Annales Univ. Sci. Budapest., Sect. Comp. 57 (2024) 117–141

A COMPARISON OF BIG-STEP SEMANTICS DEFINITION STYLES

Péter Bereczky and Dániel Horpácsi (Budapest, Hungary)
Simon Thompson (Budapest, Hungary, and Canterbury, United Kingdom)

Communicated by Zoltán Horváth
(Received January 9, 2024; accepted May 6, 2024)

Abstract. Formal semantics provides rigorous, mathematically precise definitions of programming languages, with which we can argue about program behaviour and program equivalence by formal means; in particular, we can describe and verify our arguments with a proof assistant. There are various approaches to giving formal semantics to programming languages, at different abstraction levels and applying different mathematical machinery. In this paper we investigate some of the approaches that share their roots with traditional relational big-step semantics, such as (a) functional big-step semantics, (b) pretty-big-step semantics and (c) traditional natural semantics. We compare these approaches with respect to the following criteria: executability of the semantics definition, proof complexity for essential mathematical properties and the conciseness of expression equivalence proofs. The comparison is based on formalisations of a sequential subset of Core Erlang, a simple functional programming language.

Key words and phrases: Formal semantics, natural semantics, pretty-big-step semantics, functional big-step semantics, Coq.
2010 Mathematics Subject Classification: 18C50.
This work was supported by the project "Integrált kutatói utánpótlás-képzési program az informatika és számítástudomány diszciplináris területein (Integrated program for training new generation of researchers in the disciplinary fields of computer science)", No. EFOP-3.6.3-VEKOP-16-2017-00002. The project has been supported by the European Union and co-funded by the European Social Fund.
"Application Domain Specific Highly Reliable IT Solutions" project has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme.

1. Introduction

This work is part of a wider project that aims to reason about the correctness of code refactoring. To this end, a rigorous, formal definition is needed for the programming language under refactoring: in our case, Erlang. In earlier work, we developed a relational big-step semantics for sequential Core Erlang, including language features such as exceptions and side effects. This semantics was used for proving characteristic properties (e.g. determinism) of the language, as well as showing the equivalence between pairs of pattern expressions. The latter are important from the refactoring point of view: pattern equivalences can be interpreted as simple, correct refactorings for Core Erlang [3, 4].

Formalising Core Erlang in the big-step operational definitional approach was a somewhat *ad hoc* decision, supported by the following facts: it is not as detailed as small-step definitions, offering shorter proofs, and, at the same time, unlike in denotational definitions, semantics and proofs of nondeterministic and divergent programs do not need special treatment in the proof assistant embedding. Nonetheless, relational big-step semantics comes with its drawbacks: in general, it is not directly executable, the proof of determinism is complex, and we cannot use this style of semantics to argue about concurrency. After working with the relational big-step semantics formalisation for a while, the shortcomings of this approach became apparent, and we decided to investigate whether other semantics definition styles would be more suitable.

Selecting the semantics definition style seems to be a simple choice: the purpose of defining the semantics should determine the applied definition approach. However, conflicting requirements can make the decision unclear. For instance, in related work, different approaches have been applied to reason about program transformations: Grigore *et al.* [10] and Garrido *et al.* [13] use (reduction style) small-step semantics, whilst Owens *et al.* [21] use (functional) big-step semantics. Both of these are executable and can be used to argue about program equivalence, but show different characteristics in general. There are a number of ways to create a testable and usable formal semantics, as, for example, addressed in a related discussion by Blazy and Leroy [5], but it is not obvious to tell which is the best option for our purposes. Moreover, this choice is not only about the different mathematical approaches, but also how easy it is to implement them in the Coq proof assistant.

This paper is a detailed version of our MaCS 2020 conference presentation, where we analyse and compare different methods of defining big-step style semantics for a small, Erlang-like programming language. We do this to answer the question of which method should be used when creating a semantic description of sequential Core Erlang, when the description should support equivalence proofs and be efficiently executable. In doing this we survey the

following methods: (a) traditional relational big-step semantics [14], (b) pretty-big-step semantics [8], and (c) functional big-step semantics [21], which can be seen as equivalent to supplying a definitional interpreter [23]. We also briefly discuss a coinductive approach to define big-step semantics [16]. When comparing semantic approaches, we aim to answer the following questions:

1. Does the semantics definition scale in terms of the complexity of expression equivalence proofs? Since our primary purpose is to prove expression pattern equivalences, the semantics has to be especially supportive of constructing such proofs.
2. Is the semantics effectively executable, allowing for automatic evaluation of expressions? Is this automatic execution efficient, with a performance comparable to a reference implementation? Execution of the semantics definition is crucial when it comes to validation: testing the semantics against a reference implementation needs the semantics to be executed.
3. How complex are the proofs for the common properties such as determinism or progress? For instance, some semantics are inherently deterministic, because they are presented as a semantic function, while it is a lot more cumbersome to prove this property in a relational semantics.

We note that the paper not only makes a survey of the abovementioned semantics definition styles, but implements a benchmark language in each of those, and makes the detailed comparison based on the case study. Namely, we make the following main contributions:

- Traditional big-step, pretty-big-step and functional big-step semantics definitions for a simple functional programming language, with proofs of their equivalence.
- Proofs of basic properties of each semantics and proofs for simple expression pattern equivalences (local refactorings) in each definition style.
- A systematic comparison of the approaches with respect to execution and proof complexity.

We will often quote Coq code to highlight the fact that all these concepts have been formalised in Coq [24]. Inductive constructors in the relational semantics are described as inference rules.

The rest of the paper is structured as follows. In Section 2 we describe the syntax, and necessary abstractions for our benchmark language. In Section 3 we discuss the traditional big-step and the pretty-big-step semantics, and in Section 4 we cover the functional approaches, in particular the functional big-step semantics. Section 5 evaluates the presented approaches, and also briefly summarises coinductive big-step semantics [16]. Finally, Section 6 concludes and discusses future work.

2. The benchmark language

Throughout the paper, we define formal semantics for a simple but representative, functional programming language, which resembles Erlang; in fact, our case study language is a proper subset of Core Erlang. In this section, we introduce the syntax and a semantic domain for the language, based on which the later sections will define big-step operational semantics of different styles in order to make a systematic comparison between them.

2.1. Syntax

The case study language includes abstractions known from the functional paradigm (such as single assignment variables, *let*-binding, lambda abstraction and function application), but we also incorporate impure expressions (such as I/O calls and exception handling). Furthermore, the language supports recursive function definitions (*letrec*-binding), but only one name can be bound by each expression. Figure 1 defines the syntax of the language precisely, as an inductive type.

```
Inductive Expression : Type :=
| ELit (l : Literal[a])
| EVar (v : Var)
| EFunId (f : FunctionIdentifier)
| EFun (vl : list Var) (e : Expression)
| ECall (f : string) (params : list Expression)
| EApp (exp : Expression) (params : list Expression)
| ELet (v : Var) (e b : Expression)
| ELetRec (fid : FunctionIdentifier) (params : list Var) (b e : Expression)
| ETry (e1 : Expression) (v : Var) (e2 : Expression)
        (vl : list Var) (e3 : Expression).
```

[a] Literals are either atoms or integers.

Figure 1: The syntax of our case study language (subset of Core Erlang)

2.2. Semantic domain

This language has expressions of three types: atoms, integers and functions. Therefore, values of expressions can only be literal values and closures (see Figure 2). Closures are the normal forms of functions, and store the function's parameter list, body expression and an evaluation environment in which the body should be evaluated; moreover, the collection of recursive functions defined simultaneously[1].

[1] The presented approach is based on our previous work and is fairly general: it can handle multiple simultaneous function definitions, not only one; see [4] for more details.

```
Definition FunctionExpression : Type := list Var × Expression.
Inductive Value : Type :=
| VLit (l : Literal)
| VClos (Γ' : Environment)
        (ext : list (FunctionIdentifier × FunctionExpression))
        (vl : list Var) (e : Expression).

Definition Exception := ExceptionClass × Value × Value.
```

Figure 2: Semantic domain

Expression evaluations may yield exceptions. In our formalisation, exceptions are represented as triples: exception class (`error`, `throw` or `exit`) and two values describing the exception reason. In our case studies, we will use two common Erlang examples: *badarity* happens when an application evaluation fails due to the mismatch in the number of formal and actual parameters, and *badfun* is encountered when the function expression of the application (i.e. the first subexpression in *EApp*) evaluates to a value that is not a function closure.

Finally, we define the semantic domain as the union of values and exception descriptions: $Value + Exception$. In the formalisation, we use Coq's built-in union type with the standard *inl* and *inr* constructors to make elements of the semantic domain.

2.3. Environment

In order to share as much as possible in the different semantics definitions, not only we fix the semantic domain, but we define a common type for the evaluation environment. Basically, this is a collection of variable names (and function identifiers) mapped to values (we use Γ to denote it). There are several helper functions to manage this environment, namely:

- $\Gamma[x]$: Returns the value associated with a given name x in Γ. If the name is unbound, it yields an exception.
- $\Gamma[xs \mapsto vs]$: Inserts bindings into Γ. The names (variables or function identifiers) in the list xs will be bound to the values in vs, pairwise. If there is only one binding (i.e. $\Gamma[[x] \mapsto [v]]$) we omit the parentheses of the lists: $\Gamma[x \mapsto v]$.
- *get_env* Γ *ext*: Returns the evaluation environment for the body of a closure based on its stored environment Γ and extension *ext*.
- To denote lists, we use standard list notations, i.e. $[x_1, \ldots, x_n]$ denotes the list consisting of the elements $x_1, \ldots, x_n$.

We remind the reader that the case study language allows for calling some built-in I/O functions, thus the semantics will need to address the meaning

of these side-effects. For this, we define a type (*SideEffectList*), the values of which log simple input-output effects produced by the evaluation of specific *ECall* expressions. While evaluating *ECall* expressions, we use the auxiliary *eval* function, which returns a value or exception and a side effect trace — only this operation can extend the side effect trace [2].

2.4. Evaluation criteria

As mentioned already in the introduction, we will compare the different approaches of defining big-step semantics based on the following criteria:

- How complex is proving the properties of the semantics. We will use the determinism property to investigate this.
- Is the approach executable? Is the semantics efficiently executable?
- How complex is proving expression evaluation formally. We will use two smaller expressions to investigate this:

```
let X = fun(Y,Z)->Y in apply X('a', 'b')

let X = 4 in let Y = 5 in apply (fun(X,Y)->X+Y) (X, Y)
```

Listing 1: Expression evaluation examples

- How complex is proving expression equivalence? We will use one unconditional and one conditional[3] equivalence to investigate this:

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

```
let A = e1 in                 let B = e2 in
  let B = e2 in      ⇔          let A = e1 in
    A + B                         A + B
```

Listing 2: Expression equivalence examples

3. Relational big-step semantics

A traditional big-step operational semantics is a relation between the evaluable expression and its value, or more generally, between initial and final configurations, where the configurations may include the evaluation environment or the side-effects of the evaluation. Note that in big-step style, the intermediate

[2] In our previous work [4] we applied a slightly different method in the formalised traditional big-step semantics using standard list append operations in every derivation rule; however, this former approach had to be refined in order to support automatic evaluation of expressions.

[3] In the second example, the side effects produced by e_1 and e_2 are swapped during the evaluation of these expressions.

stages of the evaluation are not visible from the relation [20]. The idea of this style of semantics is originated from Kahn [14]. In Coq, such a relation can be formalised with an inductive type, where the data constructors represent the derivation rules (or judgements).

3.1. Traditional relational big-step semantics

Traditional inductive big-step semantics are used in many projects, to mention but a few: deriving such a semantics from a small-step definition [9], call-by-need semantics of let and letrec calculus (λ_{let}, λ_{letrec}) [17], or the trace-based operational semantics for *While* [18] (this one is defined coinductively), as well as our project defining Core Erlang [3, 4, 19].

For the investigation of the different big-step definition styles, we reuse our Core Erlang formalisation mentioned above, but discard parts of it since the case study language used in the comparison is a subset of it. The big-step semantics will be denoted by $\langle \Gamma, exp, eff_1 \rangle \Downarrow \{res, eff_2\}$ where Γ is the evaluation environment, exp is the evaluable expression, eff_1 and eff_2 are the initial and final side effect traces and res is the result which is either a value or an exception. Before describing the semantics, we introduce some predicates and notations for readability about evaluating a list of expressions (we use $|l|$ to denote the length of list l, $S\ i$ denotes the successor of i and $l[i]$ denotes the ith element of l). The function $nth\_def\ l\ default\ i$ works the same way as $l[i-1]$ if $i > 0$, but for $i = 0$ it returns the *default* value. We also use Coq's standard *last* function [11].

$$\langle \Gamma, es, eff_1 \rangle \Downarrow^{all} \{vs, effs\} := (|es| = |vs|) \Rightarrow (|es| = |effs|) \Rightarrow (\forall j < |es|, \langle \Gamma, es[j], nth\_def\ effs\ eff_1\ j \rangle \Downarrow \{inl\ vs[j], nth\_def\ effs\ eff_1\ (S\ j)\})$$

$$\langle \Gamma, es, eff_1 \rangle \Downarrow^{up\ to:\ i} \{vs, effs\} := (i < |es|) \Rightarrow (|vs| = i) \Rightarrow (|effs| = i) \Rightarrow (\forall j < i, \langle \Gamma, es[j], nth\_def\ effs\ eff_1\ j \rangle \Downarrow \{inl\ vs[j], nth\_def\ effs\ eff_1\ (S\ j)\}).$$

The $\langle \Gamma, es, eff_1 \rangle \Downarrow^{all} \{vs, effs\}$ states that an expression list es evaluates to a value list vs (note that this formula also expresses that the evaluation of the expressions does not produce any exceptions). The evaluation for the jth step starts with the side effect log $effs[j-1]$ (or the default initial log eff_1) and the result log is $effs[j]$. The definition of $\langle \Gamma, es, eff_1 \rangle \Downarrow^{up\ to:\ i} \{vs, effs\}$ describes the same behaviour, but only for the first i elements. Now we can describe the big-step semantics for our case study language (Figure 3 shows the evaluation of expressions without exceptions, and Figure 4 explains the semantics of exceptions).

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

$$\text{(Lit)}\quad \frac{}{\langle \Gamma, \mathit{ELit}\ l, \mathit{eff}_1\rangle \Downarrow \{\mathit{inl}\ (\mathit{VLit}\ l), \mathit{eff}_1\}}$$

$$\text{(Var)}\quad \frac{\mathit{res} = \Gamma[\mathit{inl}\ s]}{\langle \Gamma, \mathit{EVar}\ s, \mathit{eff}_1\rangle \Downarrow \{\mathit{res}, \mathit{eff}_1\}}$$

$$\text{(FunId)}\quad \frac{\mathit{res} = \Gamma[\mathit{inr}\ \mathit{fid}]}{\langle \Gamma, \mathit{EFunId}\ \mathit{fid}, \mathit{eff}_1\rangle \Downarrow \{\mathit{res}, \mathit{eff}_1\}}$$

$$\text{(Fun)}\quad \frac{}{\langle \Gamma, \mathit{EFun}\ \mathit{vl}\ e, \mathit{eff}_1\rangle \Downarrow \{\mathit{inl}\ (\mathit{VClos}\ \Gamma\ \langle\rangle\ \mathit{vl}\ e), \mathit{eff}_1\}}$$

$$\text{(Call)}\quad \frac{\begin{array}{c}\langle \Gamma, \mathit{params}, \mathit{eff}_1\rangle \Downarrow^{\mathit{all}} \{\mathit{vals}, \mathit{effs}\} \\ \mathit{eval}\ \mathit{fname}\ \mathit{vals}\ (\mathit{last}\ \mathit{effs}\ \mathit{eff}_1) = (\mathit{res}, \mathit{eff}_2)\end{array}}{\langle \Gamma, \mathit{ECall}\ \mathit{fname}\ \mathit{params}, \mathit{eff}_1\rangle \Downarrow \{\mathit{res}, \mathit{eff}_2\}}$$

$$\text{(App)}\quad \frac{\begin{array}{c}\langle \Gamma, \mathit{params}, \mathit{eff}_2\rangle \Downarrow^{\mathit{all}} \{\mathit{vals}, \mathit{effs}\} \qquad |\mathit{var\_list}| = |\mathit{vals}| \\ \langle \Gamma, \mathit{exp}, \mathit{eff}_1\rangle \Downarrow \{\mathit{inl}\ (\mathit{VClos}\ \Gamma'\ \mathit{ext}\ \mathit{var\_list}\ \mathit{body}), \mathit{eff}_2\} \\ \langle (\mathit{get\_env}\ \Gamma'\ \mathit{ext})[\mathit{var\_list} \mapsto \mathit{vals}], \mathit{body}, \mathit{last}\ \mathit{effs}\ \mathit{eff}_2\rangle \Downarrow \{\mathit{res}, \mathit{eff}_3\}\end{array}}{\langle \Gamma, \mathit{EApp}\ \mathit{exp}\ \mathit{params}, \mathit{eff}_1\rangle \Downarrow \{\mathit{res}, \mathit{eff}_3\}}$$

$$\text{(Let)}\quad \frac{\langle \Gamma, e, \mathit{eff}_1\rangle \Downarrow \{\mathit{inl}\ \mathit{val}, \mathit{eff}_2\} \qquad \langle \Gamma[v \mapsto \mathit{val}], b, \mathit{eff}_2\rangle \Downarrow \{\mathit{res}, \mathit{eff}_3\}}{\langle \Gamma, \mathit{ELet}\ v\ e\ b, \mathit{eff}_1\rangle \Downarrow \{\mathit{res}, \mathit{eff}_3\}}$$

$$\text{(LetRec)}\quad \frac{\langle \Gamma[\mathit{fid} \mapsto \mathit{VClos}\ \Gamma\ [(\mathit{fid}, (\mathit{params}, b))]\ \mathit{params}\ b], e, \mathit{eff}_1\rangle \Downarrow \{\mathit{res}, \mathit{eff}_2\}}{\langle \Gamma, \mathit{ELetRec}\ \mathit{fid}\ \mathit{params}\ b\ e, \mathit{eff}_1\rangle \Downarrow \{\mathit{res}, \mathit{eff}_2\}}$$

Figure 3: The core traditional big-step definition of our case study language

3.1.1. Making it executable

The traditional, relational big-step semantics introduced in the previous section is not inherently executable or computable: for a given pair of starting and final configurations, it needs to be proven that they are in operational semantics relation. In Coq, such a proof can be given in terms of proof primi-

$$\text{(Try)}\quad \frac{\langle \Gamma, e_1, \mathit{eff}_1\rangle \Downarrow \{\mathit{inl\ val'}, \mathit{eff}_2\} \qquad \langle \Gamma[v \mapsto \mathit{val'}], e_2, \mathit{eff}_2\rangle \Downarrow \{\mathit{res}, \mathit{eff}_3\}}{\langle \Gamma, \mathit{ETry}\ e_1\ v\ e_2\ \mathit{vl}\ e_3, \mathit{eff}_1\rangle \Downarrow \{\mathit{res}, \mathit{eff}_3\}}$$

$$\text{(Catch)}\quad \frac{\begin{array}{c}\langle \Gamma, e_1, \mathit{eff}_1\rangle \Downarrow \{\mathit{inr}\ (\mathit{ex}_1, \mathit{ex}_2, \mathit{ex}_3), \mathit{eff}_2\} \\ \langle \mathit{try\_vars\_to\_env}\ \mathit{vl}\ [\mathit{exclass\_to\_value}\ \mathit{ex}_1; \mathit{ex}_2; \mathit{ex}_3]\ \Gamma, e_3, \mathit{eff}_2\rangle \Downarrow \{\mathit{res}, \mathit{eff}_3\}\end{array}}{\langle \Gamma, \mathit{ETry}\ e_1\ v\ e_2\ \mathit{vl}\ e_3, \mathit{eff}_1\rangle \Downarrow \{\mathit{res}, \mathit{eff}_3\}}$$

For the next rule, let us consider
$\mathit{nonclosure}\ v := \forall\ \Gamma', \mathit{ext}, \mathit{var\_list}, \mathit{body} : v \neq \mathit{VClos}\ \Gamma'\ \mathit{ext}\ \mathit{var\_list}\ \mathit{body}.$

$$\text{(AppExc}_1\text{)}\quad \frac{\begin{array}{cc}\langle \Gamma, \mathit{exp}, \mathit{eff}_1\rangle \Downarrow \{\mathit{inl}\ v, \mathit{eff}_2\} & \\ \langle \Gamma, \mathit{params}, \mathit{eff}_2\rangle \Downarrow^{\mathit{all}} \{\mathit{vals}, \mathit{effs}\} & \mathit{nonclosure}\ v \\ & \mathit{eff}_3 = \mathit{last\ effs\ eff}_2\end{array}}{\langle \Gamma, \mathit{EApp}\ \mathit{exp}\ \mathit{params}, \mathit{eff}_1\rangle \Downarrow \{\mathit{inr}\ (\mathit{badfun}\ v), \mathit{eff}_3\}}$$

In the following rule, we denote $\mathit{VClos}\ \Gamma'\ \mathit{ext}\ \mathit{var\_list}\ \mathit{body}$ with v.

$$\text{(AppExc}_2\text{)}\quad \frac{\begin{array}{cc}\langle \Gamma, \mathit{exp}, \mathit{eff}_1\rangle \Downarrow \{\mathit{inl}\ v, \mathit{eff}_2\} & |\mathit{var\_list}| \neq |\mathit{vals}| \\ \langle \Gamma, \mathit{params}, \mathit{eff}_2\rangle \Downarrow^{\mathit{all}} \{\mathit{vals}, \mathit{effs}\} & \mathit{eff}_3 = \mathit{last\ effs\ eff}_2\end{array}}{\langle \Gamma, \mathit{EApp}\ \mathit{exp}\ \mathit{params}, \mathit{eff}_1\rangle \Downarrow \{\mathit{inr}\ (\mathit{badarity}\ v), \mathit{eff}_3\}}$$

$$\text{(CallExc)}\quad \frac{\begin{array}{c}\langle \Gamma, \mathit{params}, \mathit{eff}_1\rangle \Downarrow^{\mathit{up\ to:}\ i} \{\mathit{vals}, \mathit{effs}\} \\ \langle \Gamma, \mathit{params}[i], \mathit{last\ effs\ eff}_1\rangle \Downarrow \{\mathit{inr}\ \mathit{ex}, \mathit{eff}_2\}\end{array}}{\langle \Gamma, \mathit{ECall}\ \mathit{fname}\ \mathit{params}, \mathit{eff}_1\rangle \Downarrow \{\mathit{inr}\ \mathit{ex}, \mathit{eff}_2\}}$$

$$\text{(LetExc)}\quad \frac{\langle \Gamma, e, \mathit{eff}_1\rangle \Downarrow \{\mathit{inr}\ \mathit{ex}, \mathit{eff}_2\}}{\langle \Gamma, \mathit{ELet}\ v\ e\ b, \mathit{eff}_1\rangle \Downarrow \{\mathit{inr}\ \mathit{ex}, \mathit{eff}_2\}} \qquad \text{(AppExc}_3\text{)}\quad \frac{\langle \Gamma, \mathit{exp}, \mathit{eff}_1\rangle \Downarrow \{\mathit{inr}\ \mathit{ex}, \mathit{eff}_2\}}{\langle \Gamma, \mathit{EApp}\ \mathit{exp}\ \mathit{params}, \mathit{eff}_1\rangle \Downarrow \{\mathit{inr}\ \mathit{ex}, \mathit{eff}_2\}}$$

$$\text{(AppExc}_4\text{)}\quad \frac{\begin{array}{c}\langle \Gamma, \mathit{exp}, \mathit{eff}_1\rangle \Downarrow \{\mathit{inl}\ v, \mathit{eff}_2\} \\ \langle \Gamma, \mathit{params}, \mathit{eff}_2\rangle \Downarrow^{\mathit{up\ to:}\ i} \{\mathit{vals}, \mathit{effs}\} \\ \langle \Gamma, \mathit{params}[i], \mathit{last\ effs\ eff}_2\rangle \Downarrow \{\mathit{inr}\ \mathit{ex}, \mathit{eff}_3\}\end{array}}{\langle \Gamma, \mathit{EApp}\ \mathit{exp}\ \mathit{params}, \mathit{eff}_1\rangle \Downarrow \{\mathit{inr}\ \mathit{ex}, \mathit{eff}_3\}}$$

Figure 4: The traditional big-step operational semantics of exception creation and propagation

tives, or one can write a program in the tactic language to construct the proof. Automatic execution of relational semantics can be done with the latter. We could also see the evaluation tactics as a machinery that can turn the relational semantics into functional: a program in the tactic language can perform pattern matching, case distinction and even recursion, and can ultimately compute the results of the relation.

Limitations of Coq tactics. Executing the relational semantics in Coq involves technical considerations: one needs to make sure that the operational semantics derivation rules do not contain auxiliary function calls in their consequences. Otherwise, the Coq tactic language cannot do simple pattern matching on the proof goals and prevents syntax-directed evaluation. In our semantics, we needed to apply minor changes in the derivation rule of variables and at uses of the *append* operation on side effect logs in our Core Erlang semantics [4]. The issue has been solved by refactoring: we replaced the auxiliary function applications with fresh variables and added extra premises stating equality between the variables and the corresponding function applications.

On the other hand, in case of the side effects (and the mentioned append operations) to avoid the introduction of unreasonable numbers of new variables, we changed the use of these traces. Note that currently only *ECall* expressions can cause new side effects, the other rules just have to propagate the logs. Instead of handling only the additional side effects of an expression evaluation step, we rather consider using always the whole initial and final side effect traces (i.e. not only their difference like in [4]). This way we could dispose of the append operations in the consequences of the derivation rules.

Evaluation tactic. We use Coq's tactic sublanguage called Ltac [12] to automate proof construction. In our case, the evaluation of the semantics of the case study language without exceptions is syntax-directed, i.e. a tactic can be designed to evaluate any expression in any context based on pattern-matching on the expression to be evaluated (e.g. *ECall* expression can be evaluated with Call). On the other hand, after introducing exceptions, several derivation rules are applicable for evaluating a particular expression (e.g. there are two rules for *ECall*, five rules for function applications, etc.). We extended the evaluation tactic to try applying the applicable rules one after the other. In cases of rules like AppExc$_4$, we needed to apply them multiple times (with different i indices), to find the expression in the parameter list that evaluates to an exception. This can be seen as a backtracking proof-search for a successful evaluation path.

As it turned out, such evaluation tactics in Coq are rather ineffective in terms of time and space. To speed up the execution, we can create some helper functions and prove lemmas about specific expressions (e.g. the evaluation of parameters which are just literals), so that the evaluation tactic can apply

these lemmas before trying to evaluate an expression with the mentioned slow backtracking process. These lemmas can significantly speed up the evaluation of expressions which contain such specific sub-expressions; however, they only solve a small part of the problem.

3.2. Pretty-big-step semantics

As seen before, the traditional definition contains several similar rules with the same premises. The idea of Charguéraud — called pretty-big-step semantics [8] — is focusing on eliminating this redundancy. Let us discuss his idea through our case study using the evaluation rules for applications. First of all, Charguéraud identified two sources of duplication:

- The similar premises in the rules for exceptions, correct evaluation (and divergence).
- The duplication of the evaluation judgement both for values and exceptions. This is not present in our case study language, however, App could be described in form of two rules: one for exception and one for the value final result.

In the following paragraphs we focus on the first problem. Instead of using duplicated conditions, Charguéraud suggests to use "intermediate terms" which contain the satisfied conditions implicitly. These can be seen also as terms, which remember the state of the evaluation, i.e. which sub-terms have already been evaluated (this resembles a small-step semantics in some aspects).

Applications with intermediate terms. Let us see how the idea applies to our semantics. First, we need to create the syntax for intermediate terms (see Figure 5). In our case, we need three additional constructors for applications: *AApp1* corresponds to the function expression evaluation, *AList* to the evaluation of the parameters, while *AApp2* to the application exception creation and function body evaluation.

```
Inductive AuxExpression :=
| AApp1 (b : Value + Exception) (params : list Expression)
| AApp2 (v : Value) (b : list Value + Exception)
...
.

Inductive AuxList := AList (rest : list Expression)
                           (b : list Value + Exception).
```

Figure 5: The syntax of intermediate terms

In this figure, *lres* is either a list of *Values*, or an exception, so its type is *list Value + Exception*.

$$(\textsc{App}_1^{pretty}) \quad \frac{\langle \Gamma, exp, eff_1 \rangle \Downarrow_p \{res_1, eff_2\} \quad \langle \Gamma, AApp1\ res_1\ params, eff_2 \rangle \Downarrow_p \{res_2, eff_3\}}{\langle \Gamma, EApp\ exp\ params, eff_1 \rangle \Downarrow_p \{res_2, eff_3\}}$$

$$(\textsc{ExcApp}_1^{pretty}) \quad \frac{}{\langle \Gamma, AApp1\ (inr\ ex)\ params, eff_1 \rangle \Downarrow_p \{inr\ ex, eff_1\}}$$

$$(\textsc{FinApp}_1^{pretty}) \quad \frac{\langle \Gamma, AList\ params\ (inl\ []), eff_1 \rangle \Downarrow_p \{lres, eff_2\} \quad \langle \Gamma, AApp2\ v\ lres, eff_2 \rangle \Downarrow_p \{res, eff_3\}}{\langle \Gamma, AApp1\ (inl\ v)\ params, eff_1 \rangle \Downarrow_p \{res, eff_3\}}$$

$$(\textsc{ExcApp}_2^{pretty}) \quad \frac{}{\langle \Gamma, AApp2\ v\ (inr\ ex), eff_1 \rangle \Downarrow_p \{inr\ ex, eff_1\}}$$

$$(\textsc{FinApp}_2^{pretty}) \quad \frac{|var\_list| = |vals| \quad \langle (get\_env\ \Gamma'\ ext)[var\_list \mapsto vals], body, eff_1 \rangle \Downarrow_p \{res, eff_2\}}{\langle \Gamma, AApp2\ (VClos\ \Gamma'\ ext\ var\_list\ body)\ (inl\ vals), eff_1 \rangle \Downarrow_p \{res, eff_2\}}$$

$$(\textsc{ExcApp}_{2,badfun}^{pretty}) \quad \frac{nonclosure\ v}{\langle \Gamma, AApp2\ v\ (inl\ vals), eff_1 \rangle \Downarrow_p \{inr\ (badfun\ v), eff_1\}}$$

In the following rule, we denote $VClos\ \Gamma'\ ext\ var\_list\ body$ with v.

$$(\textsc{ExcApp}_{2,badarity}^{pretty}) \quad \frac{|var\_list| \neq |vals|}{\langle \Gamma, AApp2\ v\ (inl\ vals), eff_1 \rangle \Downarrow_p \{inr\ (badarity\ v), eff_1\}}$$

Figure 6: Pretty-big-step semantics for applications

After having the intermediate terms defined, we can rewrite the semantics of applications (Figure 6 and 7). We decided not to include our side effect traces in the intermediate terms, because this way the effects can be handled just like before. First, we have to evaluate the function expression of the application ($\textsc{App}_1^{pretty}$). We create the intermediate term *AApp1* with the result of this step. If this result was an exception, then the evaluation is finished with $\textsc{ExcApp}_1^{pretty}$, otherwise, the parameters follow after using $\textsc{FinApp}_1^{pretty}$.

$$(\textsc{FinList}^{pretty}) \qquad \frac{}{\langle \Gamma, \mathit{AList}\ []\ (\mathit{inl}\ \mathit{vals}), \mathit{eff}_1\rangle \Downarrow_p \{\mathit{inl}\ \mathit{vals}, \mathit{eff}_1\}}$$

$$(\textsc{ExcList}^{pretty}) \qquad \frac{}{\langle \Gamma, \mathit{AList}\ \mathit{rest}\ (\mathit{inr}\ \mathit{ex}), \mathit{eff}_1\rangle \Downarrow_p \{\mathit{inr}\ \mathit{ex}, \mathit{eff}_1\}}$$

$$(\textsc{StepList}^{pretty}) \qquad \frac{\langle \Gamma, r, \mathit{eff}_1\rangle \Downarrow_p \{\mathit{res}, \mathit{eff}_2\} \quad \langle \Gamma, \mathit{AList}\ \mathit{rest}\ (\mathit{mk\_result}\ \mathit{res}\ \mathit{vals}), \mathit{eff}_2\rangle \Downarrow_p \{\mathit{lres}, \mathit{eff}_3\}}{\langle \Gamma, \mathit{AList}\ (r :: \mathit{rest})\ (\mathit{inl}\ \mathit{vals}), \mathit{eff}_1\rangle \Downarrow_p \{\mathit{lres}, \mathit{eff}_3\}}$$

Figure 7: Pretty-big-step semantics for parameter lists

When there are parameters, we can take the first one and evaluate it with $\textsc{StepList}^{pretty}$. The notation $r :: \mathit{rest}$ means that r is appended to the front of list *rest*. The value of the first expression will be appended to the end of the value list in the constructor *AList* if it is a value by the *mk_result* function; however, in case of an exception this attribute of *AList* becomes the mentioned exception. We repeat this process until all parameter expressions are evaluated, or an exception occurs inside the *AList*. In the latter case, $\textsc{ExcList}^{pretty}$ finishes the evaluation, and the stored exception will be propagated. When there is no exception, we use $\textsc{FinList}^{pretty}$ to finish the parameter list evaluation. At this point, we can notice that this a general approach to evaluating a list of expressions, so it can be used for *ECall* expressions too.

Finally, if there was an exception during parameter evaluation, instead of the parameter values, an exception is stored in *AApp2*, and this can be propagated with $\textsc{ExcApp}_2^{pretty}$. Otherwise, all parameters were correctly evaluated and $\textsc{FinApp}_2^{pretty}$ can be applied, when the first saved value (the evaluated application function expression) is a closure, moreover, the number of formal parameters in this closure is the same as the actual parameters, which are also stored in a value list in *AApp2*. However, if the first stored value is not a closure, we use $\textsc{ExcApp}_{2,badfun}^{pretty}$ and a *badfun* exception will be the result, otherwise, we can apply $\textsc{ExcApp}_{2,badarity}^{pretty}$ if the number of formal and actual parameters mismatch to create a *badarity* exception.

Brief evaluation. Compared to the traditional semantics, in the pretty-big-step approach we see the increase in the number of inference rules, while the premise redundancy is eliminated and the number of premises drops to two at most. Obviously, pretty-big-step semantics cannot overcome all weaknesses of the big-step approach, but provides a good alternative in terms of readability and usability. Transforming the big-step semantics to pretty-big-step style was

a straightforward process, except the transformation of expression lists: if there were two or more derivation steps in the big-step premises, the intermediate results were turned into terms like *AApp1*, and the rule was split. These steps could have been automated, however, in case of expression lists, the use of accumulation in *AList* instead of $\Downarrow^{all}$ was not as simple.

We should also note, that the pretty-big-step definition is a relational semantics just like the traditional one. This means, we need a tactic again to execute this semantics, however, unlike in the traditional case, here backtracking is not needed, because the evaluation is syntax-driven (with the exception of the last step of application evaluation: $\textsc{FinApp}_2^{pretty}$, $\textsc{ExcApp}_{2,badfun}^{pretty}$ and $\textsc{ExcApp}_{2,badarity}^{pretty}$). Although, the use of this evaluation tactic is still not efficient enough.

There are interesting applications of the method in related research, such as deriving pretty-big-step style semantics from small-step semantics [2] or certified abstract interpretation using this definition style [6].

4. Functional ways to define big-step semantics

We have summarised two ways to create a relational big-step semantics, however, both of them suffer mainly from the same problem: they cannot be executed efficiently. In this section, we discuss a functional approach, called functional big-step semantics [21] and its origin, the definitional interpreter [23].

4.1. Functional big-step semantics

The idea of functional big-step semantics was developed by Owens *et al.* [21]. A semantics in this style is basically a recursive function. In order to assure its termination for arbitrary inputs (e.g. for diverging expressions too), there is also a "clock" variable which decreases in the steps of the execution. We note that the functional big-step semantics is essentially a definitional interpreter [23] equipped with a clock, and it is defined in a "higher-order logic rather than a programming language" [21].

This approach is also used in research, for example in the FEther project [26] and the type soundness proof for System F by Amin and Rompf [1] uses definitional interpreters, while a verified compiler backend for CakeML [21, 25] in based on functional big-step semantics. Now, let us see how can we create such a semantics for our case study language.

When we define a function (in Coq), it should explicitly implement behaviour for all inputs (i.e. the function is total). However, in practice, there can be syntactically valid programs or expressions with undefined or unspecified behaviour (we note that the relational semantics are partial). Moreover, be-

cause of the "clock" variable which ensures the termination of the function, an expression evaluation could terminate before finding the right result for small "clocks". This means, we can have three different results: *correct termination*, *failure*, and *timeout* (see Figure 8). In our case study language, there is no undefined behaviour, so we never get *failure* as result, however, we do not omit this from the result definition, because if we extend this definition e.g. with Core Erlang-like `case` expressions, then the guards of these expressions cannot produce observable side effects [7], so the semantics of `case` expressions with such guards would be *failure*.

```
Inductive ResultType : Type :=
| Result (res : Value + Exception)
  (eff : SideEffectList)
| Timeout
| Failure.
```

```
Inductive ResultListType : Type :=
| LResult
    (res : list Value + Exception)
    (eff : SideEffectList)
| LTimeout
| LFailure.
```

Figure 8: The possible results of the functional big-step semantics

As we have seen before, we have to define the semantics for lists of expressions too. For these lists, we can define the functional big-step semantics distinctly, just like in case of the other discussed semantics (see the definitions of $\Downarrow^{all}$, $\Downarrow^{up\ to:\ i}$ in Section 3.1 and the *AList* constructor, StepList^{pretty}, ExcList^{pretty} and FinList^{pretty} in Section 3.2). So we define also a result type for list evaluation (Figure 8).

Now, we have the result types, we can define the functional semantics (Figure 10 shows a representative part of it). The first step of this function is to check whether the clock is already consumed, in this case, the function returns the *Timeout* value. Otherwise, the expression evaluation can begin. For calls and applications, we need the above-mentioned list evaluation. This problem is solved by the other semantics function (see Figure 9), where we pass the partially applied version of the original functional big-step semantics as an argument (we decrease clock only in the original functional big-step semantics, so that Coq can find the decreasing argument of the function to ensure termination, moreover it enables us to use simple, yet powerful induction over the clock). Note that we decided to decrease the clock value on every nested recursive call, otherwise Coq cannot find the decreasing argument of the semantics function (however, this problem could be solved with the `Program` construction possibly too). We also note, that exceptions, failures and timeouts were be handled together (except in the semantics of *ETry*), because these results just needed to be propagated resulting in a short semantics definition.

```
Fixpoint eval_elems
 (f : Environment → Expression → SideEffectList → ResultType)
 (Γ : Environment) (exps : list Expression) (eff : SideEffectList)
 : ResultListType :=
match exps with
| [] ⇒ LResult (inl []) eff
| x::xs ⇒
  match f Γ x eff with
  | Result (inl v) eff' ⇒ let res := eval_elems f Γ xs eff' in
      match res with
      | LResult (inl xs') eff'' ⇒ LResult (inl (v::xs')) eff''
      | r ⇒ r
      end
  | Result (inr ex) eff' ⇒ LResult (inr ex) eff'
  | Failure ⇒ LFailure
  | Timeout ⇒ LTimeout
  end
end.
```

Figure 9: The functional big-step semantics of list evaluation

5. Discussion

In this section we evaluate and compare the semantics definitions given in the previous sections, and we also discuss a coinductive approach to handle divergence. In the diagrams below, we measure the complexity of proofs in lines of code ("Proof length"), number of case distinctions ("Case separations"), number of expressions not inferrable from the context ("Hand written input"), number of used derivation rules ("Rules"), how many times the `inversion` tactic was used ("Inversions") and how many times helper lemmas were used ("Use of helpers"). Moreover, we used examples of different sizes below to compare the efficiency of the evaluation: "Small example" contains 11, "Medium example" consists of 167, while "Large example" includes 3328 constructs.

5.1. Evaluation

First of all, we can notice that all the three semantics can handle the evaluation of list of expressions separately from the body of the semantics: $\Downarrow^{all}$ and $\Downarrow^{up\ to:\ i}$ in the traditional big-step, ExcList^{pretty}, FinList^{pretty}, and StepList^{pretty} in the pretty-big-step, and *eval_elems* in the functional big-step semantics.

In terms of definition size and complexity, the functional big-step definition is superior, it is much more compact than the other two. Besides, the pretty-

```
Fixpoint eval_fbos_expr (clock : nat) (Γ : Environment) (exp : Expression)
 (eff : SideEffectList) {struct clock} : ResultType :=
match clock with
| 0 ⇒ Timeout
| S clock' ⇒
   match exp with
   | EApp exp l ⇒
      match eval_fbos_expr clock' Γ exp eff with
      | Result (inl v) eff' ⇒
        let res := eval_elems (eval_fbos_expr clock') Γ l eff' in
         match res with
         | LResult (inl vl) eff'' ⇒
           match v with
           | VClos Γ' ext varl body ⇒
            if Nat.eqb (length varl) (length vl)
            then eval_fbos_expr clock' ((get_env Γ' ext)[varl ↦ vl]) body eff''
            else Result (inr (badarity v)) eff''
           | _ ⇒ Result (inr (badfun v)) eff''
           end
         | LResult (inr ex) eff'' ⇒ Result (inr ex) eff''
         | LFailure ⇒ Failure
         | LTimeout ⇒ Timeout
         end
       | r ⇒ r
       end
   ...
   end
end.
```

Figure 10: Functional big-step semantics of our case study language

big-step definition uses more inference rules than the traditional big-step semantics, but these rules are much simpler (they have at most two premises). This difference increases the number of subgoals in proofs in case of the pretty-big-step semantics, but these goals are usually simpler than in the other case. In turn, simpler goals could mean simpler proofs, but since the pretty-big-step semantics is defined by mutually inductive types, the related proofs in some cases can become rather complex due to involving mutual induction.

Expression evaluation. The first problem we encounter is that the traditional big-step and pretty-big-step semantics are relational semantics (defined by an inductive type) and are not inherently executable. To describe an expression evaluation, we need to prove the evaluation step-by-step using the inference rules (the constructors of the inductive type). As discussed before,

we can also create an evaluation tactic, which can find the proof for expression evaluation, however, the use of this tactic is not efficient: it takes unreasonable amounts of memory and time (see Figures 11 and 12). The use of pretty-big-step semantics is more efficient than the traditional one, because for one goal, one derivation rule can match syntactically at most (except in case of different application exceptions), thus no backtracking is needed. However, it is still not efficient enough, especially compared to the functional approach.

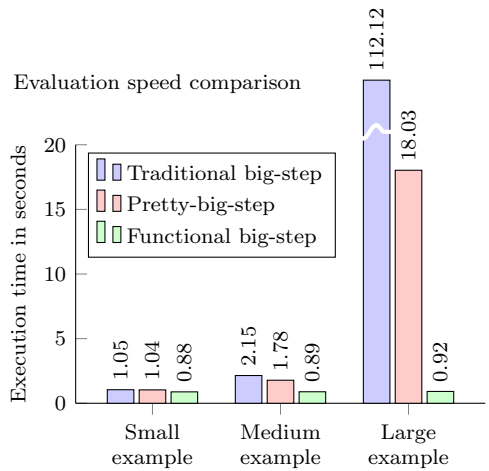

Figure 11: Comparison of formal expression evaluation: time

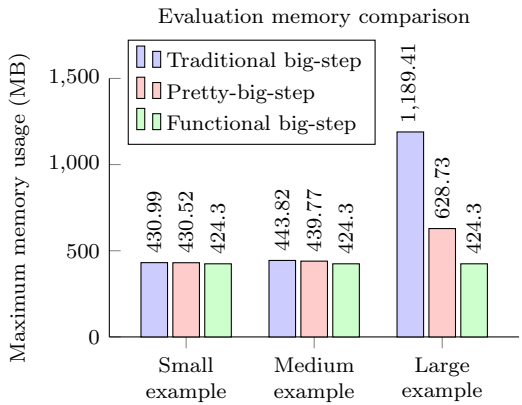

Figure 12: Comparison of formal expression evaluation: memory

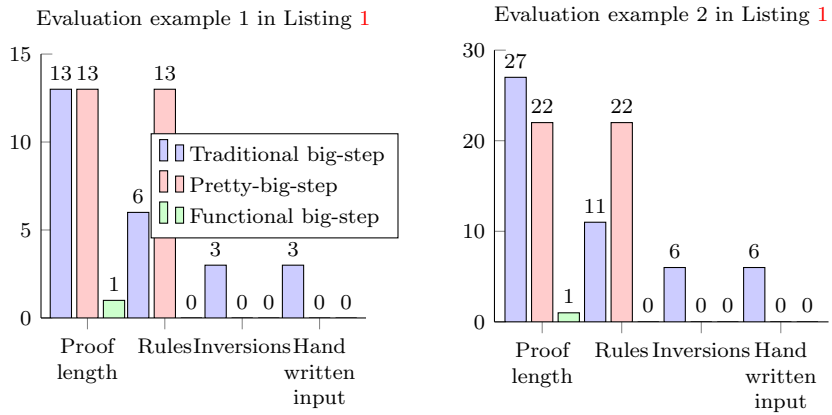

Figure 13: Comparison of formal expression evaluation

We can also see (Figure 13) that the proof length of simple expression evaluations in the traditional and pretty-big-step semantics is similar. However, in the pretty-big-step semantics we used much more inference rules to reach the result, while with the traditional semantics, we had to use inversion tactic several times and specify results by hand. This is because expression list evaluation is not described step-by-step, but in universally quantified predicates, we needed to input the result list of values and side effects (e.g. *eff* and *vs* in the definition of $\Downarrow^{all}$) during formal evaluation (alternatively, list unfolding lemmas[4] based on the length can also solve this problem). This issue is not present in the pretty-big-step semantics, because lists are handled in a step-by-step way by StepList^{pretty} (again, resembling small-step evaluation). All in all, the complexity of these proofs are similar in both relational approaches.

On the other hand, the functional big-step semantics is inherently executable (because it is just a recursive function), so expressions can be simply evaluated using it, we just have to pick an appropriate initial clock value (recursion limit).

Expression equivalence proofs. As we can see (Figure 14), surprisingly the expression equivalence proofs were the most complex in the traditional big-step semantics, while the functional big-step style performed very well. This is because in the traditional semantics we had to use list unfolding lemmas several times, which quickly increased the size of the proofs. The use of pretty-big-step semantics was quite straightforward, and the equivalence proofs were not too complex. Once again, this is partly because no list unfolding lemmas were needed.

[4]For example a list of length n can be described as $[x_1, x_2, .., x_n]$ with the $x_1, x_2, .., x_n$ existential variables (which can be inferred from the context during the proof).

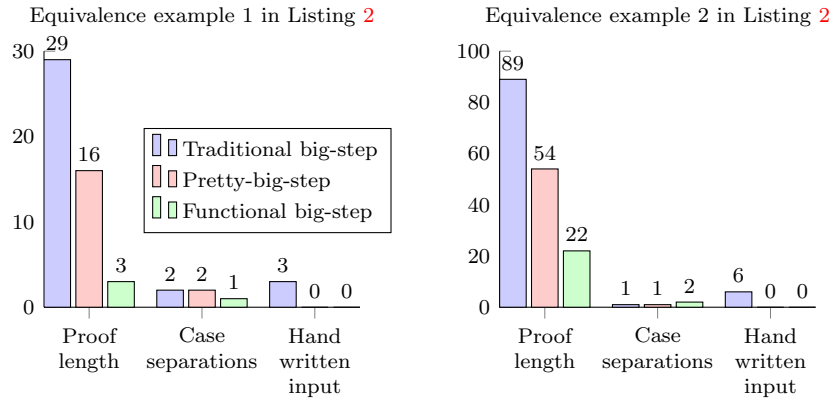

Figure 14: Complexity of expression equivalence proofs

Based on the diagrams, one could think that there is no disadvantage of using functional big-step semantics, but that is not the case. While interactively proving the equivalences (and also semantics properties), the intermediate subgoals and assumptions were hard to read and understand, because Coq usually oversimplified the function definition, and we often saw the whole definition of this semantics, and not just necessary parts of it. We used `remember` tactics [12] on the clock values which allowed us to simplify the semantics step-by-step, however, this solution is not the most convenient one.

Semantics property proofs. For evaluating the complexity of semantics property proofs (see Figure 15), we chose determinism in case of the pretty-big-step and traditional big-step semantics, while a *clock increasing lemma* in case of the functional big-step semantics[5]. The determinism proof for the traditional approach is very complex. We needed to use various helper theorems about (partial) evaluation of lists of expressions and a lot of case distinctions. On the other hand, proving determinism in the pretty-big-step approach was simple, we did not need to create any helper lemmas, we used only a few case distinctions and the proof is short in spite of having to use mutual induction principle.

The proof complexity of the clock increasing lemma for the functional big-step semantics is between the previous two. We had to create and prove one helper theorem, and use several case distinctions. However, this proof is not as complex as the determinism of the traditional style, we have used simple induction over the clock variable. This style of induction is usable, because the clock is decreased in every recursive call of the semantics[6]. We should also

[5] When we evaluate an expression and get a result with the constructor *Result*, then we can increase the initial clock, and get the same result.

[6] We could have used functional induction (similar to the one mentioned by Owens *et al.* [21]), however, we faced limitations of Coq when generating an induction principle.

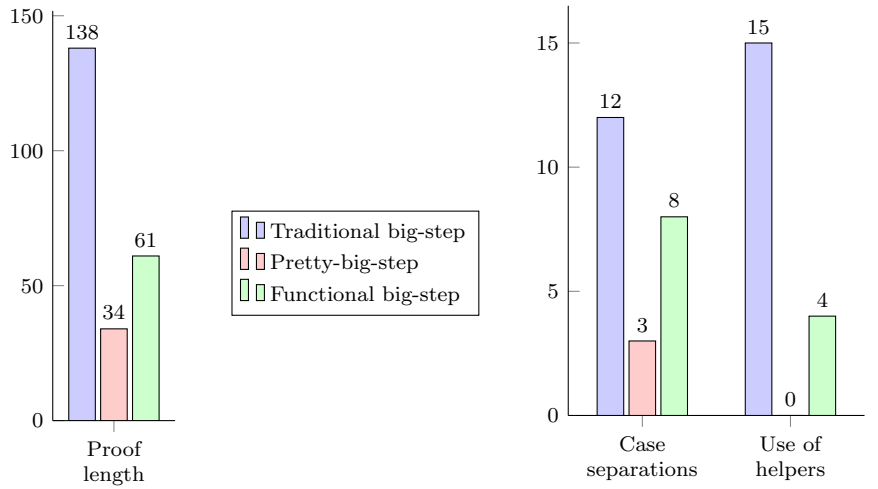

Figure 15: Complexity of semantics property proofs

note that while interactively proving this theorem, the subgoals were difficult to understand because of the reasons mentioned before.

In addition, we also proved the equivalence of these approaches: between pretty-big-step and functional approaches, and between traditional big-step and functional approaches by induction. Thereafter using the previous two, we also proved the equivalence of traditional and pretty-big-step semantics. We encountered one difficulty: while proving the equivalence of pretty and functional big-step semantics, the mutual induction could not be used (in functional big-step semantics, we can not give a meaning for "intermediate terms"). To solve this problem, we followed the footsteps of Charguéraud's [22] formalisation, and defined another (equivalent) version of the pretty-big-step semantics, equipped with a counter which increased when using the derivation rules of the semantics, in order to use induction over this counter. We proved the equivalence using this semantics as an intermediate step.

5.2. Coinductive approach

There are two concept which were not investigated in detail: concurrency and divergence. In general, a big-step semantics can not express concurrency efficiently, because it can not handle interleaving. For this purpose, a small-step approach is more suitable.

We should note that functional big-step semantics can handle divergence in the same way Owens *et al.* [21] described: the evaluation is divergent, when for any possible clock value the result is *Timeout*. We also proved an expression evaluation divergent using this idea and an induction on the clock.

The previously described relational approaches are suitable to describe semantics of terminating expressions, however, they can not effectively express divergence. If one wants to reason about divergence too, a coinductive big-step semantics can be used. We have found the work of Leroy and Grall [16] the most influential, where they define a semantics for λ-calculus extended with constants. They also extend this semantics with traces, a similar feature to our side effect logging approach. Moreover, they also implemented these semantics in Coq and the source is available publicly. We followed their footsteps to define a coinductive big-step semantics for our case-study language (in particular, for applications) with a distinct relation. For the divergence rules, we needed infinite traces for side effects too. However, this approach is not straightforward to use because of the guardedness of subgoals.

6. Conclusion and future work

In this paper we defined various approaches (primarily traditional big-step, pretty-big-step and functional big-step semantics) to define the semantics of a functional programming language, and used a small subset of sequential Core Erlang as a case study.

We proved the equivalence of the various styles of semantics, and evaluated, compared them from different aspects in order to choose the most fitting way to reason about refactoring correctness. Each has its advantages and disadvantages. Our main three aspects were the executability of the approach, the complexity of expression equivalence proofs and proofs about the properties of the semantics, and from this point of view, the functional big-step style semantics proved to be the most useful.

We highlight the fact that the semantics definitions and the proofs are all formalised in the Coq proof management system [24].

In the future, we are planning to formalise functional big-step semantics for sequential Core Erlang to enable effective testing of the semantics and then use comparative testing of our Core Erlang semantics and a small-step Erlang semantics [15] defined by one of our former project members. Naturally, we also plan to prove the existing big-step and the mentioned functional big-step semantics equivalent, after having finished the implementation. We also plan to investigate the coinductive approach more in detail. Our long term goal is to formalise entire Core Erlang and Erlang in Coq to reason about refactoring correctness on Erlang programs.

References

[1] **Amin, N. and T. Rompf**, Type soundness proofs with definitional interpreters, *SIGPLAN Not.*, **52** (2017), 666–679.

[2] **Bach Poulsen, C. and P. D. Mosses**, Deriving pretty-big-step semantics from small-step semantics, in: Z. Shao, Ed., *Programming Languages and Systems* (Berlin, Heidelberg, 2014), Springer Berlin Heidelberg, 270–289.

[3] **Bereczky, P., D. Horpácsi, and S. Thompson**, A proof assistant based formalisation of a subset of sequential Core Erlang, in: A. Byrski and J. Hughes, Eds., *Trends in Functional Programming* (Cham, 2020), Springer International Publishing, 139–158.

[4] **Bereczky, P., D. Horpácsi, and S. J. Thompson**, Machine-checked natural semantics for Core Erlang: Exceptions and side effects, in: *Proceedings of the 19th ACM SIGPLAN International Workshop on Erlang* (New York, NY, USA, 2020), Association for Computing Machinery, 1–13.

[5] **Blazy, S. and X. Leroy**, Mechanized semantics for the Clight subset of the C language, *Journal of Automated Reasoning* **43** (2009) 263–288.

[6] **Bodin, M., T. Jensen, and A. Schmitt**, Certified abstract interpretation with pretty-big-step semantics, in: *Proceedings of the 2015 Conference on Certified Programs and Proofs* (New York, NY, USA, 2015), Association for Computing Machinery, 29–40.

[7] **Carlsson, R., B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding**, Core Erlang 1.0.3 language specification, Technical report, 2004.

[8] **Charguéraud, A.**, Pretty-big-step semantics, in: M. Felleisen and P. Gardner, Eds., *Programming Languages and Systems* (Berlin, Heidelberg, 2013), Springer Berlin Heidelberg, 41–60.

[9] **Ciobâcă, Ş.**, From small-step semantics to big-step semantics, automatically, in: E. B. Johnsen and L. Petre, Eds., *Integrated Formal Methods* (Berlin, Heidelberg, 2013), Springer Berlin Heidelberg, 347–361.

[10] **Ciobâcă, Ş., D. Lucanu, V. Rusu, and G. Roşu**, A language-independent proof system for mutual program equivalence, in: S. Merz and J. Pang, Eds., *Formal Methods and Software Engineering* (Cham, 2014), Springer International Publishing, 75–90.

[11] The Coq Proof Assistant documentation, https://coq.inria.fr/documentation, 2020, Accessed on October 1st, 2020.

[12] Ltac documentation, https://coq.inria.fr/refman/proof-engine/ltac.html, 2020, Accessed on September 22nd, 2020.

[13] **Garrido, A. and J. Meseguer**, Formal specification and verification of Java refactorings, in: *Sixth IEEE International Workshop on Source Code Analysis and Manipulation* (2006), 165–174.

[14] **Kahn, G.**, Natural semantics, in: F. J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds., *STACS 87* (Berlin, Heidelberg, 1987), Springer Berlin Heidelberg, 22–39.

[15] **Kőszegi, J.**, KErl: Executable semantics for Erlang, *CEUR Workshop Proceedings* **2046** (2018) 144–160.

[16] **Leroy, X. and H. Grall**, Coinductive big-step operational semantics, *Information and Computation* **207** (2009), Special issue on Structural Operational Semantics, 284–304.

[17] **Nakata, K. and M. Hasegawa**, Small-step and big-step semantics for call-by-need, *Journal of Functional Programming* **19** (2009), 699–722.

[18] **Nakata, K. and T. Uustalu**, Trace-based coinductive operational semantics for While, in: S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., *Theorem Proving in Higher Order Logics* (Berlin, Heidelberg, 2009), Springer Berlin, Heidelberg, 375–390.

[19] Core Erlang formalization, https://github.com/harp-project/Core-Erlang-Formalization, 2020, Accessed on September 24th, 2020.

[20] **Nipkow, T. and G. Klein**, *Concrete semantics: with Isabelle/HOL*, Springer, 2014.

[21] **Owens, S., M. O. Myreen, R. Kumar, and Y. K. Tan**, Functional big-step semantics, in: P. Thiemann, Ed., *Programming Languages and Systems* (Berlin, Heidelberg, 2016), Springer Berlin Heidelberg, 589–615.

[22] Pretty-big-step semantics formalisation, http://www.charguéraud.org/research/2012/pretty/, Accessed on October 26th, 2020.

[23] **Reynolds, J. C.**, Definitional interpreters for higher-order programming languages.

[24] Semantics comparison, https://github.com/harp-project/Semantics-comparison, 2020, Accessed on January 18th, 2023.

[25] **Tan, Y. K., M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish**, A new verified compiler backend for CakeML, *SIGPLAN Not.* **51** (2016), 60–73.

[26] **Yang, Z. and H. Lei**, FEther: An extensible definitional interpreter for smart-contract verifications in Coq, *IEEE Access* **7** (2019), 37770–37791.

P. Bereczky and D. Horpácsi
Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest
Hungary
berpeti@inf.elte.hu and daniel-h@elte.hu

S. Thompson
Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest
Hungary
University of Kent
School of Computing
Canterbury
United Kingdom
S.J.Thompson@kent.ac.uk