Pretty-Big-Step Semantics

Arthur Charguéraud

Inria Saclay – Île-de-France & LRI, Université Paris Sud, CNRS
arthur.chargueraud@inria.fr

Abstract. In spite of the popularity of small-step semantics, big-step semantics remain used by many researchers. However, big-step semantics suffer from a serious duplication problem, which appears as soon as the semantics account for exceptions and/or divergence. In particular, many premises need to be copy-pasted across several evaluation rules. This duplication problem, which is particularly visible when scaling up to full-blown languages, results in formal definitions growing far bigger than necessary. Moreover, it leads to unsatisfactory redundancy in proofs. In this paper, we address the problem by introducing pretty-big-step semantics. Pretty-big-step semantics preserve the spirit of big-step semantics, in the sense that terms are directly related to their results, but they eliminate the duplication associated with big-step semantics.

1 Introduction

There are two traditional approaches to formalizing the operational semantics of a programming language: small-step semantics [11], and big-step semantics [7]. In small-step semantics, the subterm in evaluation position is reduced step by step and these transitions are reflected at the top level. In big-step semantics, a term is directly related to its result, and the behavior of a term is expressed in terms of the behavior of its subterms. While provably equivalent, these two approaches are fundamentally different in terms of how evaluation rules are stated and how proofs are conducted.

This paper describes and proposes a solution to a severe limitation of big-step semantics: the fact that a number of rules and premises need to be duplicated in order to handle exceptions and divergence. In particular, this limitation typically discourages the use of big-step semantics in mechanized definitions of large-scale languages. Before trying to address this limitation of the big-step semantics, we may ask ourselves: Why should we care about big-step semantics? Why not just use small-step semantics all the time?

To find out whether big-step semantics are still being used, we opened up proceedings from recent programming language conferences. We counted the number of research papers making use of a big-step semantics. In ICFP'11, 5 papers were describing results based on a big-step semantics, out of 8 papers that had an operational semantics. In POPL'11, there were 7 out of 23. In ICFP'12, there were 5 out of 9. An immediate conclusion that we can draw from these rough statistics is that *big-step is not dead*.

A closer look at the papers involved reveals that the choice of the operational semantics usually depends on the topic covered by the paper. Papers on type systems nearly always use a small-step semantics to conduct a soundness proof in Wright and Felleisen's style [12]. Papers describing a machine-level language typically use a small-step relation to describe transitions between pairs of machine configurations. Papers concerned with concurrent languages are also almost exclusively described using small-step semantics. Furthermore, a majority of the mechanized definitions of full-blown programming languages that have been developed in recent years were based on small-step semantics.

There are, however, topics for which the use of a big-step semantics appears to prevail. Cost semantics, which associate a cost to the evaluation of every expression, are mainly presented as big-step relations. Program logics often have soundness and completeness proofs that are easier to conduct with respect to a big-step relation. In particular, there are cases of completeness proofs, such as that developed in the author's thesis [1], that need to be conducted by induction over a big-step derivation; any attempt to build the completeness proof with respect to a small-step semantics amounts to re-proving on-the-fly the equivalence between small-step and big-step semantics. Moreover, there are compiler transformations that are easier to prove correct with respect to big-step semantics, in particular for transformations introducing so-called "administrative redexes", which typically clutter simulation diagrams based on small-step semantics.

Big-step semantics are also widely used in informal descriptions. For example, the reference manual of a programming language typically contain sentences of the form "to evaluate $\mathsf{if}\, e1\, \mathsf{then}\, e2\, \mathsf{else}\, e3$, first evaluate $e1$; if the result is true, then evaluate $e2$; otherwise, evaluate $e3$." None of the many reference manuals that we have looked at contains a sentence of the form "if $e1$ takes a step to expression $e1'$ then $\mathsf{if}\, e1\, \mathsf{then}\, e2\, \mathsf{else}\, e3$ takes a step to $\mathsf{if}\, e1'\, \mathsf{then}\, e2\, \mathsf{else}\, e3$." Thus, we speculate that it would be easier to convince the standards committee in charge of a given programming language of the adequacy of a big-step formalization than to convince them of the adequacy of a small-step formalization.

Given that there are a number of important applications for which big-step semantics seem to have an edge on small-step semantics, any significant improvement to big-step semantics should be considered as a valuable contribution.

In this paper, we focus on a critical issue associated with big-step semantics: the amount of duplication involved in the statement of the evaluation rules. To illustrate the extent of the problem, consider a C-style for-loop of the form "$\mathsf{for}\,(\,;\, t_1\,;\, t_2)\,\{\, t_3\,\}$", that is, a for-loop where the initialization expression has already been executed. We use the notation "$\mathsf{for}\, t_1\, t_2\, t_3$" to describe such a loop. We next formalize its big-step semantics. For terminating executions, the evaluation judgment takes the form $t_{/m_1} \Rightarrow v_{/m_2}$, asserting that, in a store m_1, the evaluation of t terminates on the value v in a store m_2. The two rules at the top of Figure 1 describe the regular execution of a loop. When the loop condition t_1 evaluates to false, the loop terminates. Otherwise, if t_1 evaluates to true, we evaluate the body t_3 of the loop, and obtain the unit value, written tt. We then evaluate the stepping expression t_2, and start over.

$$\frac{t_{1/m_1} \Rightarrow \mathsf{false}_{/m_2}}{\mathsf{for}\, t_1\, t_2\, t_{3/m_1} \Rightarrow tt_{/m_2}}$$

$$\frac{t_{1/m_1} \Rightarrow \mathsf{true}_{/m_2} \quad t_{3/m_2} \Rightarrow tt_{/m_3} \quad t_{2/m_3} \Rightarrow tt_{/m_4} \quad \mathsf{for}\, t_1\, t_2\, t_{3/m_4} \Rightarrow tt_{/m_5}}{\mathsf{for}\, t_1\, t_2\, t_{3/m_1} \Rightarrow tt_{/m_5}}$$

$$\frac{t_{1/m_1} \Rightarrow^{\mathsf{exn}}{}_{/m_2}}{\mathsf{for}\, t_1\, t_2\, t_{3/m_1} \Rightarrow^{\mathsf{exn}}{}_{/m_2}} \qquad \frac{t_{1/m_1} \Rightarrow^{\infty}}{\mathsf{for}\, t_1\, t_2\, t_{3/m_1} \Rightarrow^{\infty}}\,\mathsf{co}$$

$$\frac{t_{1/m_1} \Rightarrow \mathsf{true}_{/m_2} \quad t_{3/m_2} \Rightarrow^{\mathsf{exn}}{}_{/m_3}}{\mathsf{for}\, t_1\, t_2\, t_{3/m_1} \Rightarrow^{\mathsf{exn}}{}_{/m_3}} \qquad \frac{t_{1/m_1} \Rightarrow \mathsf{true}_{/m_2} \quad t_{3/m_2} \Rightarrow^{\infty}}{\mathsf{for}\, t_1\, t_2\, t_{3/m_1} \Rightarrow^{\infty}}\,\mathsf{co}$$

$$\frac{t_{1/m_1} \Rightarrow \mathsf{true}_{/m_2} \quad t_{3/m_2} \Rightarrow tt_{/m_3} \quad t_{2/m_3} \Rightarrow^{\mathsf{exn}}{}_{/m_4}}{\mathsf{for}\, t_1\, t_2\, t_{3/m_1} \Rightarrow^{\mathsf{exn}}{}_{/m_4}} \qquad \frac{t_{1/m_1} \Rightarrow \mathsf{true}_{/m_2} \quad t_{3/m_2} \Rightarrow tt_{/m_3} \quad t_{2/m_3} \Rightarrow^{\infty}}{\mathsf{for}\, t_1\, t_2\, t_{3/m_1} \Rightarrow^{\infty}}\,\mathsf{co}$$

$$\frac{t_{1/m_1} \Rightarrow \mathsf{true}_{/m_2} \quad t_{3/m_2} \Rightarrow tt_{/m_3} \quad t_{2/m_3} \Rightarrow tt_{/m_4} \quad \mathsf{for}\, t_1\, t_2\, t_{3/m_4} \Rightarrow^{\mathsf{exn}}{}_{/m_5}}{\mathsf{for}\, t_1\, t_2\, t_{3/m_1} \Rightarrow^{\mathsf{exn}}{}_{/m_5}} \qquad \frac{t_{1/m_1} \Rightarrow \mathsf{true}_{/m_2} \quad t_{3/m_2} \Rightarrow tt_{/m_3} \quad t_{2/m_3} \Rightarrow tt_{/m_4} \quad \mathsf{for}\, t_1\, t_2\, t_{3/m_4} \Rightarrow^{\infty}}{\mathsf{for}\, t_1\, t_2\, t_{3/m_1} \Rightarrow^{\infty}}\,\mathsf{co}$$

Fig. 1. Big-step rules for C loops of the form "$\mathsf{for}(\,;t_1\,;t_2)\,\{\,t_3\,\}$", written "$\mathsf{for}\, t_1\, t_2\, t_3$".

The four rules at the bottom-left of Figure 1 describe the case of an exception being raised during the execution of the loop. These rules are expressed using another inductive judgment, written $t_{/m_1} \Rightarrow^{\mathsf{exn}}{}_{/m_2}$. They capture the fact that the exception may be triggered during the evaluation of any of the subexpressions, or during the subsequent iterations of the loop. The four rules at the bottom-right of Figure 1 describe the case of the loop diverging. These rules rely on a coinductive big-step judgment, written $t_{/m} \Rightarrow^{\infty}$ [2, 8]. "Coinductive" means that a derivation tree for a judgment of the form $t_{/m} \Rightarrow^{\infty}$ may be infinite.

The amount of duplication in Figure 1 is overwhelming. There are two distinct sources of duplication. First, the rules for exceptions and the rules for divergence are extremely similar. Second, a number of evaluation premises are repeated across many of the rules. For example, even if we ignore the rules for divergence, the premise $t_{1/m_1} \Rightarrow \mathsf{true}_{/m_2}$ appears 4 times. Similarly, $t_{3/m_2} \Rightarrow tt_{/m_3}$ appears 3 times and $t_{2/m_3} \Rightarrow tt_{/m_4}$ appears 2 times. This pattern is quite typical in big-step semantics for constructs with several subterms.

One may wonder whether the rules from Figure 1 can be factorized. The only obvious factorization consists of merging the regular evaluation judgment ($t_{/m_1} \Rightarrow v_{/m_2}$) with the judgment for exceptions ($t_{/m_1} \Rightarrow^{\mathsf{exn}}{}_{/m_2}$), using a single evaluation judgment that relates a term to a *behavior*, which consists of either a value or an exception. This factorization, quite standard in big-step semantics, here only saves one evaluation rule: the second rule from the top of Figure 1 would be merged with the rule at the bottom-left corner. It is, however, not easy to factorize the evaluation judgment with the divergence judgment, because one is inductive while the other is coinductive. Another trick sometimes used to reduce

the amount of duplication is to define the semantics of a for-loop in terms of other language constructs, using the encoding "$\mathsf{if}\, t_1\, \mathsf{then}\, (t_3\, ;\, t_2\, ;\, \mathsf{for}\, t_1\, t_2\, t_3)\, \mathsf{else}\, tt$". Yet, this approach does not support *break* and *continue* instructions, so it cannot be applied in general. In summary, just to define the semantics of for-loops, even if we merge the two inductive judgments, we need at least 9 evaluation rules with a total number of 21 evaluation premises. We will show how to achieve a much more concise definition, using only 6 rules with 7 evaluation premises.

With the *pretty-big-step semantics* introduced in this paper, we are able to eliminate the two sources of duplication associated with big-step definitions. First, we eliminate the duplication of premises. To that end, we break down evaluation rules into simpler rules, each of them evaluating at most one subterm. This transformation introduces a number of intermediate terms and increases the number of evaluation rules, but it eliminates the need for duplicating premises across several rules. Overall, the size of the formal definitions usually decreases.

Second, we set up the set of evaluation rules in such a way that it characterizes either *terminating* executions or *diverging* executions, depending on whether we consider an *inductive* or a *coinductive* interpretation for this set of rules. In contrast to Cousot and Cousot's bi-inductive semantics [3, 4], which are based on the construction of a least fixed point of the set of evaluation rules with respect to a non-standard ordering that corresponds neither to induction nor coinduction, our definitions are based on the standard notions of induction and coinduction (as provided, e.g., by Coq).

Furthermore, we show that, when adding traces to the pretty-big-semantics, the coinductive judgment suffices to describe both terminating and diverging executions. Our definitions syntactically distinguish finite traces from infinite traces. This approach leads to rules that are, in our opinion, simpler to understand and easier to reason about than rules involving possibly-infinite traces (coinductive lists), as used by Nakata and Uustalu [10] and Danielsson [5].

In theory, the fact that we are able to capture the semantics through a single judgment means that we should be able to establish, through a *single* proof, that a program transformation correctly preserves both terminating and diverging behaviors. Unfortunately, the guard condition implemented in existing proof assistants such as Coq or Agda prevents us from conducting such reasoning. Workarounds are possible, but the encodings involved are so tedious that it would not be realistic to use them in practice. For this reason, we have to postpone the construction of proofs based on pretty-big-step trace semantics.

In this paper, we also investigate the formalization of type soundness proofs. Interestingly, the pretty-big-step semantics allows for a *generic error rule* that replaces all the error rules that are typically added manually to the semantics. This generic error rule is based on a *progress judgment*, whose definition can be derived in a simple and very systematic way from the set of evaluation rules.

To demonstrate the ability of the pretty-big-step to accommodate realistic languages, we formalized a large fragment of Caml Light. Compared with the big-step semantics, the pretty-big-step semantics has a size reduced by about 40%.

This paper is organized as follows. In §2, we explain how to turn a big-step semantics into its pretty-big-step counterpart. In §3, we discuss error rules and type soundness proofs. In §4, we show how to extend the semantics with traces. In §5, we explain how to handle more advanced language constructs and report on the formalization of core-Caml. We then discuss related work (§6), and conclude (§7). All the definitions and proofs from this paper have been formalized in Coq and put online at: `http://arthur.chargueraud.org/research/2012/pretty`.

2 Pretty-big-step semantics

2.1 Decomposition of big-step rules

We present the pretty-big-step semantics using the call-by-value λ-calculus. The grammar of values and terms are as follows.

$$v \;:=\; \mathsf{int}\, n \;\mid\; \mathsf{abs}\, x\, t \qquad\qquad t \;:=\; \mathsf{val}\, v \;\mid\; \mathsf{var}\, x \;\mid\; \mathsf{app}\, t\, t$$

Thereafter, we leave the constructor val implicit, writing simply v instead of $\mathsf{val}\, v$ whenever a term is expected. (In Coq, we register val as a coercion.) We recall the definition of the standard big-step judgment, which is written $t \Rightarrow v$.

$$\frac{}{v \Rightarrow v} \qquad\qquad \frac{t_1 \Rightarrow \mathsf{abs}\, x\, t \qquad t_2 \Rightarrow v \qquad [x \to v]\, t \Rightarrow v'}{\mathsf{app}\, t_1\, t_2 \Rightarrow v'}$$

The rules of the pretty-big-step semantics are obtained by decomposing the rules above into more atomic rules that consider the evaluation of at most one subterm at a time. A first attempt at such a decomposition consists of replacing the evaluation rule for applications with the following three rules.

$$\frac{t_1 \Rightarrow v_1 \quad \mathsf{app}\, v_1\, t_2 \Rightarrow v'}{\mathsf{app}\, t_1\, t_2 \Rightarrow v'} \qquad \frac{t_2 \Rightarrow v_2 \quad \mathsf{app}\, v_1\, v_2 \Rightarrow v'}{\mathsf{app}\, v_1\, t_2 \Rightarrow v'} \qquad \frac{[x \to v]\, t \Rightarrow v'}{\mathsf{app}\, (\mathsf{abs}\, x\, t)\, v \Rightarrow v'}$$

These rules, without further constraints, suffer from an overlapping problem. For example, consider the term $\mathsf{app}\, v_1\, t_2$. This term is subject to the application of the second rule, which evaluates t_2. However, it is also subject to application of the first rule, whose first premise would reduce v_1 to itself and whose second premise would be identical to the conclusion of the rule. The fact that two different rules can be applied to a same term means that the evaluation judgment is not syntax-directed and thus not very convenient to work with. Even worse, the fact that an evaluation rule can be applied without making progress is problematic when considering a coinductive interpretation of the evaluation rules; typically, one could prove, by applying the first reduction rule infinitely many times, that any term of the form $\mathsf{app}\, v_1\, t_2$ diverges.

Cousot and Cousot [3, 4], who use a similar decomposition of the big-step rules as shown above, prevent the overlapping of the rules by adding side-conditions. For example, the evaluation rule that reduces $\mathsf{app}\, t_1\, t_2$ has a side-condition enforcing t_1 to not be already a value. However, such side-conditions are numerous and they need to be discharged in formal proofs.

Instead of using side-conditions, we ensure that the three evaluation rules introduced above are always applied one after the other by introducing *intermediate terms*, whose grammar is shown below. Observe that intermediate terms are not defined as an extension of the grammar of regular terms, but as a new grammar that embeds that of regular terms. This presentation avoids polluting the syntax of source terms with purely-semantical entities.

$$e := \mathsf{trm}\, t \;\mid\; \mathsf{app1}\, v\, t \;\mid\; \mathsf{app2}\, v\, v$$

We extend the evaluation judgment to intermediate terms, defining an inductive judgment of the form $e \Downarrow v$. In the particular case where e describes a regular term, the judgment takes the form $(\mathsf{trm}\, t) \Downarrow v$. Thereafter, we leave the constructor trm implicit and thus simply write $t \Downarrow v$. The predicate $e \Downarrow v$ is defined inductively by the rules shown below. The evaluation of an application $\mathsf{app}\, t_1\, t_2$ takes three step. First, we reduce t_1 into v_1 and obtain the term $\mathsf{app1}\, v_1\, t_2$. Second, we reduce t_2 into v_2 and obtain the term $\mathsf{app2}\, v_1\, v_2$. Third, assuming v_1 to be of the form $\mathsf{abs}\, x\, t$, we proceed to the β-reduction and evaluate $[x \to v]\, t$ in order to obtain some final result v'.

$$\frac{}{v \Downarrow v} \qquad \frac{t_1 \Downarrow v_1 \qquad \mathsf{app1}\, v_1\, t_2 \Downarrow v'}{\mathsf{app}\, t_1\, t_2 \Downarrow v'}$$

$$\frac{t_2 \Downarrow v_2 \qquad \mathsf{app2}\, v_1\, v_2 \Downarrow v'}{\mathsf{app1}\, v_1\, t_2 \Downarrow v'} \qquad \frac{[x \to v]\, t \Downarrow v'}{\mathsf{app2}\, (\mathsf{abs}\, x\, t)\, v \Downarrow v'}$$

The definitions above provide an adequate reformulation of the big-step semantics by which complex rules have been decomposed into a larger number of more elementary rules. This decomposition avoids the duplication of premises when adding support for exceptions and divergence. Observe that the intermediate terms introduced in the process correspond to the intermediate states of an interpreter. For example, the form $\mathsf{app1}\, v_1\, t_2$ corresponds to the state of the interpreter after the evaluation of the first let-binding in the code "$\mathsf{let}\, v_1 = \mathsf{eval}\, t_1\, \mathsf{in}\;\; \mathsf{let}\, v_2 = \mathsf{eval}\, t_2\, \mathsf{in}\;\; \mathsf{let}\, (\mathsf{abs}\, x\, t) = v_1\, \mathsf{in}\;\; \mathsf{eval}\, ([x \to v_2]\, t)$".

2.2 Treatment of exceptions

We now extend the source language with value-carrying exceptions and exception handlers. The term $\mathsf{raise}\, t$ builds an exception and throws it. The term $\mathsf{try}\, t_1\, t_2$ is an exception handler with body t_1 and handler t_2. Its semantics is as follows. If t_1 produces a regular value, then $\mathsf{try}\, t_1\, t_2$ returns this value. However, if t_1 raises an exception carrying a value v_1, then $\mathsf{try}\, t_1\, t_2$ reduces to $\mathsf{app}\, t_2\, v_1$.

To describe the fact that a term can produce either a regular value or an exception carrying a value, the evaluation judgment is generalized to the form $e \Downarrow b$, where b denotes a *behavior*, built according to the grammar below.

$$b := \mathsf{ret}\, v \;\mid\; \mathsf{exn}\, v$$

Because we generalize the form of the judgment, we also need to generalize the form of the intermediate terms. For example, consider the evaluation of an application $\mathsf{app}\, t_1\, t_2$. First, we evaluate t_1 into a behavior b_1. We then obtain the intermediate term $\mathsf{app1}\, b_1\, t_2$. To evaluate this later term, we need to distinguish two cases. On the one hand, if b_1 is of the form $\mathsf{ret}\, v_1$, then we should evaluate the second branch t_2. On the other hand, if b_1 is of the form $\mathsf{exn}\, v$, then we should directly propagate $\mathsf{exn}\, v$. The updated grammar of intermediate terms, augmented with intermediate forms for raise and try, is as follows.

$$e := \mathsf{trm}\, t \mid \mathsf{app1}\, b\, t \mid \mathsf{app2}\, v\, b \mid \mathsf{raise1}\, b \mid \mathsf{try1}\, b\, t$$

The definition of $e \Downarrow b$ follows a similar pattern as previously. It now also includes rules for propagating exceptions. For example, $\mathsf{app1}\, (\mathsf{exn}\, v)\, t$ evaluates to $\mathsf{exn}\, v$. Moreover, the definition includes rules for evaluating raise and try. For example, to evaluate $\mathsf{try}\, t_1\, t_2$, we first evaluate t_1 into a behavior b_1, and then we evaluate the term $\mathsf{try1}\, b_1\, t_2$. In the rules shown below, the constructor ret is left implicit.

$$\frac{}{v \Downarrow v} \qquad \frac{t_1 \Downarrow b_1 \qquad \mathsf{app1}\, b_1\, t_2 \Downarrow b}{\mathsf{app}\, t_1\, t_2 \Downarrow b} \qquad \frac{}{\mathsf{app1}\, (\mathsf{exn}\, v)\, t \Downarrow \mathsf{exn}\, v}$$

$$\frac{t_2 \Downarrow b_2 \qquad \mathsf{app2}\, v_1\, b_2 \Downarrow b}{\mathsf{app1}\, v_1\, t_2 \Downarrow b} \qquad \frac{}{\mathsf{app2}\, v\, (\mathsf{exn}\, v) \Downarrow \mathsf{exn}\, v} \qquad \frac{[x \to v]\, t \Downarrow b}{\mathsf{app2}\, (\mathsf{abs}\, x\, t)\, v \Downarrow b}$$

$$\frac{t \Downarrow b_1 \qquad \mathsf{raise1}\, b_1 \Downarrow b}{\mathsf{raise}\, t \Downarrow b} \qquad \frac{}{\mathsf{raise1}\, v \Downarrow \mathsf{exn}\, v} \qquad \frac{}{\mathsf{raise1}\, (\mathsf{exn}\, v) \Downarrow \mathsf{exn}\, v}$$

$$\frac{t_1 \Downarrow b_1 \qquad \mathsf{try1}\, b_1\, t_2 \Downarrow b}{\mathsf{try}\, t_1\, t_2 \Downarrow b} \qquad \frac{}{\mathsf{try1}\, v\, t \Downarrow v} \qquad \frac{\mathsf{app}\, t\, v \Downarrow b}{\mathsf{try1}\, (\mathsf{exn}\, v)\, t \Downarrow b}$$

2.3 Treatment of divergence

The above set of rules only describes terminating evaluations. To specify diverging evaluations, we are going to generalize the grammar of behaviors and to consider a coinductive interpretation of the same set of rules as that describing terminating evaluations.

First, we introduce the notion of *outcome*: the outcome of an execution is either to terminate on a behavior b (i.e., to return a value or an exception), or to diverge. We explicitly materialize the divergence outcome with a constant, called div. An outcome, written o, is thus described as follows: $o := \mathsf{ter}\, b \mid \mathsf{div}$.

We update accordingly the grammar of intermediate terms. For example, consider the evaluation of an application $\mathsf{app}\, t_1\, t_2$. First, we evaluate t_1 into some outcome o_1 (a value, an exception, or divergence). We then consider the term $\mathsf{app1}\, o_1\, t_2$, whose evaluation depends on o_1. If o_1 describes a value v_1, we can continue as usual by evaluating t_2. However, if o_1 describes an exception or the constant div, then the term $\mathsf{app1}\, o_1\, t_2$ directly propagates the outcome o_1.

$$b := \mathsf{ret}\, v \mid \mathsf{exn}\, v \qquad o := \mathsf{ter}\, b \mid \mathsf{div} \qquad e := \mathsf{trm}\, t \mid \mathsf{app1}\, o\, t \mid \mathsf{app2}\, v\, o \mid \mathsf{raise1}\, o \mid \mathsf{try1}\, o\, t$$

$$\frac{}{\mathsf{abort}\,(\mathsf{exn}\, v)} \qquad \frac{}{\mathsf{abort}\,\mathsf{div}} \qquad \frac{}{v \Downarrow v} \qquad \frac{t_1 \Downarrow o_1 \qquad \mathsf{app1}\, o_1\, t_2 \Downarrow o}{\mathsf{app}\, t_1\, t_2 \Downarrow o}$$

$$\frac{\mathsf{abort}\, o}{\mathsf{app1}\, o\, t \Downarrow o} \qquad \frac{t_2 \Downarrow o_2 \qquad \mathsf{app2}\, v_1\, o_2 \Downarrow o}{\mathsf{app1}\, v_1\, t_2 \Downarrow o} \qquad \frac{\mathsf{abort}\, o}{\mathsf{app2}\, v\, o \Downarrow o} \qquad \frac{[x \to v]\, t \Downarrow o}{\mathsf{app2}\,(\mathsf{abs}\, x\, t)\, v \Downarrow o}$$

$$\frac{t \Downarrow o_1 \qquad \mathsf{raise1}\, o_1 \Downarrow o}{\mathsf{raise}\, t \Downarrow o} \qquad \frac{\mathsf{abort}\, o}{\mathsf{raise1}\, o \Downarrow o} \qquad \frac{}{\mathsf{raise1}\, v \Downarrow \mathsf{exn}\, v} \qquad \frac{t_1 \Downarrow o_1 \qquad \mathsf{try1}\, o_1\, t_2 \Downarrow o}{\mathsf{try}\, t_1\, t_2 \Downarrow o}$$

$$\frac{}{\mathsf{try1}\, v\, t \Downarrow v} \qquad \frac{\mathsf{app}\, t\, v \Downarrow o}{\mathsf{try1}\,(\mathsf{exn}\, v)\, t \Downarrow o} \qquad \frac{\mathsf{abort}\, o \qquad \forall v.\ o \neq \mathsf{exn}\, v}{\mathsf{try1}\, o\, t \Downarrow o}$$

Fig. 2. Pretty-big-step semantics: $e \Downarrow o$ (inductive) and $e \Downarrow^{\mathsf{co}} \mathsf{div}$ (coinductive), with the constructors val, trm, ret and ter left implicit in the rules.

To capture the fact that $\mathsf{app1}\, o_1\, t_2$ returns o_1 both when o_1 describes divergence or an exception, we use an auxiliary predicate, called abort. The predicate $\mathsf{abort}\, o_1$ asserts that o_1 "breaks the normal control flow" in the sense that o_1 is either of the form $\mathsf{exn}\, v$ or is equal to div. We are then able to factorize the rules propagating exceptions and divergence into a single *abort rule*, as shown below.

$$\frac{\mathsf{abort}\, o_1}{\mathsf{app1}\, o_1\, t_2 \Downarrow o_1}$$

For describing terminating evaluations, we use an inductive judgment of the form $e \Downarrow o$. The particular form $e \Downarrow \mathsf{ter}\, b$, simply written $e \Downarrow b$, corresponds to the same evaluation judgment as that defined previously. For describing diverging evaluations, we use a *coevaluation* judgment, written $e \Downarrow^{\mathsf{co}} o$, which is defined by taking a coinductive interpretation of the same set of rules as that defining the inductive judgment $e \Downarrow o$. The particular form $e \Downarrow^{\mathsf{co}} \mathsf{div}$ asserts that the execution of e diverges.

The complete set of rules defining both $e \Downarrow o$ and $e \Downarrow^{\mathsf{co}} o$ appears in Figure 2. One novelty is the last rule, which is used to propagate divergence out of exception handlers. The rule captures the fact that $\mathsf{try1}\,\mathsf{div}\, t$ produces the outcome div, but it is stated in a potentially more general way that will be useful when adding errors as a new kind of behavior. Remark: in Coq, we currently need to copy-paste all the rules in order to build one inductive definition and one coinductive definition, however it would be easy to implement a Coq plug-in to automatically generate the coinductive definition from the inductive one.

2.4 Properties of the judgments

While we are ultimately only interested in the forms $e \Downarrow b$ and $e \Downarrow^{\mathsf{co}} \mathsf{div}$, our definitions syntactically allow for the forms $e \Downarrow \mathsf{div}$ and $e \Downarrow^{\mathsf{co}} b$. It is worth

clarifying their interpretation. For the former, the situation is quite simple: the form $e \Downarrow \mathsf{div}$ is derivable only when e is an intermediate term that carries a div. In particular, $t \Downarrow \mathsf{div}$ is never derivable.

Lemma 1. *For any term t,* $t \Downarrow \mathit{div} \;\to\; \mathit{False}$.

The interpretation of the form $e \Downarrow^{\mathsf{co}} b$ is more subtle. On the one hand, the coinductive judgment contains the inductive one, because any finite derivation is also a potentially-infinite derivation. It is trivial to prove the following lemma.

Lemma 2. *For any term e and outcome o,* $e \Downarrow o \;\to\; e \Downarrow^{\mathsf{co}} o$.

On the other hand, due to coinduction, it is sometimes possible to derive $e \Downarrow^{\mathsf{co}} b$ even when e diverges. For example, consider $\omega = \mathsf{app}\,\delta\,\delta$, where $\delta = \mathsf{abs}\,x\,(\mathsf{app}\,x\,x)$; one can prove by coinduction that, for any outcome o, the relation $\omega \Downarrow^{\mathsf{co}} o$ holds. Nevertheless, the coevaluation judgment is relatively well-behaved, in the sense that if $e \Downarrow^{\mathsf{co}} o$ holds, then either e terminates on some behavior b, or e diverges. This property is formalized in the next lemma.

Lemma 3. *For any term e and outcome o,* $e \Downarrow^{\mathsf{co}} o \;\to\; e \Downarrow o \;\vee\; e \Downarrow^{\mathsf{co}} \mathit{div}$.

We have proved in Coq that the pretty-big-step semantics shown in Figure 2 yields an operational semantics adequate with respect to the standard big-step evaluation judgment for terminating programs ($t \Rightarrow b$) and with respect to the coinductive big-step evaluation judgment ($t \Rightarrow^{\infty}$) introduced by Leroy and Grall [8, 9] for diverging programs. (The proof requires the excluded middle.)

Theorem 1 (Equivalence with big-step semantics). *For any term t, and for any behavior b (describing either a value or an exception),*

$t \Downarrow b$ *if and only if* $t \Rightarrow b$ *and* $t \Downarrow^{\mathsf{co}} \mathit{div}$ *if and only if* $t \Rightarrow^{\infty}$.

All the results presented so far can be generalized to non-deterministic semantics. In the particular case of a deterministic semantics, such as our call-by-value λ-calculus, we can express the determinacy property as follows.

Lemma 4 (Determinacy). $\forall e o_1 o_2.\;\; e \Downarrow o_1 \;\wedge\; e \Downarrow^{\mathsf{co}} o_2 \;\to\; o_1 = o_2$

As corollaries, we can prove that if a given term e evaluates to a behavior o_1, then it cannot evaluate to a different behavior o_2 and it cannot diverge.

3 Error rules and type soundness proofs

3.1 Explicit error rules

When considering a deterministic language, one can express the type soundness theorem in the form "if a term is well-typed, then it either terminates or diverges". However, for a non-deterministic language, such a statement does not ensure soundness, because a term could execute safely in some execution but get stuck in other executions. For a non-deterministic big-step semantics, the traditional approach to proving type soundness consists of adding explicit *error rules* to the

semantics, and then proving a theorem of the form "if a term is well-typed, then it cannot evaluate to an error".

Adding error rules to a pretty-big-step semantics turns out to be much easier than for a big-step semantics, because we are able to reuse the abort rules for propagating errors to the top level. To describe stuck terms in our language, it suffices to add a behavior err, to state that it satisfies the predicate abort, and to add two error rules, one for variables and one for stuck applications.

$$b \;:=\; \ldots \mid \mathsf{err} \qquad \frac{}{\mathsf{abort}\,\mathsf{err}} \qquad \frac{}{\mathsf{var}\,x \Downarrow \mathsf{err}} \qquad \frac{\forall x t.\ v_1 \neq \mathsf{abs}\,x\,t}{\mathsf{app2}\,v_1\,v_2 \Downarrow \mathsf{err}}$$

3.2 The generic error rule

A classic problem with the introduction of explicit error rules for proving type soundness is that the theorem can be compromised if an error rule is missing. Indeed, if we remove a few error rules, then it makes it *easier* to prove that "if a term is well-typed, then it cannot evaluate to an error". So, the omission of an error rule may hide a flaw in the type system that we want to prove sound.

For a language as simple as the λ-calculus, the error rules are few. However, for a more realistic language, they can be numerous. In such a case, it becomes fairly easy to forget a rule and thereby compromise the adequacy of the type soundness theorem. One could hope to be able to prove (say, in Coq) that a semantics is not missing any error rules. Yet, as far as we know, there is no way of formally stating this property. (The formulation "every term either evaluates to a value or to an error, or diverges" is not appropriate, due to non-determinism.)

In what follows, we explain how a pretty-big-step semantics can be equipped with a *generic error rule*, which directly captures the intuition that "a term should evaluate to an error if no other evaluation rule can be applied". Remark: this intuition was at the source of the work by Gunter and Rémy [6] on *partial proof semantics*, which consists of a specialized proof theory that allows describing derivation trees with exactly one unproved leaf; our approach at handling error rules in a generic manner can be viewed as a realization of Gunter and Rémy's idea of partial proofs within a standard proof theory.

The generic error rule is defined in terms of the *progress judgment*, written $e \downarrow$, which asserts that there exists at least one pretty-big-step evaluation rule whose conclusion matches the term e. The rules defining the progress judgment can be derived in a systematic manner from the pretty-big-step evaluation rules, as described next. An evaluation rule has a conclusion of the form $e \Downarrow o$, a number of evaluation premises and some other premises. The corresponding *progress rule* is obtained by changing the conclusion to $e \downarrow$ (i.e., dropping the outcome o) and by removing all the evaluation premises. The progress judgment associated with the semantics described in Figure 2 is defined in Figure 3.

Then, the generic error rule, shown below, simply asserts that "if a term e cannot make progress ($e \downarrow$ is false) then e should evaluate to an error".

$$\frac{\neg\ (e \downarrow)}{e \Downarrow \mathsf{err}}$$

$$\frac{}{v \downarrow} \qquad \frac{}{\mathsf{app}\, t_1\, t_2 \downarrow} \qquad \frac{\mathsf{abort}\, o}{\mathsf{app1}\, o\, t_2 \downarrow} \qquad \frac{}{\mathsf{app1}\, v_1\, t_2 \downarrow} \qquad \frac{\mathsf{abort}\, o}{\mathsf{app2}\, v\, o \downarrow}$$

$$\frac{}{\mathsf{app2}\,(\mathsf{abs}\, x\, t)\, v \downarrow} \qquad \frac{}{\mathsf{raise}\, t \downarrow} \qquad \frac{\mathsf{abort}\, o}{\mathsf{raise1}\, o \downarrow} \qquad \frac{}{\mathsf{raise1}\, v \downarrow} \qquad \frac{}{\mathsf{try}\, t_1\, t_2 \downarrow}$$

$$\frac{}{\mathsf{try1}\, v\, t \downarrow} \qquad \frac{}{\mathsf{try1}\,(\mathsf{exn}\, v)\, t \downarrow} \qquad \frac{\mathsf{abort}\, o \qquad \forall v.\ o \neq \mathsf{exn}\, v}{\mathsf{try1}\, o\, t \downarrow}$$

Fig. 3. Progress judgment.

We have proved in Coq that using the generic error rule yields evaluation and coevaluation judgments that are equivalent to those obtained with the traditional approach to introducing explicit error rules.

There are two main benefits to using the generic error rule. First, deriving the progress rules from the evaluation rules is easier than deriving explicit error rules. Indeed, instead of having to find out which rules are needed to complete the semantics, we can apply to each of the evaluation rules a very systematic process —so systematic that we believe it could be automated by a Coq plug-in. Second, forgetting a progress rule does not compromise the type soundness theorem. Indeed, omitting a progress rule makes it *easier* to prove that a term evaluates to an error, and therefore makes it *harder* (if not impossible) to prove the statement of the type soundness theorem. To be fair, it should be acknowledged that adding arbitrary progress rules can compromise type soundness. That said, we believe that it is much more unlikely for a researcher to add arbitrary progress rules than to omit a few legitimate rules.

3.3 Type soundness proofs

To give an example of a type soundness proof, we equip our λ-calculus with simple types. For simplicity, we enforce exceptions to carry only values of type int. A source program can be typed using the standard typing judgment, of the form $E \vdash t : T$. We write $\vdash t : T$ when the typing context E is empty. The typing rules for terms are standard, so we do not show them.

To prove type soundness, we first need to consider a typing judgment for intermediate terms, written $\vdash e : T$, and another one for outcomes, written $\vdash o : T$. The proposition $\vdash o : T$ asserts that the outcome o describes either a value of type T, or an exception carrying a value of type int, or the outcome div. Note that err, the error outcome, is never well-typed. The rules defining the new typing judgments appear in Figure 4. The type soundness theorem states that "if a closed term is well-typed, then it cannot evaluate to an error".

Theorem 2 (Type soundness). *For any t and T,* $\quad \vdash t : T \;\rightarrow\; \neg\, t \Downarrow \mathit{err}$.

The proof is conducted by induction on the preservation property: $(e \Downarrow o) \rightarrow (\vdash e : T) \rightarrow (\vdash o : T)$. To see why the above proposition implies the type soundness theorem, it suffices to instantiate e with t, instantiate o with err,

$$\frac{\vdash v : T}{\vdash \mathsf{ret}\, v : T} \qquad \frac{\vdash v : \mathsf{int}}{\vdash \mathsf{exn}\, v : T} \qquad \frac{}{\vdash \mathsf{div} : T} \qquad \frac{\vdash t : T}{\vdash \mathsf{trm}\, t : T} \qquad \frac{\vdash o : S \to T \quad \vdash t : S}{\vdash \mathsf{app1}\, o\, t : T}$$

$$\frac{\vdash v : S \to T \qquad \vdash o : S}{\vdash \mathsf{app2}\, v\, o : T} \qquad \frac{\vdash o : \mathsf{int}}{\vdash \mathsf{raise1}\, o : T} \qquad \frac{\vdash o : T \qquad \vdash t : \mathsf{int} \to T}{\vdash \mathsf{try1}\, o\, t : T}$$

Fig. 4. Typing rules for outcomes and intermediate terms.

and observe that $\vdash \mathsf{err} : T$ is equivalent to False. There are two particularly interesting cases in the proof. First, when the evaluation rule is an abort rule, we need to exploit the fact that a well-typed outcome satisfying abort admits any type. Formally: $(\vdash o : T) \wedge (\mathsf{abort}\, o) \to (\vdash o : T')$. Second, when the evaluation rule is the error rule, we need to establish that if a term is well-typed then it must satisfy the progress judgment. Formally: $(\vdash e : T) \to (e \downarrow)$.

All the other proof cases are straightforward. Compared with the big-step semantics, the pretty-big-step semantics leads to a type soundness proof that involves a slightly larger number of cases, however these proof cases are typically simpler, due to the fact that the evaluation rules have at most two premises. In practice, we have found that having simpler proof cases makes the proofs easier to complete and easier to automate.

In summary, the pretty-big-step semantics, by reusing its abort rules, reduces the amount of work needed for adding error behaviors. It also allows for a generic error rule that makes it faster and less error-prone to add all the error rules. Moreover, even though it requires additional typing rules for intermediate terms, it leads to proofs that involve cases that are simpler and easier to automate.

4 Traces

Traces are typically used to record the interactions of a program with its environment, for example i/o operations. In what follows, we show how to extend the pretty-big-step evaluation rules with traces. A trace describes a sequence of *effects*. Here, an effect, written α, describes a read operation $(\mathsf{in}\, n)$, or a write operation $(\mathsf{out}\, n)$, or the absence of an operation (ϵ). We use the ϵ effect to make the evaluation rules *productive* with respect to traces. Productivity is needed in particular to ensure that a diverging program that does not perform any i/o cannot be associated with arbitrary traces. A trace can be finite or infinite. A finite trace, written τ, consists of a list of effects. An infinite trace, written σ, consists of a stream of effects (i.e., an infinite list). The outcome of a program can be either "termination on a value with a finite trace" or "divergence with an infinite trace". These definitions are summarized below.

$$\alpha := \epsilon \mid \mathsf{in}\, n \mid \mathsf{out}\, n \qquad o := \mathsf{ter}\, \tau\, b \mid \mathsf{div}\, \sigma \qquad (\tau \text{ list of } \alpha, \text{ and } \sigma \text{ stream of } \alpha)$$

In several evaluation rules, we need to append a finite trace to the head of a finite or an infinite trace. We write $\tau \cdot \tau'$ and $\tau \cdot \sigma$ the corresponding concatenation

$$\frac{}{\mathsf{abort}\,(\mathsf{ter}\,\tau\,(\mathsf{exn}\,v))} \qquad \frac{}{\mathsf{abort}\,(\mathsf{div}\,\sigma)}$$

$$\frac{}{v \;\Downarrow\; \mathsf{ter}\,[\epsilon]\,v} \qquad \frac{t_1 \;\Downarrow\; o_1 \qquad \mathsf{app1}\,o_1\,t_2 \;\Downarrow\; o}{\mathsf{app}\,t_1\,t_2 \;\Downarrow\; [\epsilon]\cdot o} \qquad \frac{\mathsf{abort}\,o}{\mathsf{app1}\,o\,t \;\Downarrow\; [\epsilon]\cdot o}$$

$$\frac{t_2 \;\Downarrow\; o_2 \qquad \mathsf{app2}\,v_1\,o_2 \;\Downarrow\; o}{\mathsf{app1}\,(\mathsf{ter}\,\tau\,v_1)\,t_2 \;\Downarrow\; [\epsilon]\cdot\tau\cdot o} \qquad \frac{\mathsf{abort}\,o}{\mathsf{app2}\,v\,o \;\Downarrow\; [\epsilon]\cdot o} \qquad \frac{[x \to v]\,t \;\Downarrow\; o}{\mathsf{app2}\,(\mathsf{abs}\,x\,t)\,(\mathsf{ter}\,\tau\,v) \;\Downarrow\; [\epsilon]\cdot\tau\cdot o}$$

$$\frac{t \;\Downarrow\; o_1 \qquad \mathsf{read1}\,o_1 \;\Downarrow\; o}{\mathsf{read}\,t \;\Downarrow\; [\epsilon]\cdot o} \qquad \frac{\mathsf{abort}\,o}{\mathsf{read1}\,o \;\Downarrow\; [\epsilon]\cdot o} \qquad \frac{}{\mathsf{read1}\,(\mathsf{ter}\,\tau\,tt) \;\Downarrow\; \mathsf{ter}\,([\epsilon]\cdot\tau\cdot[\mathsf{in}\,n])\,n}$$

$$\frac{t \;\Downarrow\; o_1 \qquad \mathsf{write1}\,o_1 \;\Downarrow\; o}{\mathsf{write}\,t \;\Downarrow\; [\epsilon]\cdot o} \qquad \frac{\mathsf{abort}\,o}{\mathsf{write1}\,o \;\Downarrow\; [\epsilon]\cdot o} \qquad \frac{}{\mathsf{write1}\,(\mathsf{ter}\,\tau\,n) \;\Downarrow\; \mathsf{ter}\,([\epsilon]\cdot\tau\cdot[\mathsf{out}\,n])\,tt}$$

Fig. 5. Pretty-big-step semantics with traces.

operations. By extension, we define an operation, written $\tau \cdot o$, to concatenate a finite trace τ to the trace contained in the outcome o. The updated definition for abort and the evaluation rules appear in Figure 5. ($[\cdot]$ denotes a singleton list.)

With traces, the inductive interpretation of the rules is no longer needed because, thanks to the productivity of the rules with respect to the trace, a diverging expression cannot coevaluate to a terminating behavior. We have:

Lemma 5. *For any finite trace* τ, $(e \Downarrow^{\mathsf{co}} \mathit{ter}\,\tau\,v) \;\Leftrightarrow\; (e \;\Downarrow\; \mathit{ter}\,\tau\,v)$.

An important consequence of Lemma 5 is that, when the semantics includes traces, we do not need the inductive judgment $(e \;\Downarrow\; o)$ anymore. In theory, all our reasoning can be conducted using solely the coevaluation judgment. In particular, we should be able to prove a program transformation correct with respect to both terminating and diverging programs through a single coinductive proof. In practice, though, coinductive reasoning in proof assistants such as Coq or Agda remains problematic because they only accept statement of theorems whose conclusion is a coinductive judgment and where all applications of the coinduction hypothesis are guarded by constructors. As soon as we fall out of this basic pattern, we need to resort to heavy encodings in order to transform the statement and the proof in the appropriate form.

The verification of program transformations, one important applications of formal semantics, almost systematically departs from the basic pattern. Their correctness proof typically relies on a simulation diagram establishing that any behavior exhibited by the compiled code is indeed a behavior of the original code. Consider for example a source-to-source translation, written $[\![\cdot]\!]$. Its correctness would typically be captured by a statement of the form $([\![t]\!] \Downarrow^{\mathsf{co}} o) \to \exists o'.\,(o' \approx o) \wedge (t \Downarrow^{\mathsf{co}} o')$, where $o' \approx o$ asserts that o' and o describe the same behavior and contain traces that are bisimilar up to insertion or deletion of a finite number of ϵ between every two items of the traces. (The equivalence relation $\approx$ is defined coinductively, by a single rule with premise $o \approx o'$ and with conclusion

$\epsilon^n \cdot [\alpha] \cdot o \;\approx\; \epsilon^m \cdot [\alpha] \cdot o'$.) Intuitively, such a statement could be established by coinduction, performing a case analysis on the derivation of $[\![t]\!] \Downarrow^{\mathsf{co}} o$ and, in each case, picking the right o' to build the proof of $t \Downarrow^{\mathsf{co}} o'$.

Unfortunately, this form of reasoning currently cannot be mechanized in Coq because the conclusion of the statement is not just a coinductive judgment; indeed, the conclusion starts with an existential quantifier and a conjunction. One possible work-around consists in defining o' as a function of o and t (this definition is non-constructive), and then proving $o' \approx o$ and $t \Downarrow^{\mathsf{co}} o'$, through two independent proofs. These two proofs have a chance of satisfying the guard condition because they conclude on coinductive judgments. Yet, overall, the work-around described here is extremely unpleasant. First, defining the function that produces o' amounts to building the core of a proof term by hand. Second, the process requires one to go three times over the structure of the intended proof: once for the function definition, and once for each of the two coinductive proofs.

We must therefore acknowledge that, with the support for coinduction currently provided by Coq, mechanizing proofs based on pretty-big-step trace semantics appears to be unrealistic in practice. Nevertheless, we hope that further developments of proof assistants could allow us to conduct the intended reasoning without resorting to painful encodings, either by automating the generation of the encoding, or by somehow relaxing the guard condition. We should then be able to reason about both terminating and diverging programs in a single pass.

5 Scaling up to real languages

So far, we have only considered a toy λ-calculus with exceptions. In this section, we explain how to set up pretty-big-step rules for more advanced programming language constructs, such as effectful operations, tuples of arbitrary arity, and C-style for loops. We also show how to handle constructs for which the order of evaluation of the subterms needs to remain deliberately unspecified.

5.1 Factorization of the abort evaluation rules

The pretty-big-step semantics of a realistic language may involve a fair number of intermediate terms. For each intermediate term, we typically need to introduce an abort rule, i.e., a rule with a premise of the form $\mathsf{abort}\, o$, to propagate exceptions, divergence and errors. Fortunately, it is possible to factorize all the abort rules using the *generic abort rule*. This rule formalizes the following intuition: if an intermediate term e is not an exception handler and if one of its arguments is an outcome o that satisfies the predicate abort, then e should directly evaluate to o. The definition of the generic abort rule relies on an auxiliary function, called getout. It is defined in such a way that $\mathsf{getout}\, e$ returns the outcome contained in e (there is at most one), except for exception handlers, which are treated specially. Formally:

$$\begin{array}{l|l} \mathsf{getout}\ (\mathsf{app1}\, o\, t) \equiv \mathsf{Some}\, o & \mathsf{getout}\ (\mathsf{trm}\, t) \equiv \mathsf{None} \\ \mathsf{getout}\ (\mathsf{app2}\, v\, o) \equiv \mathsf{Some}\, o & \mathsf{getout}\ (\mathsf{try1}\, o\, t) \equiv \mathsf{None} \\ \mathsf{getout}\ (\mathsf{raise1}\, o) \equiv \mathsf{Some}\, o & \end{array}$$

The generic abort rule, shown below, replaces the three abort rules from Figure 2.

$$\frac{\mathsf{getout}\, e = \mathsf{Some}\, o \qquad \mathsf{abort}\, o}{e \Downarrow o}$$

Throughout the rest of this section, when we introduce new intermediate terms, we assume the definition of getout to be extended accordingly.

5.2 Side effects

We now extend the source language with side effects. When the evaluation of a term terminates, it produces not just a value or an exception, but also an updated memory store. We therefore update the grammar of outcomes as follows.

$$o \;:=\; \mathsf{ter}\, m\, b \;\mid\; \mathsf{div}$$

The pretty-big-step evaluation judgment now takes the form $e\,{}_{/m} \Downarrow o$, asserting that the evaluation of the term e in the store m has o for outcome. In particular, the proposition $t\,{}_{/m} \Downarrow \mathsf{ter}\, m'\, b$ corresponds to the traditional big-step judgment $t_{/m} \Rightarrow b_{/m'}$ and, similarly, the proposition $t\,{}_{/m} \Downarrow \mathsf{div}$ corresponds to $t_{/m} \Rightarrow^{\infty}$. The evaluation rules are extended so as to take memory stores into account. For example, the first rules for reducing applications are as shown below. Observe that the intermediate term $\mathsf{app1}\, o_1\, t_2$ is evaluated in the store m in which t_1 was evaluated, and not yet in the store produced by t_1. Indeed, at this point, we do not yet know whether the evaluation of t_1 terminates or diverges. In the particular case where t_1 terminates, the store produced by the evaluation of t_1 can be pulled out of the outcome o_1 and used for the evaluation of t_2.

$$\frac{t_1\,{}_{/m} \Downarrow o_1 \qquad \mathsf{app1}\, o_1\, t_2\,{}_{/m} \Downarrow o}{\mathsf{app}\, t_1\, t_2\,{}_{/m} \Downarrow o} \qquad\qquad \frac{t_2\,{}_{/m} \Downarrow o_2 \qquad \mathsf{app2}\, v_1\, o_2\,{}_{/m} \Downarrow o}{\mathsf{app1}\, (\mathsf{ter}\, m\, v_1)\, t_2\,{}_{/m'} \Downarrow o}$$

We end this section with an example of a rule that modifies the store. Consider a term $\mathsf{ref}\, t_1$. Its evaluation goes through an intermediate term $\mathsf{ref1}\, o_1$. If o_1 is a value, then a memory cell is allocated at a fresh location. The updated store is then returned together with the address of the new memory cell.

$$\frac{t_1\,{}_{/m} \Downarrow o_1 \qquad \mathsf{ref1}\, o_1\,{}_{/m} \Downarrow o}{\mathsf{ref}\, t_1\,{}_{/m} \Downarrow o} \qquad\qquad \frac{l \notin \mathrm{dom}(m)}{\mathsf{ref1}\, (\mathsf{ter}\, m\, v)\,{}_{/m'} \Downarrow \mathsf{ter}\, (m[l \mapsto v])\, l}$$

Other rules accessing and updating the memory store follow a similar pattern.

5.3 C-style for loops

We now come back to the example of C-style for loops described in the introduction, revisiting the evaluation rules from Figure 1 using a pretty-big-step semantics. We introduce a single intermediate term, written "$\mathsf{for}\, i\, o\, t_1\, t_2\, t_3$", where $i \in \{1, 2, 3\}$. The pretty-big-step evaluation rules, shown below, are significantly

more concise than their big-step counterpart. Note that we included an abort rule, even though it would typically be covered by the generic abort rule (§5.1).

$$\frac{t_{1\ /m} \Downarrow o_1 \qquad \mathsf{for}\,1\,o_1\,t_1\,t_2\,t_{3\ /m} \Downarrow o}{\mathsf{for}\,t_1\,t_2\,t_{3\ /m} \Downarrow o} \qquad \frac{}{\mathsf{for}\,1\,(\mathsf{ret}\,m\,\mathsf{false})\,t_1\,t_2\,t_{3\ /m'} \Downarrow \mathsf{ret}\,m\,\mathit{tt}}$$

$$\frac{t_{3\ /m} \Downarrow o_3 \qquad \mathsf{for}\,2\,o_3\,t_1\,t_2\,t_{3\ /m} \Downarrow o}{\mathsf{for}\,1\,(\mathsf{ret}\,m\,\mathsf{true})\,t_1\,t_2\,t_{3\ /m'} \Downarrow o} \qquad \frac{t_{2\ /m} \Downarrow o_2 \qquad \mathsf{for}\,3\,o_2\,t_1\,t_2\,t_{3\ /m} \Downarrow o}{\mathsf{for}\,2\,(\mathsf{ret}\,m\,\mathit{tt})\,t_1\,t_2\,t_{3\ /m'} \Downarrow o}$$

$$\frac{\mathsf{for}\,t_1\,t_2\,t_{3\ /m} \Downarrow o}{\mathsf{for}\,3\,(\mathsf{ret}\,m\,\mathit{tt})\,t_1\,t_2\,t_{3\ /m'} \Downarrow o} \qquad \frac{\mathsf{abort}\,o}{\mathsf{for}\,i\,o\,t_1\,t_2\,t_{3\ /m} \Downarrow o}$$

5.4 List of subterms

Consider a tuple expression, written $\mathsf{tuple}\,\overline{t}$, where $\overline{t}$ denotes a list of terms of arbitrary length, and assume a left-to-right evaluation order. The semantics needs to describe the fact that if one of the subterms of the tuple raises an exception or diverge, then the remaining subterms should not be evaluated. In what follows, we describe a technique for evaluating an ordered list of subterms in a way that is not specific to tuples, so that we are able to reuse the same rules for reducing other language constructs that involve lists of subterms (e.g., records).

We introduce an intermediate term, written $\mathsf{list1}\,\overline{t}\,\overline{v}\,K$, where $\overline{v}$ represents the list of values that have already been produced, $\overline{t}$ represents the list of terms remaining to be evaluated, and K denotes the continuation describing what term to transition to once all the subterms have been evaluated. Here, K is a logical function that takes a list of values as arguments and produces an intermediate term. In practice, K is usually a partially-applied constructor.

To evaluate $\mathsf{tuple}\,\overline{t}$, we evaluate $\mathsf{list1}\,\overline{t}\,\mathsf{nil}\,(\mathsf{tuple1})$, where the continuation $\mathsf{tuple1}$ indicates that, when we get the list of values $\overline{v}$ describing the results of the terms $\overline{t}$, we should evaluate the term $\mathsf{tuple1}\,\overline{v}$. This latter term will immediately evaluate to the value $\mathsf{vtuple}\,\overline{v}$. The corresponding evaluation rules are:

$$\frac{\mathsf{list1}\,\overline{t}\,\mathsf{nil}\,(\mathsf{tuple1})_{\ /m} \Downarrow o}{\mathsf{tuple}\,\overline{t}_{\ /m} \Downarrow o} \qquad \frac{}{\mathsf{tuple1}\,\overline{v}_{\ /m} \Downarrow \mathsf{ter}\,m\,(\mathsf{vtuple}\,\overline{v})}$$

It remains to describe the rules involved in the evaluation of $\mathsf{list1}\,\overline{t}\,\overline{v}\,K$. If $\overline{t}$ is empty, we apply (in the logic) the continuation K to $\overline{v}$ and obtain the term from which to continue the evaluation. Otherwise, $\overline{t}$ is of the form $t_1 :: \overline{t}$. In this case, we evaluate the head term t_1, obtaining some outcome o, and we then evaluate the term $\mathsf{list2}\,o\,\overline{t}\,\overline{v}\,K$. If o corresponds to a value, we can save this value at the tail of the list $\overline{v}$ and continue. Otherwise, we can apply the generic abort rule to propagate this outcome directly, skipping the evaluation of the remaining terms $\overline{t}$. The corresponding evaluation rules are shown below.

$$\frac{(K\,\overline{v})_{\ /m} \Downarrow o}{\mathsf{list1}\,\mathsf{nil}\,\overline{v}\,K_{\ /m} \Downarrow o} \qquad \frac{t_{1\ /m} \Downarrow o_1 \quad \mathsf{list2}\,o_1\,\overline{t}\,\overline{v}\,K_{\ /m} \Downarrow o}{\mathsf{list1}\,(t_1 :: \overline{t})\,\overline{v}\,K_{\ /m} \Downarrow o} \qquad \frac{\mathsf{list1}\,\overline{t}\,(\overline{v} +\!\!+ [v_1])\,K_{\ /m} \Downarrow o}{\mathsf{list2}\,(\mathsf{ter}\,m\,v_1)\,\overline{t}\,\overline{v}\,K_{\ /m'} \Downarrow o}$$

5.5 Unspecified order of evaluation

Some programming languages choose to deliberately *not* specify the order of evaluation of the subterms of particular language constructs. For example, Caml does not specify the order of evaluation of the arguments of a function call. In what follows, we explain how to describe the semantics of a list of subterms without specifying the order of evaluation. We use a list $\overline{r}$ whose items are either values or unevaluated terms. Formally, $r := \mathsf{Trm}\, t \mid \mathsf{Val}\, v$, and $\overline{r} := \mathsf{list}\, r$.

We start from an intermediate term $\mathsf{ulist1}\, \overline{t}\, K$, where, as previously, $\overline{t}$ denotes the list of subterms and K is a logical function that denotes the continuation. To evaluate $\mathsf{ulist1}\, \overline{t}\, K$, we evaluate another intermediate term, $\mathsf{ulist2}\, \overline{r}\, K$, where $\overline{r}$ is obtained by mapping the constructor Trm to all the elements of $\overline{t}$. Then, we pick any unevaluated term from the list $\overline{r}$ and evaluate it. We repeat this process until either the evaluation of one of the term diverges or produces an exception, or until all the items in $\overline{r}$ are values. The rules, shown below, involve an intermediate term of the form $\mathsf{ulist3}\, \overline{r_1}\, o\, \overline{r_2}\, K$, where o denotes the outcome that has just been produced, and where $\overline{r_1}$ and $\overline{r_2}$ denote the prefix and the suffix of $\overline{r}$, respectively.

$$\frac{\mathsf{ulist2}\,(\mathsf{map}\,(\mathsf{Trm})\,\overline{t})\,K\;{}_{/m} \Downarrow o}{\mathsf{ulist1}\,\overline{t}\,K\;{}_{/m} \Downarrow o} \qquad \frac{t_1\;{}_{/m} \Downarrow o_1 \qquad \mathsf{ulist3}\,\overline{r_1}\,o_1\,\overline{r_2}\,K\;{}_{/m} \Downarrow o}{\mathsf{ulist2}\,(\overline{r_1} +\!\!+ [\mathsf{Trm}\,t_1] +\!\!+ \overline{r_2})\,K\;{}_{/m} \Downarrow o}$$

$$\frac{\mathsf{ulist2}\,(\overline{r_1} +\!\!+ [\mathsf{Val}\,v_1] +\!\!+ \overline{r_2})\,K\;{}_{/m} \Downarrow o}{\mathsf{ulist3}\,\overline{r_1}\,(\mathsf{ter}\,m\,v_1)\,\overline{r_2}\,K\;{}_{/m'} \Downarrow o} \qquad \frac{(K\,\overline{v})\;{}_{/m} \Downarrow o}{\mathsf{ulist2}\,(\mathsf{map}\,(\mathsf{Val})\,\overline{v})\,K\;{}_{/m} \Downarrow o}$$

5.6 Formalization of core-Caml

To assess the ability of the pretty-big-step semantics to scale up to a realistic programming language, we formalized the semantics of *core-Caml*, both in big-step and in pretty-big-step style. By core-Caml, we refer to the subset of Caml Light made of booleans, integers, tuples, algebraic data types, mutable records, boolean operators (lazy and, lazy or, negation), integer operators (negation, addition, subtraction, multiplication, division), comparison operator, functions, recursive functions, applications, sequences, let-bindings, conditionals (with optional *else* branch), *for* loops and *while* loops, pattern matching (with nested patterns, *as* patterns, *or* patterns, and *when* clauses), *raise* construct, *try-with* construct with pattern matching, and assertions. (The features missing from Caml Light are: floats, mutual recursion, recursive values, *with* construct for records, and arrays. Objects and modules are not covered either.)

Translating the big-step semantics into a pretty-big-step one following the ideas described in this paper was straightforward and took little time. Apart from adapting the rules, the only extra work required consisted of the definition of outcomes and of the abort predicate (4 lines), the definition of the 28 intermediate terms, and the definition of the auxiliary function getout using a simple pattern matching with 22 cases (one case per intermediate term carrying an outcome).

The table shown below quantifies the improvement. It reports on the number of evaluation rules, the number of evaluation premises, and the number of tokens

(excluding head quantified variables, which are typically omitted in paper definitions). It shows that switching to the pretty-big-step semantics reduced the number of the evaluation rules by 38%, reduced the total number of evaluation premises by more than a factor of 2, and reduced the total size of the evaluation rules (as counted by the number of tokens) by 40%.

	rules	premises	tokens
Big-step without divergence	71	83	1540
Big-step with divergence	113	143	2263
Pretty-big-step	70	60	1361

6 Related work

Cousot and Cousot [2] proposed a coinductive big-step characterization of divergence for λ-terms. Leroy and Grall [8, 9] showed how to represent coinductive big-step semantics in a theorem prover such as Coq, and used this semantics to prove that nontrivial program transformations preserve diverging behaviors. They justify the need to introduce separate coinductive rules by observing that naively taking the coinductive interpretation of the standard evaluation rules yields a coevaluation judgment that does not properly characterizes diverging terms. Indeed, there exist terms that diverge but do not coevaluate. Leroy and Grall also explained how to extend their semantics with traces, using two judgments: $t \Rightarrow v/\tau$ asserts that the execution of t produces the value v and a finite trace τ (a list), and $t \Rightarrow^{\infty}/\sigma$ asserts that the execution of t diverges producing an infinite trace σ (a stream). We have shown in this paper, among other things, how to factorize these two separate judgments into a single one.

Following up on earlier work [2], Cousot and Cousot further developed the notion of bi-inductive semantics [3, 4]. These semantics are able to characterize both terminating and diverging executions using a common set of inference rules. Their approach is based on the construction of a least fixed point of a set of evaluation rules with respect to a non-standard ordering, which corresponds neither to induction nor coinduction. By contrast, we have shown in this paper how to achieve the same goal using only the standard notions of induction and coinduction. In their work, Cousot and Cousot also decompose the evaluation rule for application in separate rules. However, their decomposition does not go as far as ours. For example, two of their rules perform the evaluation of the left branch of an application, whereas with the pretty-big-step semantics we only need one such rule.

Nakata and Uustalu [10] propose a coinductive relation that provides a big-step semantics for both terminating and diverging programs, using possibly-infinite traces (coinductive lists) that record all the intermediate memory states of an execution. Formally, they define traces coinductively: $\phi := \langle m \rangle \mid m ::: \phi$. Their (coinductive) big-step evaluation judgment takes the form $t_{/m} \Rightarrow \phi$. Its definition, whose key rules are shown below, is mutually-recursive with another

judgment, $t/\phi \stackrel{*}{\Rightarrow} \phi'$. The definition is quite subtle. It is explained next.

$$\frac{t_{1/m} \Rightarrow \phi \qquad t_2/(m ::: \phi) \stackrel{*}{\Rightarrow} \phi'}{(t_1 \,;\, t_2)_{/m} \Rightarrow \phi'} \qquad \frac{t_{/m} \Rightarrow \phi}{t/\langle m \rangle \stackrel{*}{\Rightarrow} \phi} \qquad \frac{t/\phi \stackrel{*}{\Rightarrow} \phi'}{t/(m ::: \phi) \stackrel{*}{\Rightarrow} (m ::: \phi')}$$

To evaluate a sequence $(t_1 \,;\, t_2)$, we first evaluate t_1 and obtain a trace ϕ. Using the relation $t_2/(m ::: \phi) \stackrel{*}{\Rightarrow} \phi'$, we ensure that the trace ϕ produced by t_1 corresponds to the prefix of the trace ϕ' associated with the term $(t_1 \,;\, t_2)$. If the trace ϕ is finite, then we reach the judgment $t_2/\langle m' \rangle \stackrel{*}{\Rightarrow} \phi''$, where m' denotes the state produced by t_1 and where ϕ'' corresponds to what remains of the trace ϕ' after stripping its prefix ϕ. We can then proceed to the evaluation of t_2 in m'. Otherwise, if the trace ϕ is infinite, then the third rule shown above applies indefinitely, ensuring that the trace ϕ' associated with the term $(t_1 \,;\, t_2)$ is equal (up to bisimilarity) to the trace ϕ produced by t_1.

The manipulation of traces involved with the pretty-big-step semantics is, in our opinion, much simpler for several reasons. First, instead of working with potentially-infinite lists, we use a syntactic disjunction between finite traces and infinite traces, so it is always clear whether we are describing a finite or an infinite execution. Second, we do not need to use an auxiliary, mutually-coinductive judgment to traverse traces; instead, we use a simpler concatenation operation that only needs to traverse finite traces. Third, applying Nakata and Uustalu's approach to a λ-calculus instead of a simple imperative language would require all the rules to be stated in continuation-passing style, because the judgment $t/\phi \stackrel{*}{\Rightarrow} \phi'$ would need to be generalized to the form $K/\phi \stackrel{*}{\Rightarrow} \phi'$, where K denotes a continuation that expects the result of the previous computation (that is, the result stored at the end of the trace ϕ) and produces the term to continue the evaluation from. Such a systematic use of continuations would likely result in fairly obfuscated rules.

Danielsson [5] revisits Nakata and Uustalu's work by defining a corecursive function that yields a big-step semantics for both terminating and diverging programs. This function produces a value of type $(\mathsf{Maybe}\,\mathsf{Value})_\perp$, where the Maybe indicates the possibility of an error and where the bottom represents the partiality monad. The partiality monad involves two constructors: one that carries a value, and one that "delays" the exhibition of a value. Formally, the coinductive definition is $A_\perp := \mathsf{now}\,A \mid \mathsf{later}\,(A_\perp)$. The partiality monad thus corresponds to a degenerated version of potentially-infinite traces, where the spine of a trace does not carry any information; only the tail of a trace, if any, carries a value. Note that, to accommodate non-deterministic semantics, the type $(\mathsf{Maybe}\,\mathsf{Value})_\perp$ needs to be further extended with the non-determinism monad.

In summary, Danielsson's semantics for the λ-calculus consists of a reference interpreter, defined in a formal logic. (It is actually not so straightforward to convince the checker of the guard condition that the definition of the interpreter indeed yields a productive function.) Note that this interpreter should only be used for specification, not for execution, because it is quite inefficient: each bind operation needs to traverse the trace that carries the result that it is binding. Specifying the semantics of a language via an interpreter departs quite

significantly from the traditional statement of a big-step semantics as a relation between a term and a result. We find that pretty-big-step semantics remains much more faithful to big-step semantics, and is thus more likely to be accepted as the reference semantics of a programming language. Moreover, some forms of reasoning, such as reasoning by inversion, are typically easier to conduct when the definition is a relation than when it is a function.

7 Conclusion

In this paper, we addressed the duplication problem associated with big-step semantics by introducing pretty-big-step semantics. Pretty-big-semantics rely on four key ingredients: (1) a breakdown of complex rules into a larger number of simpler rules, (2) a grammar of intermediate terms for ensuring that rules are applied in the appropriate order, (3) an explicit constant div to represent divergence, and (4) an inductive and a coinductive interpretation of the same set of reduction rules. Pretty-big-step semantics accommodate the introduction of a generic error rule for conducting type soundness proofs, and they scale up to realistic programming languages. Moreover, they can easily be extended with traces, in which case the behavior of both terminating and diverging programs is adequately captured by the coinductive evaluation judgment alone.

Acknowledgments I am grateful to Xavier Leroy for very useful feedback.

References

1. Arthur Charguéraud. *Characteristic Formulae for Mechanized Program Verification*. PhD thesis, Université Paris-Diderot, 2010.
2. Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretation. In *POPL*, pages 83–94, 1992.
3. Patrick Cousot and Radhia Cousot. Bi-inductive structural semantics: (extended abstract). *Electronic Notes Theoretical Computer Sciences*, 192(1):29–44, 2007.
4. Patrick Cousot and Radhia Cousot. Bi-inductive structural semantics. *Information and Computation*, 207(2):258–283, 2009.
5. Nils Anders Danielsson. Operational semantics using the partiality monad. In *ICFP*, pages 127–138. ACM, 2012.
6. Carl A. Gunter and Didier Rémy. A proof-theoretic assessment of runtime type errors. Research Report 11261-921230-43TM, AT&T Bell Laboratories, 1993.
7. Gilles Kahn. Natural semantics. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.
8. Xavier Leroy. Coinductive big-step operational semantics. In *European Symposium on Programming (ESOP)*, volume 3924 of *LNCS*, pages 54–68. Springer, 2006.
9. Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *CoRR*, abs/0808.0586, 2008.
10. Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for while. In *TPHOLs*, volume 5674 of *LNCS*, pages 375–390. Springer, 2009.
11. G. D. Plotkin. A structural approach to operational semantics. Internal Report DAIMI FN-19, Department of Computer Science, Aarhus University, 1981.
12. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.