we define the step-indexed logical relations on closed values ($\mathbb{V}\_$), closed frame-stacks ($\mathbb{F}\_$), and closed expressions ($\mathbb{E}\_$) for our language (closedness and variable scoping is detailed in Appendix A). In the rest of the paper, $e[v_1|x_1, \ldots, v_k|x_k]$ denotes the substitution of variables (or function identifiers) $x_1, \ldots, x_k$ to $v_1, \ldots, v_k$ in e.

$$(l_1, l_2) \in \mathbb{V}_n \iff l_1 = l_2$$
$$(\texttt{fun}(x_1, \ldots, x_k) \to e, \texttt{fun}(x_1, \ldots, x_k) \to e') \in \mathbb{V}_n \iff \forall m < n : \forall v_1, v_1', \ldots, v_k, v_k' :$$
$$(v_1, v_1') \in \mathbb{V}_m \wedge \cdots \wedge (v_k, v_k') \in \mathbb{V}_m \implies (e[v_1|x_1, \ldots, v_k|x_k], e'[v_1'|x_1, \ldots, v_k'|x_k]) \in \mathbb{E}_m$$
$$(\texttt{fun}\ f/k(x_1, \ldots, x_k) \to e, \texttt{fun}\ f/k(x_1, \ldots, x_k) \to e') \in \mathbb{V}_n \iff$$
$$\forall m < n : \forall v_1, v_1', \ldots, v_k, v_k' : (v_1, v_1') \in \mathbb{V}_m \wedge \ldots \wedge (v_k, v_k') \in \mathbb{V}_m \implies$$
$$(e[\texttt{fun}\ f/k(x_1, \ldots, x_k) \to e|f/k, v_1|x_1, \ldots, v_k|x_k], e'[\texttt{fun}\ f/k(x_1, \ldots, x_k) \to e'|f/k, v_1'|x_1, \ldots, v_k'|x_k]) \in \mathbb{E}_m$$

$$(Fs_1, Fs_2) \in \mathbb{F}_n \iff \forall m \leq n, v_1, v_2 : (v_1, v_2) \in \mathbb{V}_m \implies \langle Fs_1, v_1 \rangle \Downarrow^m \implies \langle Fs_2, v_2 \rangle \Downarrow$$

$$(e_1, e_2) \in \mathbb{E}_n \iff \forall m \leq n, Fs_1, Fs_2 : (Fs_1, Fs_2) \in \mathbb{F}_m \implies \langle Fs_1, e_1 \rangle \Downarrow^m \implies \langle Fs_2, e_2 \rangle \Downarrow$$

Then we extend these relations to any expression (or value) pairs with closing substitutions (i.e. the free variables of the expression are replaced by closed values). A substitution is a function (we will denote it with ξ), which assigns expressions to free variables and function identifiers. Technically, the substitution of variables above is also defined by such a function in the Coq formalisation. The notation $e[\xi]$ means that we apply the substitution ξ to the expression e (i.e. we replace all free variables and function identifiers in e with the corresponding expressions in ξ).

$$\begin{aligned}
(\xi_1, \xi_2) \in \mathbb{G}_n^\Gamma &\iff dom(\xi_1) = dom(\xi_2) = \Gamma \wedge \forall x \in \Gamma : (\xi_1(x), \xi_2(x)) \in \mathbb{V}_n \\
(v_1, v_2) \in \mathbb{V}_\Gamma &\iff \forall n, \xi_1, \xi_2 : (\xi_1, \xi_2) \in \mathbb{G}_n^\Gamma \implies (v_1[\xi_1], v_2[\xi_2]) \in \mathbb{V}_n \\
(e_1, e_2) \in \mathbb{E}_\Gamma &\iff \forall n, \xi_1, \xi_2 : (\xi_1, \xi_2) \in \mathbb{G}_n^\Gamma \implies (e_1[\xi_1], e_2[\xi_2]) \in \mathbb{V}_n
\end{aligned}$$

After having these relations defined, we proceeded to prove the two most important properties of them (just like others did [7, 29, 24]): the "fundamental property" (a form of reflexivity) and the compatibility rules which are forms of congruence. Obviously, to be able to manage the proofs, a number of lemmas were needed (we refer to the formalisation [6] for more details).

Theorem 1 (Fundamental property)

$$\begin{aligned}
\texttt{EXP}\ \Gamma \vdash e &\implies (e, e) \in \mathbb{E}^\Gamma \\
\texttt{VAL}\ \Gamma \vdash v &\implies (v, v) \in \mathbb{V}^\Gamma
\end{aligned}$$

Theorem 2 (Compatibility rules)

$$\frac{x \in \Gamma}{(x,x) \in \mathbb{V}^\Gamma} \qquad \frac{f/k \in \Gamma}{(f/k, f/k) \in \mathbb{V}^\Gamma} \qquad \frac{}{(l,l) \in \mathbb{V}^\Gamma} \qquad \frac{(b_1, b_2) \in \mathbb{E}^{\Gamma \cup \{x_1, \ldots, x_k\}}}{(\texttt{fun}(x_1, \ldots, x_k) \to b_1, \texttt{fun}(x_1, \ldots, x_k) \to b_2) \in \mathbb{V}^\Gamma}$$

$$\frac{(b_1, b_2) \in \mathbb{E}^{\Gamma \cup \{f/k, x_1, \ldots, x_k\}}}{(\texttt{fun}\ f/k(x_1, \ldots, x_k) \to b_1, \texttt{fun}\ f/k(x_1, \ldots, x_k) \to b_2) \in \mathbb{V}^\Gamma} \qquad \frac{(v_1, v_2) \in \mathbb{V}^\Gamma}{(v_1, v_2) \in \mathbb{E}^\Gamma}$$

$$\frac{(e, e') \in \mathbb{E}^\Gamma \quad (e_1, e_1') \in \mathbb{E}^\Gamma \quad \cdots \quad (e_k, e_k') \in \mathbb{E}^\Gamma}{(\texttt{apply}\ e(e_1, \ldots, e_k), \texttt{apply}\ e'(e_1', \ldots, e_k')) \in \mathbb{E}^\Gamma} \qquad \frac{(e_1, e_1') \in \mathbb{E}^\Gamma \quad (e_2, e_2') \in \mathbb{E}^\Gamma}{(e_1 + e_2, e_1' + e_2') \in \mathbb{E}^\Gamma}$$

$$\frac{(e, e') \in \mathbb{E}^{\Gamma \cup \{f/k\}} \quad (b, b') \in \mathbb{E}^{\Gamma \cup \{f/k, x_1, \ldots, x_k\}}}{(\texttt{letrec}\ f/k = \texttt{fun}(x_1, \ldots, x_k) \to b\ \texttt{in}\ e, \texttt{letrec}\ f/k = \texttt{fun}(x_1, \ldots, x_k) \to b'\ \texttt{in}\ e') \in \mathbb{E}^\Gamma}$$

$$\frac{(e_1, e_1') \in \mathbb{E}^\Gamma \quad (e_2, e_2') \in \mathbb{E}^\Gamma \quad (e_3, e_3') \in \mathbb{E}^\Gamma}{(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \texttt{if}\ e_1'\ \texttt{then}\ e_2'\ \texttt{else}\ e_3') \in \mathbb{E}^\Gamma}$$

4.4 CIU equivalence

Alongside proving the properties of the logical relations, we have also formalised CIU ("closed instances of use") preorder and equivalence [26]. Informally, two expressions are CIU-equivalent, when they both

terminate or diverge when placed in arbitrary reduction contexts.

$$e_1 \leq_{ciu} e_2 := \forall Fs : \langle Fs, e_1 \rangle \Downarrow \implies \langle Fs, e_2 \rangle \Downarrow \text{ : for closed expressions}$$
$$e_1 \leq^{\Gamma}_{ciu} e_2 := \forall \xi : e_1[\xi] \leq_{ciu} e_2[\xi] \text{ : for closing substitutions}$$

Practice shows, that usually proving expression CIU-equivalent is simpler than proving them contextually equivalent [26] or related by the logical relations (it requires only one frame stack and one substitution, rather than related pairs) [29]. After defining the CIU-preorder, we also proved its correspondence with the logical relations (see [6]):

Theorem 3 $e_1 \leq^{\Gamma}_{ciu} e_2 \iff (e_1, e_2) \in \mathbb{E}^{\Gamma}$

4.5 Revisiting contextual preorder and equivalence

We formalised contextual preorder (following the idea of Wand et al. [29]) as the largest family of relations R^{Γ}, such that it is adequate (if $(e_1, e_2) \in R^{\emptyset}$ then $\langle [], e_1 \rangle \Downarrow \implies \langle [], e_2 \rangle \Downarrow$), transitive and reflexive (for every Γ), moreover satisfies the compatibility rules for expressions (for every Γ, see Thm. 2 which is specialised for the logical relations).

We also adjusted our previous notion of contextual preorder and equivalence, and proved that it satisfies the criteria above.

$$e_1 \leq^{\Gamma}_{ctx} e_2 := \texttt{EXP}\ \Gamma \vdash e_1 \wedge \texttt{EXP}\ \Gamma \vdash e_2 \wedge$$
$$\forall (C : \mathit{Context}) : \texttt{EXP}\ [] \vdash C[e_1] \wedge \texttt{EXP}\ [] \vdash C[e_2] \implies \langle [], C[e_1] \rangle \Downarrow \implies \langle [], C[e_2] \rangle \Downarrow$$
$$e_1 \equiv^{\Gamma}_{ctx} e_2 := e_1 \leq^{\Gamma}_{ctx} e_2 \wedge e_2 \leq^{\Gamma}_{ctx} e_1$$

After defining the contextual preorder, we started to investigate the correspondence between $\leq^{\Gamma}_{ciu}$ and $\leq^{\Gamma}_{ctx}$. Currently, we managed to prove one direction from the equality of $\leq^{\Gamma}_{ciu}$ and $\leq^{\Gamma}_{ctx}$ (Thm. 4), and we are working on proving the opposite one.

Theorem 4 $e_1 \leq^{\Gamma}_{ciu} e_2 \implies e_1 \leq^{\Gamma}_{ctx} e_2$, *moreover,* $(e_1, e_2) \in \mathbb{E}^{\Gamma} \implies e_1 \leq^{\Gamma}_{ctx} e_2$

5 Conclusion and Future Work

In this extended abstract, we described our idea of verifying compound refactorings via decomposition to local transformations. To reason about their correctness, we need a suitable program equivalence definition. We investigated and formalised simple behavioural equivalence [22] in Coq. This equivalence proved to be useful (we proved simple program patterns equivalent in our related work [1]), but not expressive enough in case of proving function expressions equivalent (only structurally equal functions can be proved equivalent).

To solve this issue, we formalised contextual, CIU preorder and equivalence alongside with logical relations (following other authors [7, 23, 24, 29]). With these equivalences, we are able to prove functions equivalent, which are not necessarily structurally equal. We presented our preliminary results (which are also formalised in Coq [6]), which contain the equality of CIU equivalence and logical relations alongside a number of additional theorems, and described our current work on finishing the proofs for the equality of all the three relations. Moreover, currently we are also working on proving program patterns equivalent.

In the short term, we are planning on defining these equivalence relations for our entire formalisation of Core Erlang [19], including side effects. In the medium and longer term, we plan to extend this formalisation with concurrent language features and also formalise Erlang in full in Coq. Our longer term goals also include the investigation of bisimulation relations for program equivalence, covering *inter alia* formalised concurrent language features.

Acknowledgements

The project has been supported by ÚNKP-20-4 New National Excellence Program of the Ministry for Innovation and Technology and "Application Domain Specific Highly Reliable IT Solutions" financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme by the National Research, Development and Innovation Fund of Hungary and "Integrated program for training new generation of researchers in the disciplinary fields of computer science", No. EFOP-3.6.3-VEKOP-16-2017-00002 by the European Union and by the European Social Fund.

References

[1] Péter Bereczky, Dániel Horpácsi & Simon Thompson (2020): *A Proof Assistant Based Formalisation of a Subset of Sequential Core Erlang*. In Aleksander Byrski & John Hughes, editors: *Trends in Functional Programming*, Springer International Publishing, Cham, pp. 139–158, doi:10.1007/978-3-030-57761-2_7.

[2] Péter Bereczky, Dániel Horpácsi & Simon J. Thompson (2020): *Machine-Checked Natural Semantics for Core Erlang: Exceptions and Side Effects*. In: *Proceedings of Erlang 2020*, ACM, p. 1–13, doi:10.1145/3406085.3409008.

[3] Péter Bereczky, Dániel Horpácsi, Judit Kőszegi, Soma Szeier & Simon Thompson (2021): *Validating Formal Semantics by Property-Based Cross-Testing*. ACM, New York, NY, USA, doi:10.1145/3462172.3462200. To appear.

[4] Péter Bereczky, Dániel Horpácsi & Simon Thompson (2020): *A Comparison of Big-step Semantics Definition Styles*. Available at https://arxiv.org/abs/2011.10373.

[5] Ştefan Ciobâcă, Dorel Lucanu, Vlad Rusu & Grigore Roşu (2014): *A Language-Independent Proof System for Mutual Program Equivalence*. In Stephan Merz & Jun Pang, editors: *Formal Methods and Software Engineering*, Springer International Publishing, Cham, pp. 75–90.

[6] (2021): *Core Erlang Mini*. Available at https://github.com/harp-project/Core-Erlang-mini/tree/nameless-names.

[7] Ryan Culpepper & Andrew Cobb (2017): *Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring*. In Hongseok Yang, editor: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 368–392.

[8] Martin Fowler (1999. ISBN: 0201485672): *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., USA.

[9] Lars-Åke Fredlund (2001): *A framework for reasoning about Erlang code*. Ph.D. thesis, Mikroelektronik och informationsteknik.

[10] Kofi Gumbs (2017): *The Core of Erlang*. Available at https://8thlight.com/blog/kofi-gumbs/2017/05/02/core-erlang.html.

[11] Dániel Horpácsi, Judit Kőszegi & Zoltán Horváth (2017): *Trustworthy Refactoring via Decomposition and Schemes: A Complex Case Study*. doi:10.4204/EPTCS.253.8.

[12] Dániel Horpácsi, Judit Kőszegi & Simon Thompson (2016): *Towards Trustworthy Refactoring in Erlang*. doi:10.4204/EPTCS.216.5.

[13] B. Jonsson & J. Parrow (1993): *Deciding Bisimulation Equivalences for a Class of Non-Finite-State Programs*. Information and Computation 107(2), pp. 272–302, doi:https://doi.org/10.1006/inco.1993.1069.

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

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

[16] Dorel Lucanu & Vlad Rusu (2013): *Program Equivalence by Circular Reasoning*. In Einar Broch Johnsen & Luigia Petre, editors: Integrated Formal Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 362–377.

[17] (2021): *Matching logic formalisation in Coq*. Available at https://github.com/harp-project/AML-Formalization.

[18] Robin Milner (1983): *Calculi for synchrony and asynchrony*. Theoretical Computer Science 25(3), pp. 267–310, doi:https://doi.org/10.1016/0304-3975(83)90114-7.

[19] (2021): *Core Erlang Formalization*. Available at https://github.com/harp-project/Core-Erlang-Formalization.

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

[21] Scott Owens, Magnus O. Myreen, Ramana Kumar & Yong Kiam Tan (2016): *Functional Big-Step Semantics*. In Peter Thiemann, editor: Programming Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 589–615, doi:10.1007/978-3-662-49498-1_23.

[22] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg & Brent Yorgey (2010): *Software foundations*. Available at https://softwarefoundations.cis.upenn.edu/.

[23] Andrew M. Pitts (2002): *Operational Semantics and Program Equivalence*. In Gilles Barthe, Peter Dybjer, Luís Pinto & João Saraiva, editors: Applied Semantics, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 378–412.

[24] Andrew M. Pitts (2010): *Step-Indexed Biorthogonality: a Tutorial Example*. In Amal Ahmed, Nick Benton, Lars Birkedal & Martin Hofmann, editors: Modelling, Controlling and Reasoning About State, Dagstuhl Seminar Proceedings 10351, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany. Available at http://drops.dagstuhl.de/opus/volltexte/2010/2806.

[25] Andrew M Pitts & Ian DB Stark (1998): *Operational reasoning for functions with local state*. Higher order operational techniques in semantics, pp. 227–273.

[26] R Ramanujam & V Arvind (1998): *Foundations of Software Technology and Theoretical Computer Science*. Springer.

[27] Alex Simpson & Niels Voorneveld (2019): *Behavioural Equivalence via Modalities for Algebraic Effects*. ACM Trans. Program. Lang. Syst. 42(1), doi:10.1145/3363518.

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

[29] Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos & Andrew Cobb (2018): *Contextual Equivalence for a Probabilistic Language with Continuous Random Variables and Recursion*. Proc. ACM Program. Lang. 2(ICFP), doi:10.1145/3236782.

A Scoping

We also introduce the concept of scoping for our language, following the footsteps of Wand et al. [29], to be able to distinguish "closed" expressions (i.e. that do not contain free variables). We use $\texttt{EXP}\ \Gamma \vdash e$ (and $\texttt{VAL}\ \Gamma \vdash v$) to denote such e expressions (and values) whose free variables (and function identifiers) are contained in Γ. The scoping rules are given in Figure 1.

$$\frac{}{\texttt{VAL}\ \Gamma \vdash l} \qquad \frac{x \in \Gamma}{\texttt{VAL}\ \Gamma \vdash x} \qquad \frac{f/k \in \Gamma}{\texttt{VAL}\ \Gamma \vdash f/k} \qquad \frac{\texttt{EXP}\ \Gamma \cup \{x_1,\ldots,x_k\} \vdash e}{\texttt{VAL}\ \Gamma \vdash \texttt{fun}(x_1,\ldots,x_k) \to e} \qquad \frac{\texttt{EXP}\ \Gamma \cup \{f/k,x_1,\ldots,x_k\} \vdash e}{\texttt{VAL}\ \Gamma \vdash \texttt{fun}\ f/k(x_1,\ldots,x_k) \to e}$$

$$\frac{\texttt{VAL}\ \Gamma \vdash e}{\texttt{EXP}\ \Gamma \vdash e} \qquad \frac{\texttt{EXP}\ \Gamma \vdash e \quad \texttt{EXP}\ \Gamma \vdash e_1 \quad \cdots \quad \texttt{EXP}\ \Gamma \vdash e_k}{\texttt{EXP}\ \Gamma \vdash \texttt{apply}\ e(e_1,\ldots,e_k)} \qquad \frac{\texttt{EXP}\ \Gamma \vdash e_1 \quad \texttt{EXP}\ \Gamma \cup \{x\} \vdash e_2}{\texttt{EXP}\ \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2}$$

$$\frac{\texttt{EXP}\ \Gamma \cup \{f/k,x_1,\ldots,x_k\} \vdash e_0 \quad \texttt{EXP}\ \Gamma \cup \{f/k\} \vdash e}{\texttt{EXP}\ \Gamma \vdash \texttt{letrec}\ f/k = \texttt{fun}(x_1,\ldots,x_k) \to e_0 \texttt{in}\ e} \qquad \frac{\texttt{EXP}\ \Gamma \vdash e_1 \quad \texttt{EXP}\ \Gamma \vdash e_2}{\texttt{EXP}\ \Gamma \vdash e_1 + e_2}$$

$$\frac{\texttt{EXP}\ \Gamma \vdash e_1 \quad \texttt{EXP}\ \Gamma \vdash e_2 \quad \texttt{EXP}\ \Gamma \vdash e_3}{\texttt{EXP}\ \Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3}$$

Figure 1: Scoping rules

Based on scoping, a number of useful theorems can be proven about the semantics, and expression substitution. Substitutions can also be scoped too (based on the idea of Wand et al. [29]). The following definition states, that every value associated with a variable (or function identifier) in Γ is scoped in Δ:

$$\texttt{SUB}\ \Gamma \vdash \xi :: \Delta := \forall x, x \in \Gamma \Rightarrow \texttt{VAL}\ \Delta \vdash x$$

We highlight some theorems that have been proven in Coq [6] about the connection between substitutions and scoping.

Theorem 5 (Closed expressions are not modified by substitutions) $\forall e, \xi : \texttt{VAL}\ [] \vdash e \Rightarrow e[\xi] = e$

Theorem 6 (Substitution preserves scoping) $\forall e, \Gamma : \texttt{EXP}\ \Gamma \vdash e \Rightarrow \forall \Delta, \xi : \texttt{SUB}\ \Gamma \vdash \xi :: \Delta \Rightarrow \texttt{EXP}\ \Delta \vdash e[\xi]$

Theorem 7 (Substitution implies scoping) $\forall e, \Gamma, \Delta : (\forall \xi, \texttt{SUB}\ \Gamma \vdash \xi :: \Delta \Rightarrow \texttt{EXP}\ \Delta \vdash e[\xi]) \Rightarrow \texttt{EXP}\ \Gamma \vdash e$

B Frame stack-style Termination Relation

The syntax of frames and frame stacks is the following.

$$\begin{aligned} F ::= {} & \texttt{apply}\ \Box(e_1,\ldots,e_k) \mid \texttt{apply}\ v_0(\Box,e_2,\ldots,e_k) \mid \texttt{apply}\ v_0(v_1,\Box,e_3,\ldots,e_k) \mid \\ & \texttt{apply}\ v_0(v_1,\ldots,v_{k-1},\Box) \mid \texttt{let}\ x = \Box\ \texttt{in}\ e_2 \mid \Box + e_2 \mid v_1 + \Box \mid \texttt{if}\ \Box\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ Fs ::= {} & [F_1,\ldots,F_n] \end{aligned}$$

In Figure 2 we show the inductive, step-indexed termination relation.

$$\frac{}{\langle [], v\rangle \Downarrow^0} \qquad \frac{\langle \texttt{if}\ \Box\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: Fs, e\rangle \Downarrow^n}{\langle Fs, \texttt{if}\ e\ \texttt{then}\ e_2\ \texttt{else}\ e_3\rangle \Downarrow^{1+n}} \qquad \frac{\langle \Box + e_2 :: Fs, e\rangle \Downarrow^k}{\langle Fs, e + e_2\rangle \Downarrow^{1+n}} \qquad \frac{\langle \texttt{apply}\ \Box(e_1,\ldots,e_k) :: Fs, e\rangle \Downarrow^n}{\langle Fs, \texttt{apply}\ e(e_1,\ldots,e_k)\rangle \Downarrow^{1+n}}$$

$$\frac{\langle \texttt{let}\ x = \Box\ \texttt{in}\ e_2 :: Fs, e\rangle \Downarrow^n}{\langle Fs, \texttt{let}\ x = e\ \texttt{in}\ e_2\rangle \Downarrow^{1+n}} \qquad \frac{\langle Fs, e_2\rangle \Downarrow^n}{\langle \texttt{if}\ \Box\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: Fs, 0\rangle \Downarrow^{1+n}}$$

$$\frac{\langle Fs, e_3\rangle \Downarrow^n \qquad v \neq 0}{\langle \texttt{if}\ \Box\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: Fs, v\rangle \Downarrow^{1+n}}$$

$$\frac{\langle v + \Box :: Fs, e_2\rangle \Downarrow^n}{\langle \Box + e_2 :: Fs, v\rangle \Downarrow^{1+n}} \qquad \frac{\langle Fs, l_1 + l_2\rangle \Downarrow^n}{\langle l_1 + \Box :: Fs, l_2\rangle \Downarrow^{1+n}} \qquad \frac{\langle Fs, e_2[v|x]\rangle \Downarrow^n}{\langle \texttt{let}\ x = \Box\ \texttt{in}\ e_2 :: Fs, v\rangle \Downarrow^{1+n}}$$

$$\frac{\langle Fs, e[\texttt{fun}\ f/k(x_1,\ldots,x_k) \to b \mid f/k]\rangle \Downarrow^n}{\langle Fs, \texttt{letrec}\ f/k = \texttt{fun}(x_1,\ldots,x_k) \to b\ \texttt{in}\ e\rangle \Downarrow^{1+n}} \qquad \frac{\langle \texttt{apply}\ v(\Box, e_2,\ldots,e_k) :: Fs, e_1\rangle \Downarrow^n}{\langle \texttt{apply}\ \Box(e_1,\ldots,e_k) :: Fs, v\rangle \Downarrow^{1+n}}$$

For empty parameter list, the following two rules need to be introduced:

$$\frac{\langle Fs, b\rangle \Downarrow^n}{\langle \texttt{apply}\ \Box() :: Fs, \texttt{fun}() \to b\rangle \Downarrow^{1+n}} \qquad \frac{\langle Fs, b[\texttt{fun}\ f/0() \to b|f/0\rangle \Downarrow^n}{\langle \texttt{apply}\ \Box() :: Fs, \texttt{fun}\ f/0() \to b\rangle \Downarrow^{1+n}}$$

$$\frac{\langle \texttt{apply}\ v(v_1,\ldots,v_i,\Box,e_{i+2},\ldots,e_k) :: Fs, e_{i+1}\rangle \Downarrow^n}{\langle \texttt{apply}\ v(v_1,\ldots,v_{i-1},\Box,e_{i+1},\ldots e_k) :: Fs, v_i\rangle \Downarrow^{1+n}}$$

$$\frac{\langle Fs, b[v_1|x_1,\ldots,v_k|x_k]\rangle \Downarrow^n}{\langle \texttt{apply}\ (\texttt{fun}(x_1,\ldots,x_k) \to b)(v_1,\ldots,v_{k-1},\Box) :: Fs, v_k\rangle \Downarrow^{1+n}}$$

$$\frac{\langle Fs, b[\texttt{fun}\ f/k(x_1,\ldots,x_k) \to b|f/k, v_1|x_1,\ldots,v_k|x_k]\rangle \Downarrow^n}{\langle \texttt{apply}\ (\texttt{fun}\ f/k(x_1,\ldots,x_k) \to b)(v_1,\ldots,v_{k-1},\Box) :: Fs, v_k\rangle \Downarrow^{1+n}}$$

Figure 2: Step-indexed, frame stack-style termination relation