Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp

Program equivalence in an untyped, call-by-value functional language with uncurried functions

Dániel Horpácsi [a,*], Péter Bereczky [a,*], Simon Thompson [a,b]

[a] ELTE, Eötvös Loránd University, Hungary
[b] University of Kent, United Kingdom

A R T I C L E I N F O

Article history:
Received 15 November 2021
Received in revised form 18 October 2022
Accepted 3 February 2023
Available online 9 February 2023

Keywords:
Contextual equivalence
Program equivalence
Logical relation
CIU theorem
Erlang
Coq

A B S T R A C T

We aim to reason about the correctness of behaviour-preserving transformations of Erlang programs. Behaviour preservation is characterised by semantic equivalence. Based upon our existing formal semantics for Core Erlang, we investigate potential definitions of suitable equivalence relations. In particular we adapt a number of existing approaches of expression equivalence to a simple functional programming language that carries the main features of sequential Core Erlang; we then examine the properties of the equivalence relations and formally establish connections between them. The results presented in this paper, including all theorems and their proofs, have been machine checked using the Coq proof assistant.

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, refactoring tools are expected not to change the behaviour of any program, but refactoring processes in most of the current tools are validated by testing only. 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 behaviour preservation. This requires a rigorous formal definition of the programs being refactored, a precise description of the effect of refactorings on these programs, and a suitable definition of program equivalence.

1.1. Refactoring and program equivalence

A program transformation is said to be a refactoring if it preserves the observable behaviour, while improving the internal structure [1]. The verification of whether a transformation is refactoring can be based upon the concept of (semantic) program equivalence: the internal workings of the software may be altered, but the result has to be observationally indistinguishable from the original in an arbitrary environment.

The typical refactoring process – reworking a piece of code to increase its quality – can be decomposed into multiple, smaller refactoring steps (known as *micro-refactorings*). These smaller steps may vary from eliminating unused variables to extracting code portions to function abstractions. Some are local to particular portions of code – such as an expression,

* Corresponding authors.
E-mail addresses: daniel-h@elte.hu (D. Horpácsi), berpeti@inf.elte.hu (P. Bereczky), s.j.thompson@kent.ac.uk (S. Thompson).

https://doi.org/10.1016/j.jlamp.2023.100857

```
apply 'map'/2(fun(X) -> call '+'(X,1), Xs) ≡
  apply 'foldr'/3(fun(X, A) -> [call '+'(X,1) | A], [], Xs)
```

Fig. 1. An equivalence formula in Core Erlang which can be seen as a simple refactoring.

a function or a module – but others make modifications across a code base, which seems to require the entire software project to be examined for equivalence when reasoning about the correctness of a complex refactoring.

A key observation in our previous work [2,3] is that, when expressed with a well-designed set of skeletons, the verification of both local and non-local refactoring transformations can be reduced to checking the equivalence of simple expressions. Thus, the work presented here aims to provide a basis for reasoning about refactorings by establishing the appropriate semantics and definitions of program equivalence for languages like Erlang.

Although our ultimate goal is to prove Erlang refactorings correct, 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. For readers unfamiliar with Erlang, we note that Core Erlang contains all the essential features of the full language, so that the work presented here can be extended to the full language in a routine way.

1.2. Running example

Let us show a motivating example informally, which we are going to revisit formally towards the end of the paper. In functional programming, it is a fairly well-known fact that the higher-order function *map* can be expressed in terms of the function *foldr* (also higher-order). Let us consider a special case where the function mapped over the list is the incrementation function. In a functional pseudo-language, we would express this as an equivalence as follows:

$$map\ (+1)\ xs \equiv foldr\ (\lambda x.\lambda a.[x+1|a])\ []\ xs$$

Somewhat less readable, Fig. 1 shows the formulation of the corresponding equivalence formula in Core Erlang. As mentioned already, such equivalences can be seen as proof obligations of refactoring steps. In particular, rewriting the former expression to the latter is a correct refactoring step if the two expressions are (contextually) equivalent; in the rest of the paper, we will prove that this is indeed the case when using the standard definitions of *map* and *foldr* (see Section 3.2).

1.3. Scope and contributions

Core Erlang is the intermediate language for a number of other programming languages, including Erlang, Elixir and LFE, meaning that program equivalence in the core language can be used for checking equivalence in these other languages, assuming the existence of a trusted translation from the source language to Core Erlang.

We have already developed natural and functional big-step semantics for sequential Core Erlang [4–6] in the Coq proof assistant and defined simple behavioural equivalence. In this paper we distil the essential features of sequential Core Erlang into an extended lambda calculus, and systematically define and compare various equivalence relations on that system. In contrast with related work with similar goals, the novelty of our approach lies with the Erlang-like features (call-by-value evaluation, pattern matching, and uncurried functions) of the programming language, and full, machine-checked formalisation.

The main contributions of the paper are:

- A (frame stack style) formal semantics for an extended, untyped, strict lambda calculus with uncurried functions, which is essentially a subset of sequential Core Erlang.
- A formalisation of a number of termination-based equivalence concepts (namely, contextual equivalence, logical relations, and "closed instances of use" equivalence), and a proof of their coincidence for this language.
- A number of expression equivalence proofs, including the equivalence of the higher-order functions (`map`, `foldr`) as shown in Fig. 1.
- A characterisation of behavioural equivalence that is not solely based on termination, and which is also proved to coincide with the rest of the equivalence definitions.
- A machine-checked formalisation of the language, the definitions and the proofs in the Coq proof assistant.

The rest of the paper is structured as follows. In Section 2 we overview related work on program equivalence for similar languages and point to some influential results that we reused in our work. In Section 3 we formally define the syntax and semantics of the language we investigate, then in Section 4 we specify various equivalence definitions for Core Erlang, including simple behavioural equivalence [7]. Next in Section 5 we make these definitions more accurate using an approach based on logical relations [8,9], pointing out their advantages and disadvantages. Finally, Section 6 concludes.

2. Related work

In the early stages of our project, we have developed an inductive big-step semantics for sequential Core Erlang [4,5] 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 [10–12], a framework for reasoning about Erlang [13] and symbolic execution [14]. We have also investigated different big-step definition styles [15], and recently we also implemented an equivalent functional big-step semantics [16] which enabled extensive validation [6].

However, for proving refactoring correctness, we need accurate definitions of program equivalence. We have proved some simple expressions equivalent in our formalisation, however, the used equivalence concept is rather an ad hoc definition of behavioural equivalence, and it has not been suitable to reason about equivalence of function expressions. This motivates the investigation of (other) approaches.

The literature discusses a number of ways to describe and investigate equivalence between programs of a language after having its semantics formally defined. In this section, we provide a brief overview of related results.

The simplest notion of program equivalence is behavioural equivalence [7], which requires syntactical equality of the program evaluation results. This notion is a rather strict characterisation of the equivalence concept, but may prove sufficient in some simple cases. Clearly, its main advantage is the simple definition lacking any reference to expression contexts.

Another fundamental equivalence definition is the *contextual* (also called *observational*) equivalence, which characterises the congruence property of the equivalence relation. Two programs are contextually equivalent if they are indistinguishable in arbitrary expression contexts observing their behaviour. In most cases, the observed behaviour is simply the termination property of the programs (in all syntactical contexts): mutual termination is sufficient to ensure the result values to be equivalent by a suitable value equivalence relation; Pitts provides a draft proof of this in [17], and we give a formal proof in Section 5.6 with our formalisation. The idea behind the proof is that supposing that two programs show the same termination behaviour in every syntactical context, and they are evaluated to non-equivalent results, then we can show a context, in which one of the programs terminate while the other does not, contradicting our assumptions. In general, while it is straightforward to disprove programs equivalent with this notion by giving such a counterexample context, establishing contextual equivalence requires induction on all expression contexts. To avoid this burdensome induction, related work proposes alternative definitions of the relation, such as bisimulations, CIU ("closed instances of uses") equivalence and logical relations.

Bisimulations (or applicative bisimulations [18]) are binary relations between expressions based on their reduction being in the same relation. Bisimulation approaches [17,19] are naturally proved to be sound for contextual equivalence, ensuring that bisimilar programs are equivalent (using Howe's method for proving congruence [20]); therefore, bisimulations can be used to prove expressions equivalent. However, completeness does not necessarily hold as bisimulations sometimes provide a finer approach (defining a stricter equivalence), distinguishing some terms that are contextually equivalent [21].

The other widespread approach to establishing contextual equivalence is using alternative equivalences (CIU equivalence, logical relations) and proving that these relations coincide with contextual equivalence. The idea of the CIU equivalence originates from Mason and Talcott [22], but since then other authors investigated and used this characterisation with success for imperative languages [23], for variants of lambda calculus [9,24,25], lambda calculus with recursive types [26,27], and with quantified types [26]. Some authors also included effects in this characterisation [22,28]. Most of these results used the same idea: they defined a form of continuation-style semantics (such as reduction semantics, frame stack semantics) or termination relation, then defined CIU and contextual equivalence and proved that they coincide. The novelty of the various results lie in the choice of the type system and the language constructs under investigation.

Other authors [8,9,25,26,29] also defined logical relations and proved that they also coincide with the previous equivalences. In case of the logical relations, there was two main approaches: type-indexed [8] or step- (and type)-indexed [9,25,26,29].

There are further approaches to prove program equivalence, for example using algorithms; however, for these to work we need either an operational semantics based on term rewriting [30], or reasoning in some dedicated logics like matching logic [31].

In the remainder of this paper, we investigate step-indexed logical relations, CIU equivalence, and contextual equivalence for a simple untyped functional language based on the previous results of the above-mentioned authors. We remind the reader that our definitions and theorems are fully formalised in the Coq proof assistant [32] (similarly to the formalisation by Wand et al. in Coq [9], or the formalisation by McLaughlin et al. in Agda [24]).

3. An untyped, strict, uncurried functional language with recursion and pattern matching

In this section, we present the syntax and semantics of a functional language that is a subset of Core Erlang. In later sections we define equivalence relations over expressions of this language.

3.1. Syntax

We start with the formal definition of the syntax. We use i to range over integers, x over program identifiers, k, n to denote natural numbers, and a, f are used for atoms. Although not complete, the language we present here is a representative subset of Core Erlang. It is future work to formalise the omitted language constructs too (such as tuples, pattern matching with more than two branches, value lists, value sequences), but the equivalence concepts presented in this paper are expected to be extensible to these expressions in an analogous way.

Definition 3.1 *(Syntax of the language).*

$$p \in Pattern ::= i \mid a \mid x \mid \texttt{[}p_1 \mid p_2\texttt{]} \mid \texttt{[]}$$

$$\begin{aligned} e, v \in Exp ::= {} & i \mid a \mid x \mid f/k \mid \texttt{fun}\ f/k(x_1, \dots, x_k) \to e \mid \texttt{[}e_1 \mid e_2\texttt{]} \mid \texttt{[]} \\ & \mid \texttt{[}e_1 \mid e_2\texttt{]}^{\,v} \mid \texttt{apply}\ e(e_1, \dots, e_k) \mid \texttt{case}\ e_1\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ & \mid \texttt{rec}\ f/k = \texttt{fun}(x_1, \dots, x_k) \to e_0\ \texttt{in}\ e \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\ & \mid \texttt{call}\ e(e_1, \dots, e_k) \end{aligned}$$

The language we investigate has integers (denoted by numbers), atoms (enclosed in single quotation marks), lists and variables as patterns. The expressions include atoms, integers, variables, function identifiers, lists, uncurried function abstraction and application, pattern matching with `case`, `let`-binding, explicitly recursive function abstraction with `rec`, and built-in function (BIF) call with `call`. We note that having uncurried function abstraction and application poses a number of challenges in the formalisation, e.g. mutual induction between lists and expressions in proofs, expressing the evaluation order of parameters in the semantics.

Let binding Binding expressions are `let` and `rec`. With $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, the variable x (which can be used in e_2) is bound to the expression e_1, while in $\texttt{rec}\ f/k = \texttt{fun}(x_1, \dots, x_k) \to e_0\ \texttt{in}\ e$ the function identifier f/k (which can be used both in e_0 and e) is bound to a (potentially recursive) function expression.

Names Both variable and function identifiers are called *names*. For simplicity, in the machine-checked formalisation [32] we use a nameless variable representation, that is, names are de Bruijn indices [33]. This way, in the body of a binder, the outermost indices denote the bound variables, i.e. for functions, the 0th index points to the identifier of the function, while the next k indices denote the formal parameters. One could write non-recursive functions by simply omitting the use of the 0th index. For readability, in the paper we present our results with explicit names for readability; however, we implicitly regard alpha equivalent expressions, patterns and values as equal.

Pattern matching Note that unlike the majority of the cited related work, we included pattern matching, which allows us to observe the differences between values and implement conditionals in the object theory. In the `case` expression, if e_1 matches p, then e_2 will be evaluated, if not, then e_3 will be evaluated. Without pattern matching (or similar language features that allow for inspecting and comparing the contents of values) we cannot prove that contextual equivalence (which coincides with CIU equivalence) and behavioural equivalence coincide (Theorem 5.18). We use the following three auxiliary functions for pattern matching (we omit the formal definitions):

- $vars(p)$ is the list of variables used in p;
- $is\_match(p, v)$ decides whether the value v matches the pattern p;
- $match(p, v)$ yields the substitution (Section 3.3) that matches p with v.

Built-in function (BIF) calls The BIF call expressions are used to express function calls between modules in (Core) Erlang (e.g. calling to standard library functions). Since we do not include the formalisation of the module system, we simulate the behaviour of these functions by interpreting them directly. In our formalisation, we chose the addition of integers as a representative example, but other operations can be implemented analogously.[1]

Expressions and values On the level of the syntax, we do not differentiate between values and expressions. Their separation is expressed by the scoping judgement (Section 3.4). There is one exception; we distinguished two constructs for lists (namely $\texttt{[}e_1 \mid e_2\texttt{]}$ and $\texttt{[}e_1 \mid e_2\texttt{]}^{\,v}$), one for expression lists and one for value lists. This distinction is necessary to preserve the determinism of the semantics, and implement the reduction rules for lists more accurately without negative preconditions (we refer to Section 3.5.1 for more details). For long lists we use the notation of $\texttt{[}e_1, e_2, \dots, e_k\texttt{]}$ ($\texttt{[}v_1, v_2, \dots, v_k\texttt{]}^{\,v}$ respectively for lists of values) with the meaning of $\texttt{[}e_1 \mid \texttt{[}e_2 \mid \dots \texttt{[}e_k \mid \texttt{[]]} \dots \texttt{]]}$.

It is also to be highlighted that, especially in contrast to the work of Wand et al. [9], our language does not require parts of compound expressions to be in normal form (i.e. reduced to value by explicit sequencing with `let` expressions). This poses some challenges in the formalisation, namely subexpressions of all expressions need to be evaluated by the semantics explicitly. However, this also keeps the consistency between the formalised language syntax and the syntax of Core Erlang.

[1] Furthermore, we use the BIF construct to express the meaning of concurrent features, but those are not in the scope of this paper.

```
rec 'map'/2 = fun(F, L) ->
  case L of  [H | T]
  then [apply F(H) | apply 'map'/2(F, T)]
  else []
in apply 'map'/2(e_f, e_l)
```

(a) A mapping function and its application

```
rec 'foldr'/3 = fun(F, D, L) ->
  case L of  [H | T]
  then apply F(H, apply 'foldr'/3(F, D, T))
  else D
in apply 'foldr'(e_f, e_d, e_l)
```

(b) A folding function and its application

Fig. 2. Example expressions.

3.2. Running examples

Next, we show two Core Erlang expressions as running examples for the next sections. Both (Fig. 2a, Fig. 2b) examples are higher-order functions with their application. To present these examples in a general way, we use the meta-variables e_f, e_l, e_d (which can be replaced by a function expression, a list expression, and any expression, respectively) to denote the inputs of the functions in `rec`. The function `'map'/2` transforms the elements of the list e_l based on the parameter function e_f. On the other hand, `'foldr'/3` aggregates the list e_l, based on e_f. In Section 5, we show the conditions for the equivalence of the expressions in Figs. 2a and 2b.

3.3. Substitution

Before giving semantics to expressions, we develop some metatheory. In particular, we define capture-avoiding, parallel substitutions as functions mapping names to expressions as usual. We refer to substitutions with σ and define the following notations (the formal definitions are present in the formalization [32]):

- We use *id* to denote the identity substitution.
- $\sigma(x)$ is the expression associated with the name x in the substitution σ.
- We use $\sigma\{x_1 \mapsto e_1, \dots, x_k \mapsto e_k\}$ to denote an updated substitution that maps the names $x_1, x_2, \dots, x_k$ to the given expressions $e_1, e_2, \dots, e_k$. Every other name x is mapped to $\sigma(x)$.
- $\sigma \setminus \{x_1, \dots, x_k\}$ stands for a restricted substitution of σ, which is obtained by the removal of the bindings for $x_1, x_2, \dots, x_k$ from the substitution σ.
- The application of the substitution σ to the expression e is denoted by $e[\sigma]$. This operation replaces all the free variables and function identifiers of e with expressions they are associated with in σ.
- $e[x_1 \mapsto e_1, \dots, x_k \mapsto e_k]$ abbreviates $e[id\{x_1 \mapsto e_1, \dots, x_k \mapsto e_k\}]$.

Partial substitutions The Coq proof assistant requires functions to be total; thus, in the implementation, the substitutions were implemented as total functions on the set of names. With this approach, names that should be not affected by the substitution are mapped to themselves. This approach comes with a number of lemmas and theorems to argue about substitution updates, but it is still more favourable in a proof assistant than using other formalisations of (sub)sets for the domain of the substitutions, which would potentially require proof carrying in parameters and loss of computability of the substitution.

3.4. Variable scoping judgement

Note that the small language we work with is untyped, we cannot define a typing judgement and index equivalence relations with types; instead, we define a variable scoping judgement, following the techniques of Wand et al. [9]. The scoping judgement is used to express the set of free variables used by an expression. We separate the values of the language from the expressions with this definition. It also allows us to define our equivalence concepts in a scope-indexed way, where expressions can only be equivalent if they have the same scope.

Definition 3.2 *(Name scoping of values and expressions).* We mutually define two scoping judgements in Fig. 3, one for values and one for expressions: $\Gamma \vdash_{val} v$ holds if all free variables of v are in Γ, and $\Gamma \vdash_{exp} e$ holds if all free variables of e are in Γ. We say that v is a *closed value* if $\emptyset \vdash_{val} v$; similarly, e is a *closed expression* if $\emptyset \vdash_{exp} e$. An expression v is a value, if it satisfies $\Gamma \vdash_{val} v$ for some Γ. We call Γ the scope of the expression (or value) in the judgement.

With the scoping judgement, we can derive that the expressions in Fig. 2a and 2b are closed, supposing that the e_f, e_l and e_d are closed.

The scoping judgement can be generalised to substitutions. This relation characterises that the substitution σ maps the names in Γ to *values* that are scoped in Δ.

Definition 3.3 *(Substitution scoping).* A substitution maps the names in Γ to Δ (denoted $\Gamma \vdash_{sub} \sigma \multimap \Delta$) if $\forall x \in \Gamma : \Delta \vdash_{val} \sigma(x)$.

$$\frac{}{\Gamma \vdash_{val} l} \qquad \frac{x \in \Gamma}{\Gamma \vdash_{val} x} \qquad \frac{f/k \in \Gamma}{\Gamma \vdash_{val} f/k} \qquad \frac{\Gamma \cup \{f/k, x_1, \dots, x_k\} \vdash_{exp} e}{\Gamma \vdash_{val} \texttt{fun}\ f/k(x_1, \dots, x_k) \to e} \qquad \frac{\Gamma \vdash_{val} v_1 \quad \Gamma \vdash_{val} v_2}{\Gamma \vdash_{val} \texttt{[}v_1 \mid v_2\texttt{]}^{v}} \qquad \frac{}{\Gamma \vdash_{val} \texttt{[]}}$$

$$\frac{\Gamma \vdash_{val} e}{\Gamma \vdash_{exp} e} \qquad \frac{\Gamma \vdash_{exp} e_1 \quad \Gamma \vdash_{exp} e_2}{\Gamma \vdash_{exp} \texttt{[}e_1 \mid e_2\texttt{]}} \qquad \frac{\Gamma \vdash_{exp} e \quad \Gamma \vdash_{exp} e_1 \quad \cdots \quad \Gamma \vdash_{exp} e_k}{\Gamma \vdash_{exp} \texttt{apply}\ e(e_1, \dots, e_k)} \qquad \frac{\Gamma \vdash_{exp} e_1 \quad \Gamma \cup \{x\} \vdash_{exp} e_2}{\Gamma \vdash_{exp} \texttt{let}\ x = e_1\ \texttt{in}\ e_2}$$

$$\frac{\Gamma \cup \{f/k, x_1, \dots, x_k\} \vdash_{exp} e_0 \quad \Gamma \cup \{f/k\} \vdash_{exp} e}{\Gamma \vdash_{exp} \texttt{rec}\ f/k = \texttt{fun}(x_1, \dots, x_k) \to e_0\ \texttt{in}\ e} \qquad \frac{\Gamma \vdash_{exp} e \quad \Gamma \vdash_{exp} e_1 \cdots \Gamma \vdash_{exp} e_k}{\Gamma \vdash_{exp} \texttt{call}\ e(e_1, \dots, e_k)} \qquad \frac{\Gamma \vdash_{exp} e_1 \quad \Gamma \cup vars(p) \vdash_{exp} e_2 \quad \Gamma \vdash_{exp} e_3}{\Gamma \vdash_{exp} \texttt{case}\ e_1\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3}$$

Fig. 3. Scoping rules.

$$\langle K, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rangle \longrightarrow \langle \texttt{let}\ x = \square\ \texttt{in}\ e_2 :: K, e_1 \rangle \tag{SLet}$$

$$\langle K, \texttt{[}e_1 \mid e_2\texttt{]} \rangle \longrightarrow \langle \texttt{[}e_1 \mid \square\texttt{]} :: K, e_2 \rangle \tag{SCons$_1$}$$

$$\langle K, \texttt{apply}\ e(e_1, \dots, e_k) \rangle \longrightarrow \langle \texttt{apply}\ \square(e_1, \dots, e_k) :: K, e \rangle \tag{SApp}$$

$$\langle K, \texttt{call}\ e(e_1, \dots, e_k) \rangle \longrightarrow \langle \texttt{call}\ \square(e_1, \dots, e_k) :: K, e \rangle \tag{SCall}$$

$$\langle K, \texttt{rec}\ f/k = \texttt{fun}(x_1, \dots, x_n) \to e_0\ \texttt{in}\ e \rangle \longrightarrow \langle K, e[f/k \mapsto \texttt{fun}\ f/k(x_1, \dots, x_n) \to e_0] \rangle \tag{SRec}$$

$$\langle K, \texttt{case}\ e_1\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rangle \longrightarrow \langle \texttt{case}\ \square\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: K, e_1 \rangle \tag{SCase}$$

Fig. 4. Frame stack semantics rules (group 1).

We refer to Appendix A for more details about scoping and related theorems.

3.5. *Frame stack semantics*

In our previous work, we defined natural and functional big-step-style semantics for a larger subset of sequential Core Erlang [4,5,34]. However, it proved to be not fine-grained enough for accurate definitions of program equivalence (described in Section 5). On the other hand, the frame stack style is a more fine-grained small-step definition,[2] which also handles continuations (frame stacks) that behave similarly to syntactical expression contexts during the evaluation, which property is quite advantageous when proving correspondence between the equivalence definitions.

First, we describe the syntax of frame stacks and frames, which resemble syntactical contexts. For the stacks, we use lists and use the following notations: ε denotes the empty stack, and $F :: K$ prepends the frame F to K.

Definition 3.4 *(Syntax of frames, frame stacks).*

$$\begin{aligned} F \in \mathit{Frame} ::= {} & \texttt{let}\ x = \square\ \texttt{in}\ e_2 \mid \texttt{case}\ \square\ \texttt{of}\ e_2\ \texttt{then}\ e_3\ \texttt{else} \\ & \mid \texttt{call}\ \square(e_1, \dots, e_k) \mid \texttt{call}\ v(\square, \dots, e_k) \mid \cdots \mid \texttt{call}\ v(v_1, \dots, \square) \\ & \mid \texttt{apply}\ \square(e_1, \dots, e_k) \mid \texttt{apply}\ v(\square, \dots, e_k) \mid \cdots \mid \texttt{apply}\ v(v_1, \dots, \square) \\ & \mid \texttt{[}e_1 \mid \square\texttt{]} \mid \texttt{[}\square \mid v_2\texttt{]} \\ K \in \mathit{FrameStack} ::= {} & \varepsilon \mid F :: K \end{aligned}$$

Definition 3.5 *(Frame stack semantics).* We define the semantics of the language as an inductive relation that reduces configurations $\langle K, e \rangle$ consisting of a frame stack K and a reducible expression e. The frame stack K can also be seen as the continuation of the computation, after the expression e has been evaluated.

Furthermore, we define $\langle K, e \rangle \longrightarrow^n \langle K', e' \rangle$ as the step-indexed reflexive transitive closure of the reduction relation, and $\langle K, e \rangle \longrightarrow^* v$ as the any-step evaluation, formally, $\exists n : \langle K, e \rangle \longrightarrow^n \langle \varepsilon, v \rangle$.

The rules of the semantics can be categorised into three groups:

1. Rules that extract the first redex from an expression (Fig. 4), by putting this redex into the second cell of the configuration (this will be the next reducible expression), and the remaining parts of the expression on top of the frame stack. For example, SApp deconstructs an `apply` expression by promoting its first subexpression for reduction.
2. Rules that modify the frame on the top of the stack (Fig. 5). These rules put the reduced value into the top frame, and start the evaluation of the next redex which is extracted from the frame. For example, SAppNext starts the evaluation of the $(i+1)$st parameter of an `apply` expression, by putting back the value of the ith parameter back to the top frame, and extracting the $(i+1)$st expression to start its evaluation.

[2] The frame stack style definition is also suitable to express the semantics of concurrent language features, opposed to the big-step definitions [35].

$$\langle\, [e_1 \,|\, \square] :: K, v_2\rangle \longrightarrow \langle\, [\square \,|\, v_2] :: K, e_1\rangle \qquad (\textsc{SCons}_2)$$

$$\langle \texttt{apply}\ \square(e_1,\dots,e_k) :: K, v\rangle \longrightarrow \langle \texttt{apply}\ v(\square,\dots,e_k) :: K, e_1\rangle \qquad (\textsc{SAppParam})$$

$$\langle \texttt{call}\ \square(e_1,\dots,e_k) :: K, v\rangle \longrightarrow \langle \texttt{call}\ v(\square,\dots,e_k) :: K, e_1\rangle \qquad (\textsc{SCallParam})$$

$$\langle \texttt{apply}\ v(v_1,\dots,v_{i-1},\square,e_{i+1},\dots,e_k) :: K, v_i\rangle \longrightarrow \langle \texttt{apply}\ v(v_1,\dots,v_{i-1},v_i,\square,e_{i+2},\dots,e_k) :: K, e_{i+1}\rangle \quad (\text{if } i<k) \qquad (\textsc{SAppNext})$$

$$\langle \texttt{call}\ v(v_1,\dots,v_{i-1},\square,e_{i+1},\dots,e_k) :: K, v_i\rangle \longrightarrow \langle \texttt{call}\ v(v_1,\dots,v_{i-1},v_i,\square,e_{i+2},\dots,e_k) :: K, e_{i+1}\rangle \quad (\text{if } i<k) \qquad (\textsc{SCallNext})$$

Fig. 5. Frame stack semantics rules (group 2).

$$\langle \texttt{apply}\ \square() :: K, \texttt{fun}\ f/0() \to e\rangle \longrightarrow \langle K, e[f/0 \mapsto \texttt{fun}\ f/0() \to e]\rangle \qquad (\textsc{PApp}_0)$$

$$\langle \texttt{apply}\ (\texttt{fun}\ f/k(x_1,\dots,x_k) \to e)(v_1,\dots,\square) :: K, v_k\rangle \longrightarrow \langle K, e[f/k \mapsto \texttt{fun}\ f/k(x_1,\dots,x_k) \to e, x_1 \mapsto v_1,\dots,x_k \mapsto v_k]\rangle \qquad (\textsc{PApp})$$

$$\langle \texttt{call}\ \ \texttt{'+'}(i_1,\square) :: K, i_2\rangle \longrightarrow \langle K, i_1 + i_2\rangle \qquad (\textsc{PPlus})$$

$$\langle \texttt{let}\ x = \square\ \texttt{in}\ e_2 :: K, v\rangle \longrightarrow \langle K, e_2[x \mapsto v]\rangle \qquad (\textsc{PLet})$$

$$\langle\, [\square \,|\, v_2] :: K, v_1\rangle \longrightarrow \langle K,\ [v_1 \,|\, v_2]^{v}\rangle \qquad (\textsc{CCons})$$

$$\langle \texttt{case}\ \square\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: K, v\rangle \longrightarrow \langle K, e_2[match(p,v)]\rangle \quad (\text{if } is\_match(p,v)) \qquad (\textsc{PCaseTrue})$$

$$\langle \texttt{case}\ \square\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: K, v\rangle \longrightarrow \langle K, e_3\rangle \quad (\text{if } \neg is\_match(p,v)) \qquad (\textsc{PCaseFalse})$$

Fig. 6. Frame stack semantics rules (group 3).

3. Rules that remove the top frame (Fig. 6). The effect of these rules is different for each language element, thus we will give a brief description of them.
   - $\textsc{PApp}_0$ and $\textsc{PApp}$ describe the final evaluation step for function application. If the top frame contains an `apply` expression, and a function value is either in the second configuration cell (if there are no parameters), or it is inside the top frame. In the second case, $k-1$ values are also contained in the frame as the actual parameters, while the last parameter is the value currently in the second configuration cell. The next reducible expression is the body of the function value, in which the formal parameters are substituted with the actual ones, and the function name is substituted with its definition (to express recursion).
   - $\textsc{PPlus}$ describes how the BIF for addition works. It requires the top frame to contain a `call` expression with `'+'` as the BIF name, and the integer value i_1 as the first parameter, while the evaluated value i_2 is the second integer parameter. These will be added by integer addition resulting in an integer which will be in the second cell of the result configuration.
   - $\textsc{PLet}$ describes when the first subexpression of a `let` expression has been evaluated to a value. The second subexpression, where x is substituted with the computed value, is the next evaluable expression.
   - $\textsc{CCons}$ describes the last evaluation step for lists. If there is a list frame on the top of the stack with a value in its tail, and the head is also evaluated to a value, the result *value* list is constructed by putting the head into this list frame.
   - $\textsc{PCaseTrue}$ and $\textsc{PCaseFalse}$ describe the final step for the evaluation of `case` expressions. If the given pattern matches the computed value, then e_2 is the next evaluable expression, otherwise it is e_3.

Next, we show that the semantics is deterministic. Checking this property acts as a validation step since the reference implementation of sequential Core Erlang is also deterministic. Furthermore, we use this property in later proofs, e.g. in Theorem 5.5.

Theorem 3.1 *(Determinism). For all expressions e, e_1, e_2, frame stacks K, K_1, K_2, if $\langle K, e\rangle \longrightarrow \langle K_1, e_1\rangle$ and $\langle K, e\rangle \longrightarrow \langle K_2, e_2\rangle$, then $K_1 = K_2$ and $e_1 = e_2$.*

Another natural property of the semantics we proved is that the reflexive, transitive closure is indeed transitive and step-indexes add up.

Theorem 3.2 *(Transitivity). For all expressions e_1, e_2, e_3, frame stacks K_1, K_2, K_3, and step counters n_1, n_2, if $\langle K_1, e_1\rangle \longrightarrow^{n_1} \langle K_2, e_2\rangle$ and $\langle K_2, e_2\rangle \longrightarrow^{n_2} \langle K_3, e_3\rangle$, then $\langle K_1, e_1\rangle \longrightarrow^{n_1+n_2} \langle K_3, e_3\rangle$.*

Another advantage of the frame stack semantics is that if there is a reduction sequence between configurations, adding more frames to the bottom of the stacks in both configurations will not affect this evaluation sequence. This property will be important when establishing connection between observational equivalence and other expression equivalence relations. We use $K +\!\!+ K'$ to denote the concatenation of K and K'. (We expect this property to hold even if exceptions are included in the language semantics.)

Theorem 3.3 *(Extend frame stack). For all frame stacks K_1, K_2, K', expressions e_1, e_2, and step counters n, if $\langle K_1, e_1\rangle \longrightarrow^n \langle K_2, e_2\rangle$, then $\langle K_1 +\!\!+ K', e_1\rangle \longrightarrow^n \langle K_2 +\!\!+ K', e_2\rangle$.*

To prove more theorems about the semantics in Section 5.1.1, we first define the termination of configuration inductively in Section 5.1.

3.5.1. Discussion

Evaluation order We have classified the semantics rules in 3 groups. We highlight that the evaluation of expressions always includes the use of exactly one rule from category 1 and 3 above (with the exception of `rec`). Category 1 and 2 are the structural rules of the semantics, they extract the next redex either from a complex expression or from the top frame of the stack, therefore these are the rules that define the evaluation order of expressions. In the formalisation, we implemented a leftmost-innermost evaluation strategy, which follows the behaviour of the reference implementation [36], although, the language specification [37] does not define the evaluation order. Note that the evaluation of lists (defined by rules $\text{SC}\small{\text{ONS}}_1$, $\text{SC}\small{\text{ONS}}_2$ and CCONS) is right-to-left, which also reflects the behaviour of the reference implementation.

Values and expressions An obvious question about the syntax and semantics is why we need two separate list constructors. The simple answer is that we need a syntactic way of differentiating list values from list expressions, and it is related to the semantics rules. In fact, if we only had $[e_1 \,|\, e_2]$, then after having finished evaluating e_1 and e_2 (to v_1 and v_2), we could build back the list with CCONS. However, the result would be $[v_1 \,|\, v_2]$ without the distinction of value lists, thus $\text{SC}\small{\text{ONS}}_1$ could be applied again, and also another rule which would put the result list into the top frame of the stack continuing the evaluation. This anomaly causes nondeterminism and divergence. We could avoid this by using negative preconditions in the rules (i.e. $\text{SC}\small{\text{ONS}}_1$ would use $\neg(\emptyset \vdash_{val} [e_1 \,|\, e_2])$), but this would decrease the usability of the semantics. Moreover, lists of expressions that only contain values would never be evaluated step-by-step.

In the work of Wand et al. [9] with the use of explicit sequencing in `let` expressions, most of the rules from category 1 and 2 could be omitted, but this simplification poses restrictions on the syntax, as mentioned before.

Contrast to previous work Compared to our and others big-step formalisations [4,5,36], this frame stack semantics is more fine-grained, and simpler to use, because the reduction rules do have far less premises as the rules of big-step definitions, thus the use of the frame stack semantics is simpler than the use of the big-step definition for proofs. However, with the frame stack approach, we have to reason about the reflexive transitive closure which is not needed for the big-step semantics. Another advantage of frame stack approach, is that we can also define the concurrent semantics of Core Erlang (which is ongoing work of ours), while the big-step version is not suitable for this work [35].

As mentioned before, our semantics described in this paper covers a subset of sequential Core Erlang. Other authors chose slightly different subsets for their formalisation [14,38], because their formalisation focuses more on the concurrent semantics. From the current formalisations, Fredlund's formalisation of Erlang [13] is the most extensive, and uses a small-step definition. Compared to other small-step semantics, the rules in the frame stack approach do not have any premises about reductions, which means that no induction is necessary on the one-step evaluation, just case distinction.

3.5.2. Example evaluation

To demonstrate how this semantics is used for evaluation, we describe the evaluation of the expression in Fig. 2b with the following parameters: $e_f = $ `fun(X,A) -> [ call '+'(X, 1) | A ]` (this function will add 1 to each element of the parameter list), $e_d =$ `[]`, $e_l =$ `[1,2]`. We use f_{foldr} to denote the function value from the `rec` expression in Fig. 2b. For readability, we use $\longrightarrow^*$ to denote any-step reduction between two configurations here.

In the first steps, we evaluate the `rec` expression, which substitutes the `'foldr'/3` identifiers. Then we start evaluating the subexpressions of the application. The function expression and `[]` evaluate to themselves, while the list of `[1,2]` needs to be deconstructed, and reconstructed as a list of values.

$$\langle \varepsilon, \texttt{rec 'foldr'/3 = ... in apply 'foldr'/3}(e_f, \texttt{[]}, \texttt{[1,2]})\rangle \longrightarrow$$
$$\langle \varepsilon, \texttt{apply } f_{foldr}(e_f, \texttt{[]}, \texttt{[1,2]})\rangle \longrightarrow^*$$
$$\langle \texttt{apply } f_{foldr}(e_f, \texttt{[]}, \Box) :: \varepsilon, \texttt{[1,2]}\rangle \longrightarrow$$
$$\langle \texttt{[1|}\Box\texttt{]} :: \texttt{apply } f_{foldr}(e_f, \texttt{[]}, \Box) :: \varepsilon, \texttt{[2]}\rangle \longrightarrow$$
$$\langle \texttt{[2|}\Box\texttt{]} :: \texttt{[1|}\Box\texttt{]} :: \texttt{apply } f_{foldr}(e_f, \texttt{[]}, \Box) :: \varepsilon, \texttt{[]}\rangle \longrightarrow$$
$$\langle \texttt{[}\Box\texttt{| []]} :: \texttt{[1|}\Box\texttt{]} :: \texttt{apply } f_{foldr}(e_f, \texttt{[]}, \Box) :: \varepsilon, 2\rangle \longrightarrow$$
$$\langle \texttt{[1|}\Box\texttt{]} :: \texttt{apply } f_{foldr}(e_f, \texttt{[]}, \Box) :: \varepsilon, \texttt{[2]}^v\rangle \longrightarrow$$
$$\langle \texttt{[}\Box\texttt{| [2]}^v\texttt{]} :: \texttt{apply } f_{foldr}(e_f, \texttt{[]}, \Box) :: \varepsilon, 1\rangle \longrightarrow$$
$$\langle \texttt{apply } f_{foldr}(e_f, \texttt{[]}, \Box) :: \varepsilon, \texttt{[1,2]}^v\rangle$$

Next, we evaluate the application of the function value f_{foldr} by substituting in its body (see in Fig. 2b, `L` is substituted with `[1,2]`^{v}, and `D` to `[]`). The pattern matching succeeds, and the application `apply F(H, apply 'foldr'/3(`e_f`, D, T)` (denoted by *body*) is evaluated. First, the parameters of this application are evaluated, which means the recursive calls in this case.

$$\langle \texttt{apply}\ f_{foldr}(e_f,\ \texttt{[]},\Box) :: \varepsilon,\ \texttt{[1,2]}^v\rangle \longrightarrow$$
$$\langle \varepsilon, \texttt{case [1,2]}^v\ \texttt{of [H | T] then}\ \textit{body}\ \texttt{else []}\rangle \longrightarrow$$
$$\langle \varepsilon, \texttt{apply}\ e_f(1, \texttt{apply}\ f_{foldr}(e_f,\ \texttt{[]},\ \texttt{[2]}^v))\rangle \longrightarrow^*$$
$$\langle \texttt{apply}\ e_f(1,\Box) :: \varepsilon, \texttt{case [2]}^v\ \texttt{of [H | T] then}\ \textit{body}\ \texttt{else []}\rangle \longrightarrow^*$$
$$\langle \texttt{apply}\ e_f(2,\Box) :: \texttt{apply}\ e_f(1,\Box) :: \varepsilon, \texttt{case [] of [H | T] then}\ \textit{body}\ \texttt{else []}\rangle \longrightarrow^*$$
$$\langle \texttt{apply}\ e_f(2,\Box) :: \texttt{apply}\ e_f(1,\Box) :: \varepsilon, []\rangle$$

After the end of the recursion is reached, the application of the parameter function (in this case, the successor function) starts, which builds back the list.

$$\langle \texttt{apply}\ e_f(2,\Box) :: \texttt{apply}\ e_f(1,\Box) :: \varepsilon, []\rangle \longrightarrow^*$$
$$\langle \texttt{apply}\ e_f(1,\Box) :: \varepsilon, \texttt{[call '+'(2, 1) | []]}\rangle \longrightarrow^*$$
$$\langle \texttt{apply}\ e_f(1,\Box) :: \varepsilon,\ \texttt{[3]}^v\rangle \longrightarrow^*$$
$$\langle \varepsilon, \texttt{[call '+'(1, 1) | [3]}^v\texttt{]}\rangle \longrightarrow^*$$
$$\langle \varepsilon,\ \texttt{[2, 3]}^v\rangle$$

Now we proceed to prove a more general property about the evaluation of the expression in Fig. 2b: its correspondence with the *map* function. First, let us introduce some preliminaries:

- An object-level list of values v is proper, if it is built up in the following way: $v = \texttt{[}v_1\,|\,\texttt{[}v_2\,|\,\ldots\,\texttt{[}v_k\,|\,\texttt{[]]}^v\ldots\texttt{]}^v\texttt{]}^v$.
- A proper object-level list of values v can be expressed as a meta-level list of values, we will denote it with $\overline{v}$.
- Similarly, any meta-level list $[v_1, \ldots, v_n]$ of values can be expressed as a proper, object-level list, which we denote by $\underline{[v_1, \ldots, v_n]}$.
- An object-level function value v_f computes the meta-level function f, if for all closed values $v : \langle \varepsilon, \texttt{apply}\ v_f(v)\rangle \longrightarrow^* f(v)$.

Lemma 3.4 *(Evaluation of* `'foldr'`*). For all closed values v_l representing a proper list, closed function values v_f that compute a function f, if $v_g =$* `fun(X, A) -> [apply` v_f`(X) | A]` *we can prove that*

$$\langle \varepsilon, \texttt{rec 'foldr'/3 = ... in apply 'foldr'/3}(v_g,\ \texttt{[]}, v_l)\rangle \longrightarrow^* \underline{map(f, \overline{v_l})}$$

where map denotes the metatheoretical list transforming function, which applies f to every element of its parameter list.

Proof. We proceed by induction on the length of v_l. If v_l was empty, the result will also be empty.

In the inductive case, there is a first element of the parameter list: $v_l = v :: v_l' = [v, v_1, \ldots, v_k]$. From the induction hypothesis, we can derive

$$\langle \varepsilon, \texttt{apply}\ f_{foldr}(v_1, \ldots, v_k)\rangle \longrightarrow^* \underline{map\ f\ [v_1, \ldots, v_k]}$$

(where f_{foldr} denotes the function value from the `rec` expression).

Next, we take some steps with the semantics in the conclusion to reach the configuration

$$\langle \texttt{apply}\ v_g(v,\Box) :: \varepsilon, \texttt{apply}\ f_{foldr}(v_1, \ldots, v_k)\rangle$$

where we can apply the induction hypothesis with the help of Theorem 3.3 and Theorem 3.2. This results in the goal of

$$\langle \texttt{apply}\ v_g(v,\Box) :: \varepsilon, \underline{map\ f\ [v_1, \ldots, v_k]}\rangle \longrightarrow^* \underline{map\ f\ [v, v_1, \ldots, v_k]}\rangle$$

which can be reached by taking a number of steps with the semantics. □

4. Naive definitions of program equivalence

In this section we briefly overview naive and too strict definitions of expression equivalence. The following relations are simply induced by the evaluation relation, and are insufficient in the case of this higher-order functional language.

4.1. Behavioural equivalence

Naive program equivalence (simple behavioural equivalence by the textbook [7] definition) says two expressions to be (behaviourally) equivalent if and only if they evaluate to the same *result*, or they both diverge. Note that the equivalence relation is typically defined by the symmetrisation of a preorder relation.

Definition 4.1 *(Naive behavioural equivalence).*

$$e_1 \leq_b e_2 \overset{\text{def}}{=} \forall v : \langle \varepsilon, e_1 \rangle \longrightarrow^* v \implies \langle \varepsilon, e_2 \rangle \longrightarrow^* v$$

$$e_1 \equiv_b e_2 \overset{\text{def}}{=} e_1 \leq_b e_2 \wedge e_2 \leq_b e_1$$

In the formalisation, we prove the $\equiv_b$ relation to be an equivalence, that is, show that it is reflexive, symmetric, and transitive. We also show that it is indeed a behavioural equivalence over our expressions, characterised by congruence [7] property. Congruence helps prove compound expressions equivalent, but it took significant effort to formalise and prove in Coq [34].

Equivalence of function expressions The definition of behavioural equivalence checks for equality of result values. In case of integer, atom, or list expressions, it relates expressions understood equivalent indeed. However, for function expressions, checking strict equality is likely not to meet our expectations:

$$\texttt{fun f/1(X)} \rightarrow \texttt{call '+'(X, 2)} = \texttt{fun f/1(X)} \rightarrow \texttt{call '+'(call '+'(X, 1), 1)}$$

Obviously, we would expect functions with equivalent bodies to be equivalent, but as exemplified by the above snippets, the naive approach only relates function closures whose body expressions are structurally equal. With this naive definition, we can only prove identical function expressions equivalent.

4.2. Naive contextual equivalence

In case of function expressions, the equivalence relation should depend on whether the body expressions behave the same way in the same *expression contexts*, i.e. they are contextually equivalent. In other words, contexts are supposed to reveal any observable differences between the expressions.

We proceed by defining expression contexts, which are basically expressions with one of their subexpressions replaced by a *hole*. The hole can be substituted with any expression (e.g. $\texttt{apply}\ f(1, \square, 3)[2] = \texttt{apply}\ f(1, 2, 3)$) to obtain valid expressions. We define expression contexts as follows:

Definition 4.2 *(Expression context).*

$$\begin{aligned}
C \in \mathit{Context} ::= {} & \square \mid \texttt{fun}\ f/k(x_1, \ldots, x_k) \rightarrow C \\
& \mid \texttt{apply}\ C(e_1, \ldots, e_k) \mid \texttt{apply}\ e(C, \ldots, e_k) \mid \ldots \mid \texttt{apply}\ e(e_1, \ldots, C) \\
& \mid \texttt{call}\ C(e_1, \ldots, e_k) \mid \texttt{call}\ e(C, \ldots, e_k) \mid \ldots \mid \texttt{call}\ e(e_1, \ldots, C) \\
& \mid \texttt{rec}\ f/k = \texttt{fun}(x_1, \ldots, x_k) \rightarrow C\ \texttt{in}\ e \mid \texttt{let}\ x = C\ \texttt{in}\ e_2 \\
& \mid \texttt{rec}\ f/k = \texttt{fun}(x_1, \ldots, x_k) \rightarrow e_0\ \texttt{in}\ C \mid \texttt{let}\ x = e_1\ \texttt{in}\ C \\
& \mid \texttt{[}C \mid e_2\texttt{]} \mid \texttt{[}e_1 \mid C\texttt{]} \mid \texttt{case}\ C\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
& \mid \texttt{case}\ e_1\ \texttt{of}\ p\ \texttt{then}\ C\ \texttt{else}\ e_3 \mid \texttt{case}\ e_1\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ C
\end{aligned}$$

We define the substitution of the $\square$ with an expression in the usual way (e.g. see [8]), and we will denote it by $C[e]$. Next, we define the contextual equivalence using contextual preorders.

Definition 4.3 *(Naive contextual equivalence).*

$$e_1 \leq_{ctx} e_2 \overset{\text{def}}{=} \forall C, v : \langle \varepsilon, C[e_1] \rangle \longrightarrow^* v \implies \langle \varepsilon, C[e_2] \rangle \longrightarrow^* v$$

$$e_1 \equiv_{ctx} e_2 \overset{\text{def}}{=} e_1 \leq_{ctx} e_2 \wedge e_2 \leq_{ctx} e_1$$

Let us point out that this definition of naive contextual equivalence cannot overcome the issue with function expressions: if we consider syntactical contexts too, the value (i.e. the function closure) is checked for equality in the empty context. Actually, it escalates the problem even further: the latter relation coincides with syntactical equality since it requires the

$$\frac{}{\langle \varepsilon, v\rangle \Downarrow^0} \qquad \frac{\langle \texttt{[}e_1 \,|\, \Box\texttt{]} :: K, e_2\rangle \Downarrow^n}{\langle K,\ \texttt{[}e_1 \,|\, e_2\texttt{]}\rangle \Downarrow^{1+n}} \qquad \frac{\langle \texttt{[}\Box \,|\, v_2\texttt{]} :: K, e_1\rangle \Downarrow^n}{\langle \texttt{[}e_1 \,|\, \Box\texttt{]} :: K, v_2\rangle \Downarrow^{1+n}} \qquad \frac{\langle K,\ \texttt{[}v_1 \,|\, v_2\texttt{]}^{v}\rangle \Downarrow^n}{\langle \texttt{[}\Box \,|\, v_2\texttt{]} :: K, v_1\rangle \Downarrow^{1+n}}$$

$$\frac{\langle \texttt{case}\ \Box\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: K, e\rangle \Downarrow^n}{\langle K, \texttt{case}\ e\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3\rangle \Downarrow^{1+n}} \qquad \frac{\langle K, e_2[\mathit{match}(p,v)]\rangle \Downarrow^n \quad \mathit{is\_match}(p,v)}{\langle \texttt{case}\ \Box\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: K, v\rangle \Downarrow^{1+n}} \qquad \frac{\langle K, e_3\rangle \Downarrow^n \quad \neg \mathit{is\_match}(p,v)}{\langle \texttt{case}\ \Box\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: K, v\rangle \Downarrow^{1+n}}$$

$$\frac{\langle \texttt{let}\ x = \Box\ \texttt{in}\ e_2 :: K, e\rangle \Downarrow^n}{\langle K, \texttt{let}\ x = e\ \texttt{in}\ e_2\rangle \Downarrow^{1+n}} \qquad \frac{\langle K, e_2[x \mapsto v]\rangle \Downarrow^n}{\langle \texttt{let}\ x = \Box\ \texttt{in}\ e_2 :: K, v\rangle \Downarrow^{1+n}} \qquad \frac{\langle K, e[f/k \mapsto \texttt{fun}\ f/k(x_1,\dots,x_k) \to e_0]\rangle \Downarrow^n}{\langle K, \texttt{rec}\ f/k = \texttt{fun}(x_1,\dots,x_k) \to e_0\ \texttt{in}\ e\rangle \Downarrow^{1+n}}$$

$$\frac{\langle \texttt{apply}\ \Box(e_1,\dots,e_k) :: K, e\rangle \Downarrow^n}{\langle K, \texttt{apply}\ e(e_1,\dots,e_k)))\rangle \Downarrow^{1+n}} \qquad \frac{\langle \texttt{apply}\ v(\Box,\dots,e_k)) :: K, e_1\rangle \Downarrow^n}{\langle \texttt{apply}\ \Box(e_1,\dots,e_k)) :: K, v\rangle \Downarrow^{1+n}} \qquad \frac{\langle \texttt{apply}\ v(v_1,\dots,v_i,\Box,e_{i+2},\dots,e_k) :: K, e_{i+1}\rangle \Downarrow^n}{\langle \texttt{apply}\ v(v_1,\dots,v_{i-1},\Box,e_{i+1},\dots e_k) :: K, v_i\rangle \Downarrow^{1+n}}$$

$$\frac{\langle \texttt{call}\ \Box(e_1,\dots,e_k) :: K, e\rangle \Downarrow^n}{\langle K, \texttt{call}\ e(e_1,\dots,e_k)))\rangle \Downarrow^{1+n}} \qquad \frac{\langle \texttt{call}\ v(\Box,\dots,e_k)) :: K, e_1\rangle \Downarrow^n}{\langle \texttt{call}\ \Box(e_1,\dots,e_k)) :: K, v\rangle \Downarrow^{1+n}} \qquad \frac{\langle \texttt{call}\ v(v_1,\dots,v_i,\Box,e_{i+2},\dots,e_k) :: K, e_{i+1}\rangle \Downarrow^n}{\langle \texttt{call}\ v(v_1,\dots,v_{i-1},\Box,e_{i+1},\dots e_k) :: K, v_i\rangle \Downarrow^{1+n}}$$

$$\frac{\langle K, b[f/0 \mapsto \texttt{fun}\ f/0() \to e_0]\rangle \Downarrow^n}{\langle \texttt{apply}\ \Box() :: K, \texttt{fun}\ f/0() \to e_0\rangle \Downarrow^{1+n}} \qquad \frac{\langle K, i_1 + i_2\rangle \Downarrow^n}{\langle \texttt{call}\ \ \texttt{'+'}(i_1,\Box) :: K, i_2\rangle \Downarrow^{1+n}} \qquad \frac{\langle K, b[f/k \mapsto \texttt{fun}\ f/k(x_1,\dots,x_k) \to e_0, x_1 \mapsto v_1,\dots,x_k \mapsto v_k]\rangle \Downarrow^n}{\langle \texttt{apply}\ (\texttt{fun}\ f/k(x_1,\dots,x_k) \to e_0)(v_1,\dots,\Box) :: K, v_k\rangle \Downarrow^{1+n}}$$

Fig. 7. Step-indexed, frame stack-style termination relation.

expressions to yield equal values, even in function abstraction contexts. Clearly, a special notion of value equality is needed to treat function expressions properly.

In addition, while it is straightforward to prove that two expressions are not contextually equivalent (even with a less strict notion of contextual equivalence, see Section 5.5), the proof of equivalence in general is understood to be significantly more complex as it requires induction over contexts. In order to overcome these issues, we seek differently formulated equivalence relations, which coincide with our intuition and at the same time they ease proving two expressions equivalent.

5. More accurate definitions of program equivalence

To reason about the correctness of refactorings, an appropriate and accurate program equivalence definition is needed for the object language. In this section, we refine the contextual equivalence relation introduced in the previous section, and we present a number of alternative, equal definitions, such as step-indexed logical relations [8,9,39] and CIU ("closed instances of uses") relations.

5.1. Frame stack termination relation

As briefly mentioned in Section 2, it is an important result discussed in related work that two expressions can be shown contextually equivalent by proving that they both terminate or both diverge in arbitrary contexts. In other words, it suffices to prove this termination property even when arguing about expression equivalences appearing in the verification of refactoring.

Definition 5.1 *(Termination relation).* We formalise termination as an inductive, step-indexed relation, denoted by $\langle K, e\rangle \Downarrow^n$) in Fig. 7. We also introduce the notation $\langle K, e\rangle \Downarrow$ for any-step termination: $\exists n : \langle K, e\rangle \Downarrow^n$.

Remark The derivation rules of termination can be obtained by transforming the reduction rules of the semantics in Definition 3.5 in the following way. Suppose that there is a reduction rule $\langle K, e\rangle \longrightarrow \langle K', e'\rangle$. To prove that the configuration on the left-hand side terminates in some steps ($\langle K, e\rangle \Downarrow^{n+1}$), it suffices to prove that the configuration on the right-hand side terminates in one less step ($\langle K', e'\rangle \Downarrow^n$).

5.1.1. Properties of termination and the frame stack semantics

With the termination relation defined, we prove the following correspondence as the validation of the definition of termination.

Theorem 5.1 *(Step-indexed terminations coincide). For all frame stacks K, expressions e and step-counters n, $\langle K, e\rangle \Downarrow^n$ if and only if there is a* closed value *v that satisfies $\langle K, e\rangle \longrightarrow^n \langle \varepsilon, v\rangle$.*

Proof. This lemma can be proven by induction on the step-index. The subgoal can be solved by basically applying the induction hypotheses for the subexpressions. For more details, we refer to the Coq implementation [32]. □

For the following theorems, we extend the scoping to frames and frame stacks. We reuse the notation $F[e]$ to substitute the $\Box$ in frame F with e.

Definition 5.2 *(Closed frame stacks).* A frame stack K is said to be closed if and only if all of its frames are closed. A frame F is closed, if for all closed expressions e, the expression $F[e]$ is closed.

The concept of closed frame stacks is needed if we investigate evaluation which removes frames of the stack. In general, when an expression is evaluated, all of its free variables are substituted before it is pushed to the stack [17], thus evaluations that involve removing elements from the stack assume that all free variables have been substituted (i.e. the stack is closed).

We have investigated several properties of the semantics relevant to the termination relation too; here we highlight some of these, the rest can be found in the Coq formalisation [32]. The following two theorems state that frames from the stack can be transformed back to expressions by substituting the $\square$ with the current expression under evaluation.

Theorem 5.2 *(Remove frame). For all closed frames F, closed expressions e, and all frame stacks K, if $\langle F :: K, e\rangle \Downarrow$ then $\langle K, F[e]\rangle \Downarrow$.*

Proof. From $\langle F :: K, e\rangle \Downarrow$ we assume that this termination takes k steps. We proceed with case distinction on frame F: depending on the structure of F, the evaluation of $F[e]$ should take $n + k$ steps (e.g. for $F = \square + e_2$, $n = 1$, while for $F = v_1 + \square$, $n = 2$, etc.) to reach the configuration in the premise ($\langle F :: K, e\rangle \Downarrow^k$). $\square$

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

Theorem 5.3 *(Add frame). For all closed frames F, closed expressions e, and all frame stacks K, if $\langle K, F[e]\rangle \Downarrow$ then $\langle F :: K, e\rangle \Downarrow$.*

Proof. This proof is basically the reverse of the previous one. Again, we do case distinction on F, and then we inspect the premise $\langle K, F[e]\rangle \Downarrow$ and investigate how this derivation could have been done. After taking some steps (e.g. for $F = \square + e_2$, one step is enough, while for $F = v_1 + \square$, two steps are needed, etc.) we reach the configuration with some k number $\langle F :: K, e\rangle \Downarrow^k$ we need to prove to terminate. $\square$

We highlight two more theorems that relate the termination relation with the semantics. The first theorem states that if a configuration can be reduced to a terminating one, then the former terminates too, and the steps add up.

Corollary 5.4. *For all frame stacks K_1, K_2, expressions e_1, e_2, and step counters n_1, n_2, if $\langle K_1, e_1\rangle \longrightarrow^{n_1} \langle K_2, e_2\rangle$ and $\langle K_2, e_2\rangle \Downarrow^{n_2}$, then $\langle K_1, e_1\rangle \Downarrow^{n_1+n_2}$.*

Proof. This theorem is a consequence of the fact that termination can be expressed in the semantics too (Theorem 5.1). Next, transitivity (Theorem 3.2) can be used to chain the obtained reduction sequence with the one in the premise. $\square$

The next theorem states that if there is a terminating configuration that can be reduced in some steps, then the result configuration is also terminating.

Theorem 5.5. *For all frame stacks K_1, K_2, expressions e_1, e_2, and step counters n_1, n_2, if $\langle K_1, e_1\rangle \Downarrow^{n_1}$ and $\langle K_1, e_1\rangle \longrightarrow^{n_2} \langle K_2, e_2\rangle$, then $\langle K_2, e_2\rangle \Downarrow^{n_1-n_2}$.*

Note that determinism (Theorem 3.1) is crucial for this theorem, but for the sake of brevity, we omit the proof and refer to the formalisation [32] for details.

5.2. The logical relation

The majority of related work on program equivalence proposes logical relations, amongst others, for arguing about standard contextual equivalence. At first we followed the techniques of Pitts [8] and adapted his "logical simulation relation". Unfortunately, their mathematical definitions cannot be directly formalised in Coq for this untyped language as the statements they use are not well-founded without type-indexing and therefore do not pass Coq's positivity checker. Neither could we adapt the techniques of Culpepper and Cobb [25], because they also use types to base their relations on. Therefore, we decided to adopt the idea of step-indexed relations [9,29].

First, we define these logical relations for closed values, expressions and frame stacks. For better readability, we omit the assumptions of closedness from the definitions. We use the same notations as Wand et al. [9]. We invite the reader to observe how the value relation addresses the previously seen issue of function expression equivalence by relating the body expressions.

Definition 5.3 *(Logical relations for closed expressions, values and frame stacks).* We describe the mutually dependent definitions of the logical relations. 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 at index n if: whenever the first one terminates in at most n steps in some frame stack K_1 the second one terminates in any stack related to K_1.

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

We denote the set of related frame stacks $\mathbb{K}_n$, where n is a step counter. The following definition is similar to the previous one. Two stacks are related at index n if: whenever the first one terminates in at most n steps with a value v_1 the second one terminates in a configuration with any value related to v_1.

$$(K_1, K_2) \in \mathbb{K}_n \overset{\text{def}}{=} (\forall m \leq n, v_1, v_2 : (v_1, v_2) \in \mathbb{V}_m \implies \langle K_1, v_1 \rangle \Downarrow^m \implies \langle K_2, v_2 \rangle \Downarrow)$$

Finally, we define the concept of related values (the set of these value pairs is denoted by $\mathbb{V}_n$, where n is a step counter). This relation defines the base cases of the mutual definitions. Two atoms or 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. Two functions are related, if their bodies substituted with pairwise-related actual parameters are related expressions.

$$(i_1, i_2) \in \mathbb{V}_n \overset{\text{def}}{=} i_1 = i_2$$

$$(a_1, a_2) \in \mathbb{V}_n \overset{\text{def}}{=} a_1 = a_2$$

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

$$(\texttt{[}v_1 \,|\, v_2\texttt{]}^v, \texttt{[}v_1' \,|\, v_2'\texttt{]}^v) \in \mathbb{V}_n \overset{\text{def}}{=} (v_1, v_1') \in \mathbb{V}_n \land (v_2, v_2') \in \mathbb{V}_n$$

$$\begin{aligned}
&(\texttt{fun}\ f/k(x_1, \dots, x_k) \to e, \texttt{fun}\ f/k(x_1, \dots, x_k) \to e') \in \mathbb{V}_n \overset{\text{def}}{=} \\
&\quad (\forall m < n : \forall v_1, v_1', \dots, v_k, v_k' : (v_1, v_1') \in \mathbb{V}_m \land \dots \land (v_k, v_k') \in \mathbb{V}_m \implies \\
&\qquad (e[f/k \mapsto \texttt{fun}\ f/k(x_1, \dots, x_k) \to e, x_1 \mapsto v_1, \dots, x_k \mapsto v_k], \\
&\qquad e'[f/k \mapsto \texttt{fun}\ f/k(x_1, \dots, x_k) \to e', x_1 \mapsto v_1', \dots, x_k \mapsto v_k']) \in \mathbb{E}_m)
\end{aligned}$$

To ensure the well-foundedness of these relations, we used the step-index, which is decreased in $\mathbb{V}$ for functions ($m < n$, while in the other relations we use $m \leq n$). For list values on the other hand, we did not decrease this index, we only used structural recursion. Alternatively, the step-index can be reduced in this case too, but then the mechanism of pattern matching needs to be formalised in a step-indexed way too.

Just like in the work of Wand et al. [9], these relations with higher indices can differentiate more expressions, values and stacks, i.e. $\mathbb{V}_0 \supseteq \mathbb{V}_1 \supseteq \dots \supseteq \mathbb{V}_{n-1} \supseteq \mathbb{V}_n$ (also for $\mathbb{E}_n$ and $\mathbb{K}_n$), i.e. these relations have this monotonicity property defined based on the step counter. For example, $\mathbb{E}_0$ contains every expression, except the non-equal values that are not functions, because only values can terminate in 0 steps in empty stacks. The above relations can be generalised to open expressions (or values) with closing substitutions (i.e. all free variables of the expression replaced by closed values).

Definition 5.4 *(Logical relations with closing substitutions).* First, we define the notion of related, closing substitutions. We denote their set 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.

$$(\sigma_1, \sigma_2) \in \mathbb{G}_n^\Gamma \overset{\text{def}}{=} \Gamma \vdash_{sub} \sigma_1 \multimap \emptyset \land \Gamma \vdash_{sub} \sigma_2 \multimap \emptyset \land (\forall x \in \Gamma : (\sigma_1(x), \sigma_2(x)) \in \mathbb{V}_n)$$

With the concept of related closing substitutions, we can define the logical relations for open expressions (supposing that they have the same scope Γ).

$$(v_1, v_2) \in \mathbb{V}^\Gamma \overset{\text{def}}{=} \Gamma \vdash_{val} v_1 \land \Gamma \vdash_{val} v_2 \land (\forall n, \sigma_1, \sigma_2 : (\sigma_1, \sigma_2) \in \mathbb{G}_n^\Gamma \implies (v_1[\sigma_1], v_2[\sigma_2]) \in \mathbb{V}_n)$$

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

After having these relations defined, we proceeded to prove their two most important properties [9,25,29]: the compatibility rules which are forms of congruence and the "fundamental property" (a form of reflexivity). In our formalisation, we state and prove a number of lemmas that support the proof of the main theorems (we refer to the implementation [32] for more details).

Theorem 5.6 *(Compatibility of expressions and values). The logical relations satisfy each of the following implications:*

$$\frac{(v, v') \in \mathbb{V}^{\Gamma}}{(v, v') \in \mathbb{E}^{\Gamma}} \qquad \frac{x \in \Gamma}{(x, x) \in \mathbb{V}^{\Gamma}} \qquad \frac{f/k \in \Gamma}{(f/k, f/k) \in \mathbb{V}^{\Gamma}} \qquad \frac{}{(a, a) \in \mathbb{V}^{\Gamma}} \qquad \frac{}{(i, i) \in \mathbb{V}^{\Gamma}} \qquad \frac{}{(\texttt{[]}, \texttt{[]}) \in \mathbb{V}^{\Gamma}}$$

$$\frac{(v_1, v_1') \in \mathbb{V}^{\Gamma} \quad (v_2, v_2') \in \mathbb{V}^{\Gamma}}{(\texttt{[}v_1 \,|\, v_2\texttt{]}^{v}, \texttt{[}v_1' \,|\, v_2'\texttt{]}^{v}) \in \mathbb{V}^{\Gamma}} \qquad \frac{(e_1, e_1') \in \mathbb{E}^{\Gamma} \quad (e_2, e_2') \in \mathbb{E}^{\Gamma}}{(\texttt{[}e_1 \,|\, e_2\texttt{]}, \texttt{[}e_1' \,|\, e_2'\texttt{]}) \in \mathbb{E}^{\Gamma}}$$

$$\frac{(e_1, e_2) \in \mathbb{E}^{\Gamma \cup \{f/k, x_1, \dots, x_k\}}}{(\texttt{fun}\ f/k(x_1, \dots, x_k) \to e, \texttt{fun}\ f/k(x_1, \dots, x_k) \to e_2) \in \mathbb{V}^{\Gamma}} \qquad \frac{(e_1, e_1') \in \mathbb{E}^{\Gamma} \quad (e_2, e_2') \in \mathbb{E}^{\Gamma \cup \{x\}}}{(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, \texttt{let}\ x = e_1'\ \texttt{in}\ e_2') \in \mathbb{E}^{\Gamma}}$$

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

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

$$\frac{(e_1, e_1') \in \mathbb{E}^{\Gamma} \quad (e_2, e_2') \in \mathbb{E}^{\Gamma \cup vars(p)} \quad (e_3, e_3') \in \mathbb{E}^{\Gamma}}{(\texttt{case}\ e_1\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \texttt{case}\ e_1'\ \texttt{of}\ p\ \texttt{then}\ e_2'\ \texttt{else}\ e_3') \in \mathbb{E}^{\Gamma}}$$

Proof. The compatibility rules for the non-recursive language constructs follow from the definitions. For recursive functions, induction was needed by the step-index. For the other cases, we give a representative proof outline with the compatibility proof of expression lists:

1. Give proof for closed expressions, then the compatibility with the closing substitutions is just a consequence of it.
2. From the premise $\langle K, \texttt{[}e_1 \,|\, e_2\texttt{]} \rangle \Downarrow^m$ (for any $m \le n$), we can deduce $\langle \texttt{[}e_1 \,|\, \square\texttt{]} :: K, e_2 \rangle \Downarrow^{m-1}$ from the definition of the termination. For the other derivation, we have $(K, K') \in \mathbb{K}_n$. To get $\langle K', \texttt{[}e_1' \,|\, e_2'\texttt{]} \rangle \Downarrow$ it is sufficient to prove $\langle \texttt{[}e_1 \,|\, \square\texttt{]} :: K', e_2' \rangle \Downarrow$. Now, we apply the premise $(e_1, e_1') \in \mathbb{E}_n$, to conclude this subproof, but we still need to show that the two frame stacks are in relation.
3. To prove $(\texttt{[}e_1 \,|\, \square\texttt{]} :: K, \texttt{[}e_1' \,|\, \square\texttt{]} :: K') \in \mathbb{K}_m$, we need to prove that for any $k \le m$, $(v_2, v_2') \in \mathbb{V}_m : \langle \texttt{[}e_1 \,|\, \square\texttt{]} :: K, v_2 \rangle \Downarrow^k$ implies $\langle \texttt{[}e_1' \,|\, \square\texttt{]} :: K', v_2' \rangle \Downarrow$. By definition, we can transform both the premise and the conclusion (just like above): $\langle \texttt{[}\square \,|\, v_2\texttt{]} :: K, e_1 \rangle \Downarrow^{k-1}$ implies $\langle \texttt{[}\square \,|\, v_2'\texttt{]} :: K', e_2' \rangle \Downarrow$, which can be proven by $(e_2, e_2') \in \mathbb{E}_n$, since $k - 1 < k \le m \le n$. However, this step introduces another premise of the new frame stacks to be in relation.
4. To prove $(\texttt{[}\square \,|\, v_2\texttt{]} :: K, \texttt{[}\square \,|\, v_2'\texttt{]} :: K') \in \mathbb{K}_m$, we have to go through the same steps as before. We need to prove that $j \le k$, $(v_1, v_1') \in \mathbb{V}_k : \langle \texttt{[}\square \,|\, v_2\texttt{]} :: K, v_1 \rangle \Downarrow^j$ implies $\langle \texttt{[}\square \,|\, v_2'\texttt{]} :: K', v_1' \rangle \Downarrow$. Once again, we apply the definition of the termination to get $\langle K, \texttt{[}v_1 \,|\, v_2\texttt{]}^{v} \rangle \Downarrow^{j-1}$ implies $\langle K', \texttt{[}v_1' \,|\, v_2'\texttt{]}^{v} \rangle \Downarrow$. This goal can be solved by the original assumption of $(K, K') \in \mathbb{K}_n$, since $j - 1 < n$ and $\mathbb{K}$ is monotone. However, to use this assumption, we have to prove that $(\texttt{[}v_1 \,|\, v_2\texttt{]}^{v}, \texttt{[}v_1' \,|\, v_2'\texttt{]}^{v}) \in \mathbb{V}_{j-1}$. According to the definition of $\mathbb{V}_{j-1}$, it is sufficient to prove that $(v_1, v_1') \in \mathbb{V}_{j-1}$ and $(v_2, v_2') \in \mathbb{V}_{j-1}$ which can be solved by the monotonicity property and the assumptions created during the proof (namely $(v_2, v_2') \in \mathbb{V}_m$ in step 3, and $(v_1, v_1') \in \mathbb{V}_k$ in step 4).

For the complete proof we refer to the formalisation [32]. □

The fundamental property (a form of reflexivity) of the logical relations is a consequence of Theorem 5.6.

Theorem 5.7 *(Fundamental property). For all scopes Γ the following properties hold:*

- *For all expressions e, if $\Gamma \vdash_{exp} e$ then $(e, e) \in \mathbb{E}^{\Gamma}$;*
- *For all values v, if $\Gamma \vdash_{val} v$ then $(v, v) \in \mathbb{V}^{\Gamma}$;*
- *For all closing substitutions σ, if $\Gamma \vdash_{sub} \sigma \multimap \emptyset$ then for all step counters n, $(\sigma, \sigma) \in \mathbb{G}_n^{\Gamma}$ holds.*

Proof. We carry out induction on e (and v resp.). Then for all cases, we can just use the corresponding compatibility rule from Theorem 5.6, moreover, the premises of these rules are satisfied either by the scoping premises or the induction hypotheses.

The fundamental property of $\mathbb{G}_n^{\Gamma}$ follows from the fundamental property of $\mathbb{V}^{\Gamma}$. □

All in all, logical relations proved to be useful for proofs, since these definitions do not require reasoning about the same frame stacks and substitutions, but related pairs, which enables more flexibility in the proofs. However, this advantage turns into a disadvantage while proving concrete expressions equivalent.

5.3. CIU equivalence

Alongside proving the properties of the logical relations, we have also formalised CIU ("closed instances of uses") preorder and equivalence relations [22].

Definition 5.5 *(CIU preorder).* Two expressions are CIU equivalent if they both terminate or diverge when placed in arbitrary reduction contexts.

$$e_1 \leq_{ciu} e_2 \overset{\text{def}}{=} \emptyset \vdash_{exp} e_1 \wedge \emptyset \vdash_{exp} e_2 \wedge (\forall K : K \textit{ is closed } \wedge \langle K, e_1 \rangle \Downarrow \implies \langle K, e_2 \rangle \Downarrow)$$

$$e_1 \equiv_{ciu} e_2 \overset{\text{def}}{=} e_1 \leq_{ciu} e_2 \wedge e_2 \leq_{ciu} e_1$$

We extend these concepts to open expressions with closing substitutions.

$$e_1 \leq^{\Gamma}_{ciu} e_2 \overset{\text{def}}{=} \Gamma \vdash_{exp} e_1 \wedge \Gamma \vdash_{exp} e_2 \wedge \forall \sigma : \Gamma \vdash_{sub} \sigma \multimap \emptyset \implies e_1[\sigma] \leq_{ciu} e_2[\sigma]$$

$$e_1 \equiv^{\Gamma}_{ciu} e_2 \overset{\text{def}}{=} e_1 \leq^{\Gamma}_{ciu} e_2 \wedge e_2 \leq^{\Gamma}_{ciu} e_1$$

Proving expressions CIU equivalent is usually simpler than proving them contextually equivalent, since contextual equivalence potentially requires induction on the contexts [40]. In addition, a proof of two expressions being CIU equivalent requires to reason about only one frame stack and one substitution, in contrast to logical relation which requires related pairs of frame stacks and substitutions [9]. To maintain these related pair of expressions and frame stack would require additional care. On the other hand, for the same reason, logical relations can be more flexible while proving properties about the relations.

After defining the CIU preorder, we also proved its correspondence with the logical relations (see [32]):

Theorem 5.8 *(CIU coincides with the logical relations). For all expressions e_1, e_2, and scopes Γ, $e_1 \leq^{\Gamma}_{ciu} e_2$ if and only if $(e_1, e_2) \in \mathbb{E}^{\Gamma}$.*

Proof. We follow the techniques of Wand et al. [9] in this proof.

$\Rightarrow$: We prove $(e_1, e_2) \in \mathbb{E}^{\Gamma}$ and $e_2 \leq^{\Gamma}_{ciu} e_3$ implies $(e_1, e_3) \in \mathbb{E}^{\Gamma}$, which is a trivial consequence of the definitions. Thereafter, we prove our goal by using $(e_1, e_1) \in \mathbb{E}^{\Gamma}$ as the first premise of this helper statement by the fundamental property (Theorem 5.7).

$\Leftarrow$: The closing substitution required by the CIU preorder is denoted by σ. We specialize Definition 5.4 of $\mathbb{E}^{\Gamma}$ with $\sigma_1 = \sigma_2 = \sigma$, and by the fundamental property (Theorem 5.7), $(\sigma, \sigma) \in \mathbb{G}^{\Gamma}_n$. Thereafter, we just use Definition 5.4 of $\mathbb{E}_n$ to finish the proof. □

5.4. Example simple equivalences

We also proved a number of simple programs to be CIU equivalent. We show the proof sketch of the last one, but omit the others and refer to the formalisation [32]. The first equivalence is special, because it will be used in the proofs for the equality of the equivalence relations. In the examples we suppose that all meta-variables are implicitly quantified, but omit this detail for readability.

Example 5.9 *(Beta reduction 1).*

$$\Gamma \cup \{x\} \vdash_{exp} e \wedge \Gamma \vdash_{val} v \implies e[x \mapsto v] \equiv^{\Gamma}_{ciu} \texttt{let}\ x = v\ \texttt{in}\ e$$

The next example is a generalised version of the beta-reduction. We note that in this case, from the application we cannot reach the substituted expression in one reduction step, but in $k+2$, because all sub-values are evaluated in one step.

Example 5.10 *(Beta reduction 2).*

$$\Gamma \vdash_{val} \texttt{fun}\ f/k(x_1, \dots, x_k) \to e \wedge \Gamma \vdash_{val} v_1 \wedge \dots \wedge \Gamma \vdash_{val} v_k \implies$$
$$e[f/k \mapsto \texttt{fun}\ f/k(x_1, \dots, x_k) \to e, x_1 \mapsto v_1, \dots, x_k \mapsto v_k] \equiv^{\Gamma}_{ciu} \texttt{apply}\ (\texttt{fun}\ f/k(x_1, \dots, x_k) \to e)(v_1, \dots, v_k)$$

Corollary 5.11 *(Expressions are equivalent to their values). For all closed expressions e, and values v, if $\langle \varepsilon, e \rangle \longrightarrow^* v$, then $e \equiv_{ciu} v$.*

Proof. This theorem is a consequence of Theorem 3.3, and the fact that termination can be expressed in the semantics too (Theorem 5.1). □

Finally, we prove that the expressions in Fig. 2a and 2b are equivalent for particular parameters. We omit the function definitions here for readability, but they are shown in the figures mentioned before.

Example 5.12 *(Fold-map equivalence). For all closed values v_l, v_f, supposing that v_l is a proper list and v_f computes a meta-level function f (for these definitions, we refer to Section 3.5.2), we can prove that*

```
rec 'map'/2 = ... in apply 'map'/2(vf, vl) ≡ciu
  rec 'foldr'/3 = ... in apply 'foldr'/3(fun(X, A) -> [apply vf(X) | A], [], vl)
```

Proof. A general way to prove such equivalences is to prove that both expressions evaluate to the same value v. We refer to Lemma 3.4 for the evaluation of `'foldr'`, and to the formalisation for the evaluation of `'map'`. Next, we can use Corollary 5.11 to establish that both expressions are equivalent to v. Finally, the CIU preorder relations are also reflexive and transitive (we refer to Theorem 5.14 in the next section, and note that the contextual preorder is transitive by definition). □

5.5. Revisiting contextual preorder and equivalence

We describe a refined contextual equivalence relation based on the definitions by Wand et al. [9] and Gordon et al. [23].

Definition 5.6 *(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 Theorem 5.6.

We also adjusted our previous notion of contextual preorder and equivalence. In this case the context "closes" the potentially open expressions.

Definition 5.7 *(Syntax-based contextual preorder and equivalence).*

$$e_1 \leq^\Gamma_{ctx} e_2 \overset{\text{def}}{=} \Gamma \vdash_{exp} e_1 \wedge \Gamma \vdash_{exp} e_2 \wedge (\forall (C : Context) : \emptyset \vdash_{exp} C[e_1] \wedge \emptyset \vdash_{exp} C[e_2] \implies \langle \varepsilon, C[e_1] \rangle \Downarrow \implies \langle \varepsilon, C[e_2] \rangle \Downarrow)$$
$$e_1 \equiv^\Gamma_{ctx} e_2 \overset{\text{def}}{=} e_1 F e_2 \wedge e_2 \leq^\Gamma_{ctx} e_1$$

Next, we proved that $\leq^\Gamma_{ctx}$ (from Definition 5.7) satisfies the criteria of being a contextual preorder.

Theorem 5.13 *(Syntax-based contextual preorder is a contextual preorder). The relation $\leq^\Gamma_{ctx}$ satisfies the criteria of Definition 5.6.*

After defining the contextual preorder properly, we could prove the equality between $\leq^\Gamma_{ciu}$ and $\leq^\Gamma_{ctx}$, stated in Theorem 5.14 and in Theorem 5.16.

Theorem 5.14 *(CIU is a contextual preorder). For all expressions e_1, e_2, and scopes Γ, if $e_1 \leq^\Gamma_{ciu} e_2$, then $e_1 \leq^\Gamma_{ctx} e_2$.*

Proof. This theorem is just a consequence of the compatibility of the logical relations (Theorem 5.6), which coincide with CIU (Theorem 5.8). Only the proof of transitivity requires simple reasoning in first-order logic. □

Next, we prepare a lemma for the theorem that states that contextual equivalence implies CIU equivalence. This theorem states that contextually equivalent, open expressions can be closed step-by-step and we still obtain contextually equivalent expressions.

Lemma 5.15 *(Contextual equivalence is closed under substitution). For all expressions e_1, e_2, values v, names x, and scopes Γ, if $e_1 \leq^{\Gamma \cup \{x\}}_{ctx} e_2$ and $\Gamma \vdash_{val} v$, then $e_1[x \mapsto v] \leq^\Gamma_{ctx} e_2[x \mapsto v]$.*

Proof. This lemma is a consequence of Example 5.9 with the expressions `let` $x = v$ `in` e_1 and `let` $x = v$ `in` e_2, transitivity, and the fact that CIU equivalence implies contextual equivalence (Theorem 5.14). □

Theorem 5.16 *(CIU is the greatest contextual preorder). For all expressions e_1, e_2, and scopes Γ, if $e_1 \leq^\Gamma_{ctx} e_2$, then $e_1 \leq^\Gamma_{ciu} e_2$.*

Proof. We follow the idea of Wand et al. [9]. We carry out induction by the size of Γ.

- If $\Gamma = \emptyset$, both e_1 and e_2 are closed expressions, that is, we need to prove $e_1 \leq_{ciu} e_2$: for any closed frame stack K, $\langle K, e_1\rangle \Downarrow \Longrightarrow \langle K, e_2\rangle \Downarrow$. We do induction by the structure of K.
  - If $K = \varepsilon$, then we just use the fact of adequacy of $e_1 \leq^{\emptyset}_{ctx} e_2$ with the empty context to prove $\langle \varepsilon, e_1\rangle \Downarrow \Longrightarrow \langle \varepsilon, e_2\rangle \Downarrow$.
  - If $K = F :: K'$, then we apply Theorem 5.2 to the hypothesis, while Theorem 5.3 to the goal to be able to apply the induction hypothesis. Now only remains $F[e_1] \leq^{\emptyset}_{ctx} F[e_2]$ to prove. After separating cases by the structure of F, we can apply the compatibility properties of $\leq^{\emptyset}_{ctx}$ to finish the proof.
- If $\Gamma = \Gamma' \cup \{x\}$, we need to prove that for every $\Gamma' \cup \{x\} \vdash_{sub} \sigma \multimap \emptyset$, $e_1[\sigma] \leq_{ciu} e_2[\sigma]$. We can also assume that $x \notin \Gamma'$. We can divide σ into two parts: $e_1[x \mapsto \sigma(x)][\sigma \setminus \{x\}]$. Now we can apply the induction hypothesis, and the only remaining goal is $e_1[x \mapsto \sigma(x)] \leq_{ctx} e_2[x \mapsto \sigma(x)]$ which is proven by Lemma 5.15. $\square$

Putting Theorem 5.14 and Theorem 5.16 together, we prove the coincidence of the CIU and contextual equivalence.

Theorem 5.17 *(CIU theorem). For all expressions e_1, e_2, and scopes Γ, $e_1 \leq^{\Gamma}_{ciu} e_2$ if and only if $e_1 \leq^{\Gamma}_{ctx} e_2$.*

5.6. Revisiting behavioural equivalence

While defining the logical relations, CIU, and contextual equivalence we used only a termination criterion. But why is termination sufficient for the results of the evaluation to be equivalent? What would it mean for two expressions or values to be equivalent? We can take the definition of naive behavioural equivalence, and improve it so that it does not distinguish different function values. For this purpose, we define the equivalence of functions in an application-indexed way, that is equivalent functions should evaluate to the same values after the same number (n) of applications, over a limit.

Definition 5.8 *(Behavioural preorder).*

$$e_1 \leq^{R} e_2 \overset{\text{def}}{=} \forall v_1, K : K \text{ is closed } \Longrightarrow \langle K, e_1\rangle \longrightarrow^* v_1 \Longrightarrow \exists v_2 : \langle K, e_2\rangle \longrightarrow^* v_2 \wedge (v_1, v_2) \in R$$

$$v_1 \leq^{val}_0 v_2 \overset{\text{def}}{=} true$$

$$a_1 \leq^{val}_{(1+n')} a_2 \overset{\text{def}}{=} a_1 = a_2$$

$$i_1 \leq^{val}_{(1+n')} i_2 \overset{\text{def}}{=} i_1 = i_2$$

$$\texttt{[]} \leq^{val}_{(1+n')} \texttt{[]} \overset{\text{def}}{=} true$$

$$\texttt{[}v_1 \,|\, v_2\texttt{]}^{v} \leq^{val}_{(1+n')} \texttt{[}v_1' \,|\, v_2'\texttt{]}^{v} \overset{\text{def}}{=} v_1 \leq^{val}_{n'} v_1' \wedge v_2 \leq^{val}_{n'} v_2'$$

$$\begin{aligned}&\texttt{fun}\ f/k(x_1,\dots,x_k) \to e_1 \leq^{val}_{(1+n')} \texttt{fun}\ f/k(x_1,\dots,x_k) \to e_2 \overset{\text{def}}{=}\\ &\quad (\forall v_1,\dots,v_k : \emptyset \vdash_{val} v_1 \wedge \dots \wedge \emptyset \vdash_{val} v_k \Longrightarrow\\ &\qquad e_1[f/k \mapsto \texttt{fun}\ f/k(x_1,\dots,x_k) \to e_1, x_1 \mapsto v_1,\dots,x_k \mapsto v_k] \leq^{\leq^{val}_{n'}}\\ &\qquad e_2[f/k \mapsto \texttt{fun}\ f/k(x_1,\dots,x_k) \to e_2, x_1 \mapsto v_1,\dots,x_k \mapsto v_k])\end{aligned}$$

We say that two values v_1, v_2 behave the same way ($v_1 \leq^{val} v_2$, note that this is only a preorder relation), when $\forall n : v_1 \leq^{val}_n v_2$. Two expressions are equivalent ($e_1 \approx e_2$), if $e_1 \leq^{\leq^{val}} e_2 \wedge e_2 \leq^{\leq^{val}} e_1$.

Next, we prove that behavioural equivalence coincides with CIU equivalence.

Theorem 5.18 *(Behavioural equivalence coincides with CIU). For all closed expressions e_1, e_2, $e_1 \approx e_2$ if and only if $e_1 \equiv_{ciu} e_2$.*

Proof. $\Rightarrow$: Since termination can be expressed with the semantics (Theorem 5.1), this direction is just a simple consequence of the definitions.

$\Leftarrow$: Since we have $\langle K, e_1\rangle \longrightarrow^* v_1$ for some v_1 value, we can show that $\langle K, e_2\rangle \longrightarrow^* v_2$ for some v_2 by $e_1 \equiv_{ciu} e_2$ and Theorem 5.1. We only need to prove that $\forall n : v_1 \leq^{val}_n v_2$.

We carry out induction on n. The case $n = 0$ is *true* by definition. For the case $n = 1 + n'$, we show the induction hypothesis:

$$\forall K, v_1, v_2 : \langle K, e_1\rangle \longrightarrow^* v_1 \Longrightarrow \langle K, e_2\rangle \longrightarrow^* v_2 \Longrightarrow v_1 \leq^{val}_{n'} v_2$$

Now we do case distinction on v_1 and v_2. If both values were empty lists, equal atoms, or integers, then they are equivalent by definition.

Inequivalent values If the structure of v_1 and v_2 differ (e.g. v_1 is an integer, v_2 is a function, etc.), or if they are different atoms or integers, we construct a contradiction from the hypothesis $e_1 \equiv_{ciu} e_2$. If v_1 is not a function value, we can use the following idea: if $e_1 \equiv_{ciu} e_2$, then $\langle K', e_1\rangle \Downarrow$ implies $\langle K', e_2\rangle \Downarrow$ for any closed K'. We choose $K' = K +\!\!+ [\texttt{case} \ \square \ \texttt{of} \ p_{v_1} \ \texttt{then} \ 0 \ \texttt{else} \ \Omega]$, where Ω denotes the diverging expression $\texttt{apply (fun f/0()} \to \texttt{apply f/0())()}$, and p_{v_1} is the pattern that has the same structure as v_1, thus v_1 will match it, while v_2 will not. We note that because Theorems 3.3, 5.4, 5.5 the evaluation of e_1 to v_1 in K' consumes K from the stack (similarly for e_2 and v_2).

Since v_1 and v_2 were constructed differently, and $\langle K', e_1\rangle \Downarrow$ holds (for the K' above), therefore, $\langle K', e_2\rangle \Downarrow$ should also hold (because $e_1 \equiv_{ciu} e_2$), however this is a divergent configuration, because the pattern matching fails, and Ω does not terminate, so we got a contradiction.

If v_1 is a function value, while v_2 is not, we can use the same idea for the other part of $e_1 \equiv_{ciu} e_2$, i.e. $\langle K', e_2\rangle \Downarrow$ implies $\langle K', e_1\rangle \Downarrow$ with v_2.

Functions If $v_1 = \texttt{fun} \ f/k(x_1, \dots, x_k) \to b_1$ and $v_2 = \texttt{fun} \ f/k(x_1, \dots, x_k) \to b_2$, for readability, first we introduce two notations $fun_1 \overset{\text{def}}{=} \texttt{fun} \ f/k(x_1, \dots, x_k) \to b_1$ and $fun_2 \overset{\text{def}}{=} \texttt{fun} \ f/k(x_1, \dots, x_k) \to b_2$, and later redefine v_1 and v_2.

We need to prove that the bodies of these functions behave the same way when substituting their parameters to equal values. That is, in any closed frame stack K_2, for any closed values $v_1, \dots v_k, v$, $\langle K_2, b_1[f/k \mapsto fun_1, x_1 \mapsto v_1, \dots x_k \mapsto v_k]\rangle \longrightarrow^* v$ implies $\exists v' : \langle K_2, b_2[f/k \mapsto fun_2, x_1 \mapsto v_1, \dots x_k \mapsto v_k]\rangle \longrightarrow^* v'$ and $v \leq_{n'}^{val} v'$.

Now we connect the hypotheses, since $\langle K, e_1\rangle \longrightarrow^* fun_1$ and $\langle K_2, b_1[f/k \mapsto fun_1, x_1 \mapsto v_1, \dots x_k \mapsto v_k]\rangle \longrightarrow^* v$ through Theorem 3.3 and Theorem 3.2 to obtain: $\langle K +\!\!+ [\texttt{apply} \ \square(v_1, \dots, v_k)] +\!\!+ K_2, e_1\rangle \longrightarrow^* v$.

By $e_1 \equiv_{ciu} e_2$, we also prove that for some v', $\langle K +\!\!+ [\texttt{apply} \ \square(v_1, \dots, v_k)] +\!\!+ K_2, e_2\rangle \longrightarrow^* v'$. From this hypothesis, after taking some reduction steps (by $\langle K, e_2\rangle \longrightarrow^* fun_2$ and Theorem 3.3 and Theorem 3.2), we can prove $\langle K_2, b_2[f/k \mapsto fun_2, x_1 \mapsto v_1, \dots x_k \mapsto v_k]\rangle \longrightarrow^* v'$. We only need to prove that $v \leq_{n'}^{val} v'$, which is done by applying the induction hypothesis, moreover, its premises ($\langle K +\!\!+ [\texttt{apply} \ \square(v_1, \dots, v_k)] +\!\!+ K_2, e_1\rangle \longrightarrow^* v$ and $\langle K +\!\!+ [\texttt{apply} \ \square(v_1, \dots, v_k)] +\!\!+ K_2, e_2\rangle \longrightarrow^* v'$) have already been proved.

Lists If $v_1 = [v_{11} \,|\, v_{12}]^{\,v}$ and $v_2 = [v_{21} \,|\, v_{22}]^{\,v}$, then we need to show that $v_{11} \leq_{n'}^{val} v_{21}$ and $v_{12} \leq_{n'}^{val} v_{22}$. We can do that by applying the induction hypothesis twice, but there are still some evaluations to show (note that we have $\langle K, e_1\rangle \longrightarrow^* [v_{11} \,|\, v_{21}]^{\,v}$ and $\langle K, e_2\rangle \longrightarrow^* [v_{12} \,|\, v_{22}]^{\,v}$):

- We can show $\langle K', e_1\rangle \longrightarrow^* v_{11}$ implies $\langle K', e_2\rangle \longrightarrow^* v_{12}$ for $K' = K +\!\!+ [\texttt{case} \ \square \ \texttt{of} \ [\texttt{X}\,|\,\texttt{Y}] \ \texttt{then} \ \texttt{X} \ \texttt{else} \ 0]$ (which selects the *head* of the list).
- We can show $\langle K', e_1\rangle \longrightarrow^* v_{21}$ implies $\langle K', e_2\rangle \longrightarrow^* v_{22}$ for $K' = K +\!\!+ [\texttt{case} \ \square \ \texttt{of} \ [\texttt{X}\,|\,\texttt{Y}] \ \texttt{then} \ \texttt{Y} \ \texttt{else} \ 0]$ (which selects the *tail* of the list). □

6. Conclusion and future work

In this paper, 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. Initially we investigated and formalised simple behavioural equivalence [7] in Coq, but this turned out not to be expressive enough since it characterised equivalence as structural rather than semantic.

To solve this issue, in this paper we formalised contextual, CIU preorder and equivalence together with logical relations [8,9,25,29]. We discussed that logical relations are suitable for proving properties, CIU equivalence is best used for concrete expression equivalence proofs, while the verification of refactorings can be based on contextual equivalence and its congruence property. With these equivalences, we are able to prove non structurally-equivalent functions equivalent when they have the same behaviour, and we also proved the equivalence of two higher-order functions. Moreover, we also presented a proof that reasoning about termination is sufficient to characterise equivalence by giving a formal definition of behavioural equivalence which is also proved to coincide with the other definitions. Our definitions and results are formalised in the Coq [32] proof assistant.

Future work Currently, we are working on the extension of the semantics with the formalisation of concurrent features of (Core) Erlang, for example asynchronous message passing, process creation and termination, process supervision. We are also extending the semantics with other sequential concepts (e.g. tuples, exceptions, sequential side effects) too, based on our previous work [5,34]. In our earlier *ad hoc* equivalence definitions we used complete and weak equivalence of the results, where complete equivalence required the same side effects to resolve in the same order during evaluation, while weak equivalence allowed this order to be different for the two expression evaluation. It is therefore a future goal for us to investigate such “weaker” definitions of equivalences too.

In the medium and longer term, we plan to 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.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We have shared the link to the source code in the article.

Acknowledgements

The project has been supported by ÚNKP-20-4 and ÚNKP-21-3 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.

Appendix A. Scoping

Based on scoping (Section 3.4), a number of theorems can be proven about substitution; we highlight here the important ones on which either our equivalence results depend, or they are fundamental properties that should hold for substitution and scoping. The omitted ones are technical lemmas mainly parametrised by induction hypotheses to enable arguing about lists of expressions and values, while others prove the scope of some concrete substitutions. For the proofs and the omitted lemmas we refer to the formalisation [32].

In some theorems, we needed to prove the same (or similar) properties for both expressions and values due to mutual induction. For instance, when proving a general statement on function *values*, the induction hypothesis requires the statement to hold on the body *expression* of the function. Here, we only provide proof sketches, and refer to the complete proof in the formalisation [32].

The first theorem states that an updated substitution maps a larger set of names to values in the same scope.

Theorem A.1 *(Scoping of extended substitutions). For all expressions v, substitutions σ, scopes Γ, Δ, if $\Delta \vdash_{val} v$ and $\Gamma \vdash_{sub} \sigma \multimap \Delta$, then for all names $x \notin \Gamma$, $\Gamma \cup \{x\} \vdash_{sub} \sigma\{x \mapsto v\} \multimap \Delta$.*

Next, we show that a scope of an expression (or value) can be extended, thus a scope can contain any names that are not free in the expression (or value).

Theorem A.2 *(Scope extension (weakening)). For all expressions e, scopes Γ, Δ, if $\Gamma \vdash_{exp} e$ then $\Gamma \cup \Delta \vdash_{exp} e$. The same property holds for the value judgement.*

Thereafter, we show a theorem about the scopes of substitutions. If we restrict a substitution (i.e. remove bindings from it), then the removed names will appear in the scope of this new substitution, since they become mapped to themselves.

Theorem A.3 *(Scoping of restricted substitutions). For all substitutions σ, scopes Γ, Δ, if $\Gamma \vdash_{sub} \sigma \multimap \Delta$ then for all names $x_1, \dots, x_k$, $(\Gamma \cup \{x_1, \dots, x_k\}) \vdash_{sub} (\sigma \setminus \{x_1, \dots, x_k\}) \multimap (\Delta \cup \{x_1, \dots, x_k\})$.*

Now we define the set of names that are not modified by a substitution, i.e. these are mapped to themselves.

Definition A.1 *(Substitution identities).* We say that a substitution σ preserves a scope, if for all names $x \in \Gamma$, $\sigma(x) = x$.

In a restricted substitution, any names that are removed from the substitution will be mapped to themselves, i.e. they are preserved by the restricted substitution.

Theorem A.4 *(Restriction of substitution identities). For all substitutions σ, scopes Γ, if σ preserves Γ, then for all names $x_1, \dots, x_k$, $preserves(\Gamma \cup \{x_1, \dots, x_k\}, \sigma \setminus \{x_1, \dots, x_k\})$.*

If we apply a preserving substitution to an expression, which has the same scope as the preserved names, the substitution does not modify the expression.

Theorem A.5 *(Preserving substitution is identity). For all expressions e, substitutions σ, scopes Γ, if σ preserves Γ, and $\Gamma \vdash_{exp} e$, then $e[\sigma] = e$. The same property holds for values too.*

With the help of the previous lemma, we can show that closed expressions are not modified by any substitutions, since every substitution preserves the empty set of names.

Corollary A.6 *(Closed expressions are not modified by substitutions). For all expressions e, substitutions σ, if $\emptyset \vdash_{exp} e$, then $e[\sigma] = e$. The same property holds for values too.*

Scoping of values (and expressions) can be combined with the scoping of substitutions: applying a scoped substitution on a scoped value (or expression) keeps it scoped, and conversely, a substituted value (or expression) can be shown to be scoped without the substitution.

Theorem A.7 *(Substitution preserves scoping). For all expressions e, substitutions σ, scopes Γ, Δ, if $\Gamma \vdash_{exp} e$ and $\Gamma \vdash_{sub} \sigma \multimap \Delta$, then $\Delta \vdash_{exp} e[\sigma]$. The same property holds for values too.*

Theorem A.8 *(Substitution implies scoping). For all expressions e, scopes Γ, Δ, if for all substitutions σ, $\Gamma \vdash_{sub} \sigma \multimap \Delta$ implies $\Delta \vdash_{exp} e[\sigma]$, then $\Gamma \vdash_{exp} e$. The same property holds for values too.*

Appendix B. Supplementary material for Section 3.5

In this section, we provide the proofs for two properties of the frame stack semantics. First, we show the proof of determinism.

Proof of Theorem 3.1. We only give the sketch of the proof here. We proceed with case distinction on the two reductions. If different reduction rules were used, then we can find a contradiction between their premises. If the same rules were used, then the result configurations are the same. □

Next we show the proof that extending the frame stack does not affect the evaluation.

Proof of Theorem 3.3. We carry out induction on the length of the derivation (n).

- For $n = 0$, from $\langle K_1, e_1 \rangle \longrightarrow^0 \langle K_2, e_2 \rangle$ we acquire $K_1 = K_2$ and $e_1 = e_2$, from which the conclusion follows by the definition of $\longrightarrow^0$.
- For $n = 1 + n'$, we inspect the possible derivations of $\langle K_1, e_1 \rangle \longrightarrow^{1+n'} \langle K_2, e_2 \rangle$, and just take the same step in the conclusion together with the induction hypothesis (if necessary). □

Appendix C. Supplementary material for Section 5.4

In this section, we show theorems and proofs of program equivalence. The first proof is about Example 5.9.

Proof of Example 5.9. Since we prove an equivalence, it means two preorders. We also need to prove a number of closedness properties, which can be done by the hypotheses and the lemmas in Section 3.3, we leave these to the reader.

- First, we need to prove the following: for any closed frame stack K and closing substitution σ (i.e. $\Gamma \vdash_{sub} \sigma \multimap \emptyset$), $\langle K, e[x \mapsto v][\sigma] \rangle \Downarrow$ implies $\langle K, \texttt{let}\ x = v[\sigma]\ \texttt{in}\ e[\sigma] \rangle \Downarrow$. Let us assume that $\langle K, e[x \mapsto v][\sigma] \rangle \Downarrow^k$ for a step-index k. We can show that $\langle K, \texttt{let}\ x = v[\sigma]\ \texttt{in}\ e[\sigma] \rangle \Downarrow^{2+k}$ by definition. If we make these two steps, we get $\langle K, e[\sigma \setminus x][x \mapsto v[\sigma]] \rangle \Downarrow^k$, and by the properties of capture-avoiding substitution, $e[\sigma \setminus x][x \mapsto v[\sigma]] = e[x \mapsto v][\sigma]$ (we refer to the formalisation for more details [32]).
- Next, we need to prove the following: for any closed frame stack K and closing substitution σ (i.e. $\Gamma \vdash_{sub} \sigma \multimap \emptyset$), $\langle K, \texttt{let}\ x = v[\sigma]\ \texttt{in}\ e[\sigma] \rangle \Downarrow$ implies $\langle K, e[x \mapsto v][\sigma] \rangle \Downarrow$. Now we inspect the premise $\langle K, \texttt{let}\ x = v[\sigma]\ \texttt{in}\ e[\sigma] \rangle \Downarrow$, and conclude that by definition $\langle K, e[\sigma \setminus x][x \mapsto v[\sigma]] \rangle \Downarrow^k$ should hold for some k. This k is suitable for the derivation in the goal ($\langle K, e[x \mapsto v][\sigma] \rangle \Downarrow^k$), which is identical to this premise when we use the previous thought about the equality of the substitutions $e[\sigma \setminus x][x \mapsto v[\sigma]] = e[x \mapsto v][\sigma]$. □

Next, we also show another version of beta-reduction, when we suppose that the formal parameters are not used in the function's body.

Example C.1.

$$\emptyset \vdash_{val} v_1 \wedge \ldots \emptyset \vdash_{val} v_k \wedge \Gamma \vdash_{exp} e \wedge x_1, \ldots, x_k \notin \Gamma \implies e \equiv^{\Gamma}_{ciu} \texttt{apply}\ (\texttt{fun}\ f/k(x_1, \ldots, x_k) \to e)(v_1, \ldots, v_k)$$

The following equivalence (the commutativity of addition) is also special to us: in our current language (which is a simplified variant of sequential Core Erlang) it holds, but with side effects and exceptions added, other preconditions will be needed to prove it.

Example C.2 *(Commutativity of addition).*

$$\Gamma \vdash_{exp} e_1 \wedge \Gamma \vdash_{exp} e_2 \implies \texttt{call '+'}(e_1, e_2) \equiv^{\Gamma}_{ciu} \texttt{call '+'}(e_2, e_1)$$

References

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., USA, ISBN 0201485672, 1999.
[2] D. Horpácsi, J. Kőszegi, Z. Horváth, Trustworthy refactoring via decomposition and schemes: a complex case study, arXiv:1708.07225, https://doi.org/10.4204/EPTCS.253.8, 2017.
[3] D. Horpácsi, J. Kőszegi, S. Thompson, Towards trustworthy refactoring in Erlang, arXiv:1607.02228, https://doi.org/10.4204/EPTCS.216.5, 2016.
[4] P. Bereczky, D. Horpácsi, S. Thompson, A proof assistant based formalisation of a subset of sequential Core Erlang, in: A. Byrski, J. Hughes (Eds.), Trends in Functional Programming, Springer International Publishing, Cham, 2020, pp. 139–158.
[5] P. Bereczky, D. Horpácsi, S.J. Thompson, Machine-checked natural semantics for Core Erlang: exceptions and side effects, in: Proceedings of Erlang 2020, ACM, 2020, pp. 1–13.
[6] P. Bereczky, D. Horpácsi, J. Kőszegi, S. Szeier, S. Thompson, Validating Formal Semantics by Property-Based Cross-Testing, ACM, New York, NY, USA, 2021.
[7] B.C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, B. Yorgey, Software foundations, https://softwarefoundations.cis.upenn.edu/, 2022. (Accessed 14 October 2022).
[8] A.M. Pitts, Operational semantics and program equivalence, in: G. Barthe, P. Dybjer, L. Pinto, J. Saraiva (Eds.), Applied Semantics, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 378–412.
[9] M. Wand, R. Culpepper, T. Giannakopoulos, A. Cobb, Contextual equivalence for a probabilistic language with continuous random variables and recursion, Proc. ACM Program. Lang. 2 (ICFP) (Jul. 2018), https://doi.org/10.1145/3236782.
[10] I. Lanese, N. Nishida, A. Palacios, G. Vidal, A theory of reversibility for Erlang, J. Log. Algebraic Methods Program. 100 (2018) 71–97, https://doi.org/10.1016/j.jlamp.2018.06.004.
[11] I. Lanese, N. Nishida, A. Palacios, G. Vidal, CauDEr: a causal-consistent reversible debugger for Erlang, in: J.P. Gallagher, M. Sulzmann (Eds.), International Symposium on Functional and Logic Programming, Springer, Springer International Publishing, Cham, 2018, pp. 247–263.
[12] N. Nishida, A. Palacios, G. Vidal, A reversible semantics for Erlang, in: M.V. Hermenegildo, P. Lopez-Garcia (Eds.), International Symposium on Logic-Based Program Synthesis and Transformation, Springer, Springer International Publishing, Cham, 2017, pp. 259–274.
[13] L.-Å. Fredlund, A framework for reasoning about Erlang code, Ph.D. thesis, Mikroelektronik och informationsteknik, 2001.
[14] G. Vidal, Towards symbolic execution in Erlang, in: A. Voronkov, I. Virbitskaite (Eds.), International Andrei Ershov Memorial Conference on Perspectives of System Informatics, Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 351–360.
[15] P. Bereczky, D. Horpácsi, S. Thompson, A comparison of big-step semantics definition styles, arXiv:2011.10373, 2020.
[16] S. Owens, M.O. Myreen, R. Kumar, Y.K. Tan, Functional big-step semantics, in: P. Thiemann (Ed.), Programming Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 589–615.
[17] A. Pitts, Operationally-based theories of program equivalence, in: Publications of the Newton Institute, Cambridge University Press, 1997, pp. 241–298.
[18] S. Abramsky, C. Ong, Full abstraction in the lazy lambda calculus, Inf. Comput. 105 (2) (1993) 159–267, https://doi.org/10.1006/inco.1993.1044.
[19] A. Simpson, N. Voorneveld, Behavioural equivalence via modalities for algebraic effects, ACM Trans. Program. Lang. Syst. 42 (1) (Nov. 2019), https://doi.org/10.1145/3363518.
[20] D.J. Howe, Proving congruence of bisimulation in functional programming languages, Inf. Comput. 124 (2) (1996) 103–112, https://doi.org/10.1006/inco.1996.0008.
[21] U. Dal Lago, D. Sangiorgi, M. Alberti, On coinductive equivalences for higher-order probabilistic functional programs, SIGPLAN Not. 49 (1) (2014) 297–308, https://doi.org/10.1145/2578855.2535872.
[22] I. Mason, C. Talcott, Equivalence in functional languages with effects, J. Funct. Program. 1 (3) (1991) 287–327, https://doi.org/10.1017/S0956796800000125.
[23] A.D. Gordon, P.D. Hankin, S.B. Lassen, Compilation and equivalence of imperative objects, J. Funct. Program. 9 (4) (1999) 373–426, https://doi.org/10.1007/BFb0058024.
[24] C. McLaughlin, J. McKinna, I. Stark, Triangulating context lemmas, in: CPP 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 102–114.
[25] R. Culpepper, A. Cobb, Contextual equivalence for probabilistic programs with continuous random variables and scoring, in: H. Yang (Ed.), Programming Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2017, pp. 368–392.
[26] A. Ahmed, Step-indexed syntactic logical relations for recursive and quantified types, in: P. Sestoft (Ed.), Programming Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 69–83.
[27] L. Birkedal, A. Bizjak, J. Schwinghammer, Step-indexed relational reasoning for countable nondeterminism, Log. Methods Comput. Sci. 9 (4) (Oct. 2013), https://doi.org/10.2168/LMCS-9(4:4)2013.
[28] N. Benton, A. Kennedy, Monads, effects and transformations, in: hOOTS '99, Higher Order Operational Techniques in Semantics, in: Electronic Notes in Theoretical Computer Science, vol. 26, 1999, pp. 3–20.
[29] A.M. Pitts, Step-indexed biorthogonality: a tutorial example, in: A. Ahmed, N. Benton, L. Birkedal, M. Hofmann (Eds.), Modelling, Controlling and Reasoning About State, in: Dagstuhl Seminar Proceedings (DagSemProc), vol. 10351, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2010, pp. 1–10.
[30] D. Lucanu, V. Rusu, Program equivalence by circular reasoning, in: E.B. Johnsen, L. Petre (Eds.), Integrated Formal Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 362–377.
[31] Ş. Ciobâcă, D. Lucanu, V. Rusu, G. Roşu, A language-independent proof system for mutual program equivalence, in: S. Merz, J. Pang (Eds.), Formal Methods and Software Engineering, Springer International Publishing, Cham, 2014, pp. 75–90.
[32] Core Erlang mini, https://github.com/harp-project/Core-Erlang-mini/releases/tag/v1.4, 2022. (Accessed 14 October 2022).
[33] N. de Bruijn, Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem, Indag. Math. (Proceedings) 75 (5) (1972) 381–392, https://doi.org/10.1016/1385-7258(72)90034-0.
[34] Core Erlang formalization, https://github.com/harp-project/Core-Erlang-Formalization, 2022. (Accessed 20 September 2022).

[35] P.D. Mosses, Formal semantics of programming languages: — an overview, in: Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004), Electron. Notes Theor. Comput. Sci. 148 (1) (2006) 41–73, https://doi.org/10.1016/j.entcs.2005.12.012.
[36] M. Neuhäußer, T. Noll, Abstraction and model checking of Core Erlang programs in Maude, in: Proceedings of the 6th International Workshop on Rewriting Logic and Its Applications (WRLA 2006), Electron. Notes Theor. Comput. Sci. 176 (4) (2007) 147–163, https://doi.org/10.1016/j.entcs.2007.06.013.
[37] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, R. Virding, Core Erlang 1.0.3 language specification, Tech. rep., https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf, 2004. (Accessed 14 October 2022).
[38] I. Lanese, D. Sangiorgi, G. Zavattaro, Playing with bisimulation in Erlang, in: M. Boreale, F. Corradini, M. Loreti, R. Pugliese (Eds.), Models, Languages, and Tools for Concurrent and Distributed Programming, Springer, Cham, 2019, pp. 71–91.
[39] A.M. Pitts, I.D. Stark, Operational reasoning for functions with local state, in: Higher Order Operational Techniques in Semantics, 1998, pp. 227–273.
[40] R. Ramanujam, V. Arvind, Foundations of Software Technology and Theoretical Computer Science, Springer, 1998.