A Lightweight Approach to Avoiding Duplication in Big-Step Semantics

C. Bach Poulsen*, P.D. Mosses*

Department of Computer Science
Swansea University
Singleton Park, Swansea
SA2 8PP, UK

Abstract

Structural operational semantic specifications come in different styles: small-step and big-step. A problem with the big-step style is that specifying divergence and abrupt termination gives rise to annoying duplication. We present a novel lightweight approach to representing divergence and abrupt termination in big-step semantics that avoids the duplication problem, and uses fewer rules and premises for representing divergence than previous approaches in the literature.

Keywords: Structural operational semantics, SOS, Coinduction, Big-step semantics, Natural semantics, Small-step semantics

1. Introduction

Formal specifications concisely capture the meaning of programs and programming languages and provide a valuable tool for reasoning about them. A particularly attractive trait of *structural specifications* is that one can prove properties of programs and programming languages using well-known reasoning techniques, such as induction for finite structures, and coinduction for possibly-infinite ones.

In this article we consider the well-known variant of structural specifications called *structural operational semantics* (SOS) [1]. SOS rules are generally formulated in one of two styles: *small-step*, relating intermediate states in a transition system; and *big-step* (also known as *natural semantics* [2]), relating states directly to final outcomes, such as values, stores, traces, etc. Each style has its merits and drawbacks. For example, small-step is regarded as superior for specifying interleaving, whereas big-step is regarded as superior for compiler correctness proofs [3; 4; 5] and for efficient interpreters [6; 7]. Different styles can also be used for specifying different fragments of the same language.

Big-step SOS rules, however, suffer from a well-known *duplication problem* [8]. Consider, for example, the following rule for sequential composition:

$$\frac{(c_1, \sigma) \Rightarrow \sigma' \qquad (c_2, \sigma') \Rightarrow \sigma''}{(c_1; c_2, \sigma) \Rightarrow \sigma''}\ \text{B-Seq}$$

This rule is *inductively* defined and covers the sequential composition of all *finite* computations for c_1 and c_2. But what if either c_1 or c_2 are *infinite* computations, i.e., if they *diverge*? The traditional approach to representing this in big-step SOS is to introduce a separate, *coinductively* defined, relation $\overset{\infty}{\Rightarrow}$:

$$\frac{(c_1, \sigma) \overset{\infty}{\Rightarrow}}{(c_1; c_2, \sigma) \overset{\infty}{\Rightarrow}}\ \text{B-}\infty\text{-Seq1} \qquad \frac{(c_1, \sigma) \Rightarrow \sigma' \qquad (c_2, \sigma') \overset{\infty}{\Rightarrow}}{(c_1; c_2, \sigma) \overset{\infty}{\Rightarrow}}\ \text{B-}\infty\text{-Seq2}$$

*Corresponding author

Here, the first premise in B-∞-Seq2 and B-Seq is duplicated. If the language can throw exceptions we need even more (inductive) rules in order to correctly propagate such abrupt termination, which further increases duplication:[1]

$$\frac{(c_1,\sigma) \Rightarrow \mathsf{exc}(v)}{(c_1;c_2,\sigma) \Rightarrow \mathsf{exc}(v)}\text{B-Exc-Seq1} \qquad \frac{(c_1,\sigma) \Rightarrow \sigma' \quad (c_2,\sigma') \Rightarrow \mathsf{exc}(v)}{(c_1;c_2,\sigma) \Rightarrow \mathsf{exc}(v)}\text{B-Exc-Seq2}$$

The semantics of sequential composition is now given by five rules with eight premises, where two of these premises are duplicates (i.e., the first premise in B-∞-Seq2, and B-Exc-Seq2).

Recent work by Charguéraud [8] shows how to alleviate this duplication problem by using the novel *pretty-big-step* style of big-step SOS rules. The style breaks big-step rules into smaller rules such that each rule fully evaluates a single sub-term and then continues the evaluation. The pretty-big-step rules introduces an intermediate expression constructor, $\mathsf{seq2}$, and uses *outcomes* o to range over either convergence at a store σ ($\mathsf{conv}\ \sigma$), divergence (div), or abrupt termination ($\mathsf{exc}(v)$):

$$\frac{(c_1,\sigma) \Downarrow o_1 \quad (\mathsf{seq2}\ o_1\ c_2,\sigma 2) \Downarrow o}{(c_1;c_2,\sigma) \Downarrow o}\text{P-Seq1} \quad \frac{(c,\sigma) \Downarrow o}{(\mathsf{seq2}\ (\mathsf{conv}\ \sigma)\ c,\sigma_0) \Downarrow o}\text{P-Seq2} \quad \frac{\mathsf{abort}(o)}{(\mathsf{seq2}\ o\ c_2,\sigma) \Downarrow o}\text{P-Seq-Abort}$$

Following Charguéraud, these rules have a *dual* interpretation: inductive and coinductive. They use an abort predicate to propagate either exceptions or abrupt termination. This predicate is specified once-and-for-all and not on a construct-by-construct basis:

$$\frac{}{\mathsf{abort}(\mathsf{div})}\text{Abort-Div} \qquad \frac{}{\mathsf{abort}(\mathsf{exc}(v))}\text{Abort-Exc}$$

Using pretty-big-step, the semantics for sequential composition counts three rules with three premises, and two generic rules for the abort predicate. It avoids duplication by breaking the original inductive big-step rule B-Seq into smaller rules. But it also increases the number of rules compared with the original inductive big-step rules B-Seq. For semantics with rules with more premises but without abrupt termination, pretty-big-step semantics sometimes *increases* the number of rules and premises compared with traditional inductive big-step rules.

In this paper we adopt and adapt the technique (due to Klin [9, p. 216]) for modular abrupt termination in small-step Modular SOS [9]. Adapting this technique to big-step SOS avoids the duplication problem but uses fewer rules than pretty-big-step semantics. The idea is to make program states and result states record an additional status flag (ranged over by δ) which indicates that the current state is either convergent ('$\downarrow$'), divergent ('$\uparrow$'), or abruptly terminated ('$\mathsf{exc}(v)$'):

$$\frac{(c_1,\sigma,\downarrow) \Rightarrow \sigma',\delta \quad (c_2,\sigma',\delta) \Rightarrow \sigma'',\delta'}{(c_1;c_2,\sigma,\downarrow) \Rightarrow \sigma'',\delta'}\text{L-Seq}$$

Here, rules continue only if they are in a convergent state. In order to propagate divergence or abrupt termination, we use pretty-big-step inspired abort rules:

$$\frac{}{(c,\sigma,\uparrow) \Rightarrow \sigma',\uparrow}\text{L-Div} \qquad \frac{}{(c,\sigma,\mathsf{exc}(v)) \Rightarrow \sigma',\mathsf{exc}(v)}\text{L-Exc}$$

The abort rules say that all parts of the state except for the status flag are computationally irrelevant, hence the use of the free variables σ' in L-Div and L-Exc. This approach uses fewer rules and premises than both the traditional and the pretty-big-step approach: just a single rule for sequential composition with two premises, which is straightforwardly derived from the original inductive rule B-Seq. We call this style

[1]It is also possible to propagate exceptions automatically when they occur in tail-positions in big-step rules. This would make rule B-Exc-Seq2 redundant and eliminate some of the redundancy for the sequential composition construct. It would, however, require extra restrictions in the standard inductive rule for sequential composition. We emphasise that the duplication problem occurs in any case for constructs with more than two premises.

of rules *state-based big-step semantics*. State-based big-step semantics supports reasoning about possibly-infinite computations on a par with traditional big-step approaches as well as small-step semantics, and eliminates the big-step duplication problem for diverging and abruptly terminating computations.

The rest of this paper is structured as follows. We first introduce a simple While-language and recall how possibly-diverging computations in small-step semantics are traditionally expressed (Section 2). Next, we recall traditional approaches to representing possibly-diverging computations in big-step semantics, and how to prove the equivalence between semantics for these approaches (Section 3). Thus equipped, we make the following contributions:

- We present a novel lightweight approach to representing divergence and abrupt termination (Section 4) which alleviates the duplication problem in big-step semantics. For all examples considered by the authors, including applicative and imperative languages, our approach straightforwardly allows expressing divergence and abrupt termination in a way that does not involve introducing or modifying rules for existing constructs. Our approach is as expressive as small-step semantics or pretty-big-step semantics [8], but uses fewer rules and premises.
- We consider how the approach scales to more interesting language features (Section 5), including non-deterministic interactive input operation. A problem in this connection is that the traditional proof method for relating diverging computations in small-step and big-step SOS work only for deterministic semantics. We provide a generalised proof method which suffices to relate small-step and big-step semantics with non-deterministic interactive input.
- Implicitly-Modular SOS (I-MSOS) [10] is a variant of SOS that allows for implicit propagation of auxiliary entities. We enhance the definition of I-MSOS and show how this suffices for transforming standard big-step rules into variants with generic divergence (Section 6).

Our experiments show that state-based big-step SOS with divergence and abrupt termination uses fewer rules than previous approaches. The conciseness comes at the cost of states recording irrelevant information (such as the structure of the store) in abruptly terminated or divergent result states. We discuss and compare with previous approaches in Section 7 before concluding in Section 8.

2. The While-Language and its Small-Step Semantics

We use a simple While-language throughout this article. Its syntax is:

$\mathit{Var} \ni x ::= \mathrm{x} \mid \mathrm{y} \mid \ldots$	Variables
$\mathbb{N} \ni n ::= 0 \mid 1 \mid \ldots$	Natural numbers
$\mathit{Cmd} \ni c ::= \mathsf{skip} \mid \mathsf{alloc}\ x \mid x := e \mid c; c \mid \mathsf{if}\ e\ c\ c \mid \mathsf{while}\ e\ c$	Commands
$\mathit{Val} \ni v ::= \mathsf{null} \mid n$	Values
$\mathit{Expr} \ni e ::= v \mid x \mid e \oplus e$	Expressions
$\mathit{bop} \in \{+, -, *\}$	Binary operations on natural numbers

Here, null is a special value used for uninitialised locations, and is assumed not to occur in source programs. Let stores $\sigma \in \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val}$ be finite maps from variables to values. We use $\sigma(x)$ to denote the value that variable x is mapped to in the store σ. The notation $\sigma[x \mapsto v]$ denotes the update of store σ' with value v at variable x. We use $\mathrm{dom}(\sigma)$ to denote the domain of a map, and $\{x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots\}$ to denote a map which binds x_1 to v_1, x_2 to v_2, etc. We write $\oplus(n_1, n_2)$ for the result of applying the primitive binary operation $\oplus$ to n_1 and n_2. The remainder of this section introduces a mixed big-step and small-step semantics for this language, and introduces conventions we shall use.

$$\boxed{(e,\sigma) \Rightarrow_{\mathrm{E}} v}$$

$$\frac{}{(v,\sigma) \Rightarrow_{\mathrm{E}} v}\ \text{E-Val} \qquad \frac{x \in \mathrm{dom}(\sigma)}{(x,\sigma) \Rightarrow_{\mathrm{E}} \sigma(x)}\ \text{E-Var} \qquad \frac{(e_1,\sigma) \Rightarrow_{\mathrm{E}} n_1 \qquad (e_2,\sigma) \Rightarrow_{\mathrm{E}} n_2}{(e_1 \oplus e_2,\sigma) \Rightarrow_{\mathrm{E}} \oplus(n_1,n_2)}\ \text{E-Bop}$$

Figure 1: Big-step semantics for expressions

$$\boxed{(c,\sigma) \rightarrow (c',\sigma')}$$

$$\frac{x \notin \mathrm{dom}(\sigma)}{(\mathsf{alloc}\ x,\sigma) \rightarrow (\mathsf{skip},\sigma[x \mapsto \mathsf{null}])}\ \text{S-Alloc} \qquad \frac{x \in \mathrm{dom}(\sigma) \qquad (e,\sigma) \Rightarrow_{\mathrm{E}} v}{(x := e,\sigma) \rightarrow (\mathsf{skip},\sigma[x \mapsto v])}\ \text{S-Assign}$$

$$\frac{(c_1,\sigma) \rightarrow (c_1',\sigma')}{(c_1;c_2,\sigma) \rightarrow (c_1';c_2,\sigma')}\ \text{S-Seq} \qquad \frac{}{(\mathsf{skip};c_2,\sigma) \rightarrow (c_2,\sigma)}\ \text{S-SeqSkip}$$

$$\frac{(e,\sigma) \Rightarrow_{\mathrm{E}} v \qquad v \neq 0}{(\mathsf{if}\ e\ c_1\ c_2,\sigma) \rightarrow (c_1,\sigma)}\ \text{S-If} \qquad \frac{(e,\sigma) \Rightarrow_{\mathrm{E}} 0}{(\mathsf{if}\ e\ c_1\ c_2,\sigma) \rightarrow (c_2,\sigma)}\ \text{S-IfZ}$$

$$\frac{(e,\sigma) \Rightarrow_{\mathrm{E}} v \qquad v \neq 0}{(\mathsf{while}\ e\ c,\sigma) \rightarrow (c;\mathsf{while}\ e\ c,\sigma)}\ \text{S-While} \qquad \frac{(e,\sigma) \Rightarrow_{\mathrm{E}} 0}{(\mathsf{while}\ e\ c,\sigma) \rightarrow (\mathsf{skip},\sigma)}\ \text{S-WhileZ}$$

Figure 2: Small-step semantics for commands

2.1. Big-Step Expression Evaluation Relation

Expressions in our language do not affect the store and cannot diverge, although they can fail to produce a value. Big-step rules for evaluating expressions are given in Figure 1.[2] A judgment $(e,\sigma) \Rightarrow_{\mathrm{E}} v$ says that evaluating e in the store σ results in value v, and does not affect the store.

2.2. Small-Step Command Transition Relation

Commands have side-effects and can diverge. Figure 2 defines a small-step transition relation for commands, using the previously defined big-step semantics for expressions. The transition relation is defined for states consisting of pairs of a command c and a store σ. A judgment $(c,\sigma) \rightarrow (c',\sigma')$ asserts the possibility of a transition from the state (c,σ) to the state (c',σ').

2.3. Finite Computations

In small-step semantics, computations are given by sequences of transitions. The '$\rightarrow^*$' relation is the reflexive-transitive closure of the transition relation for commands, which contains the set of all computations with finite transition sequences:

$$\boxed{(c,\sigma) \rightarrow^* (c',\sigma')}$$

$$\frac{}{(c,\sigma) \rightarrow^* (c,\sigma)}\ \text{Refl}* \qquad \frac{(c,\sigma) \rightarrow (c',\sigma') \qquad (c',\sigma') \rightarrow^* (c'',\sigma'')}{(c,\sigma) \rightarrow^* (c'',\sigma'')}\ \text{Trans}*$$

[2]Following Reynolds [11], such functions are *trivial*, and their evaluation could have been given in terms of an auxiliary function instead. It is useful to model it as a relation for the purpose of our approach to abrupt termination. We discuss this in Section 5.

The following factorial function is an example of a program with a finite sequence of transitions:

$$\begin{array}{lll} \mathit{fac}\ n & \equiv & \textsf{alloc}\ \mathrm{c};\ \mathrm{c} := n; \\ & & \textsf{alloc}\ \mathrm{r};\ \mathrm{r} := 1; \\ & & \textsf{while}\ \mathrm{c}\ (\mathrm{r} := (\mathrm{r} * \mathrm{c}); \\ & & \qquad\qquad \mathrm{c} := (\mathrm{c} - 1)) \end{array}$$

Here, '$\mathit{fac}\ n \equiv \ldots$' defines a function *fac* that, given a natural number n, produces a program calculating the factorial of n. Using '$\cdot$' to denote the empty map, we can use $\to^*$ to calculate:

$$(\mathit{fac}\ 4, \cdot) \to^* (\textsf{skip}, \{\mathrm{c} \mapsto 0, \mathrm{r} \mapsto 24\})$$

This calculation is performed by constructing the finite derivation tree whose conclusion is the judgment above.

2.4. Infinite Computations

The following rule for $\overset{\infty}{\to}$ is *coinductively* defined, and can be used to reason about *infinite sequences* of transitions:

$$\boxed{(c, \sigma) \overset{\infty}{\to}}$$

$$\textsf{co}\,\frac{(c, \sigma) \to (c', \sigma') \qquad (c', \sigma') \overset{\infty}{\to}}{(c, \sigma) \overset{\infty}{\to}}\,\mathrm{Trans}\infty$$

Here and throughout this article, we use $\textsf{co}$ on the left of rules to indicate that the relation is coinductively defined by that set of rules.[3] We assume that the reader is familiar with the basics of coinduction (for introductions see, e.g., [3; 13; 14]).

Usually, coinductively defined rules describe both finite and infinite derivation trees. However, since there are no axioms for '$\overset{\infty}{\to}$', it cannot be used to construct finite derivation trees, whereby it contains exactly the set of all states with infinite sequences of transitions.

Using the coinduction proof principle allows us to construct infinite derivation trees. For example, we can prove that $\textsf{while}\ 1\ \textsf{skip}$ diverges by using $(\textsf{while}\ 1\ \textsf{skip}, \cdot) \overset{\infty}{\to}$ as our *coinduction hypothesis.* In the following derivation tree, that hypothesis is applied at the point in the derivation tree marked CIH. This constructs an infinite branch of the derivation tree:

$$\dfrac{\dfrac{(1, \cdot) \Rightarrow_{\mathrm{E}} 1 \qquad 1 \neq 0}{\begin{array}{l}(\textsf{while}\ 1\ \textsf{skip}, \cdot) \to \\ \quad (\textsf{skip}; \textsf{while}\ 1\ \textsf{skip}, \cdot)\end{array}} \qquad \dfrac{\dfrac{}{\begin{array}{l}(\textsf{skip}; \textsf{while}\ 1\ \textsf{skip}, \cdot) \to \\ \quad (\textsf{while}\ 1\ \textsf{skip}, \cdot)\end{array}} \qquad \dfrac{\vdots}{(\textsf{while}\ 1\ \textsf{skip}, \cdot) \overset{\infty}{\to}}\,\mathrm{CIH}}{(\textsf{skip}; \textsf{while}\ 1\ \textsf{skip}, \cdot) \overset{\infty}{\to}}}{(\textsf{while}\ 1\ \textsf{skip}, \cdot) \overset{\infty}{\to}}$$

There also exists an infinite derivation tree whose conclusion is:

$$(\textsf{alloc}\ \mathrm{x}; \mathrm{x} := 0; \textsf{while}\ 1\ (\mathrm{x} := +(\mathrm{x}, 1)), \cdot) \overset{\infty}{\to}$$

Proving this is slightly more involved: after two applications of Trans∞, the goal to prove is:

$$(\textsf{while}\ 1\ (\mathrm{x} := +(\mathrm{x}, 1)), \{\mathrm{x} \mapsto 0\}) \overset{\infty}{\to}$$

[3]This notation for coinductively defined relations is a variation of Cousot and Cousot's [12] notation for distinguishing inductively and coinductively (or *positively* and *negatively*) defined relations.

We might try to use this goal as the coinduction hypothesis. But after three additional applications of Trans∞, we get a goal with a store $\{x \mapsto 1\}$ that does not match this coinduction hypothesis:

$$(\mathsf{while}\ 1\ (x := +(x, 1)), \{x \mapsto 1\}) \overset{\infty}{\rightarrow}$$

The problem here is that the store changes with each step. Instead, we first prove the following straightforward lemma by coinduction:

$$\text{for any } n,\ (\mathsf{while}\ 1\ (x := +(x, 1)), \{x \mapsto n\}) \overset{\infty}{\rightarrow}$$

Now, by four applications of Trans∞ in the original proof statement, we get a goal that matches our lemma, which completes the proof.

2.5. Proof Conventions

The formal results we prove in this article about our example language are formalised in Coq and are available at: `http://www.plancomps.org/jlamp2015/`

Coq is based on the Calculus of Constructions [15; 16], which embodies a variant of constructive logic. Working within this framework, classical proof arguments, such as the law of excluded middle, are not provable for arbitrary propositions. In spite of embodying a constructive logic, Coq allows us to assert classical arguments as axioms, which are known to be consistent [17] with Coq's logic. Some of the proofs in this article rely on the law of excluded middle. For the reader concerned with implementing proofs in Coq, or other proof assistants or logics based on constructive reasoning, we follow the convention of Leroy and Grall [3] and explicitly mark proofs that rely on the law of excluded middle "*(classical)*".

3. Big-Step Semantics and Their Variants

We recall different variants of big-step semantics from the literature, and illustrate their use on the While-language defined in previous section (always extending the big-step semantics for expressions defined in Figure 1).

3.1. Inductive Big-Step Semantics

The big-step rules in Figure 3 inductively define a big-step relation, where judgments of the form $(c, \sigma) \Rightarrow_{\mathrm{B}} \sigma'$ assert that a command c evaluated in store σ terminates with a final store σ'. This corresponds to arriving at a state (skip, σ) by analogy with the small-step semantics in Section 2. Being inductively defined, the relation does not contain diverging programs.

Example 1. $\neg\ \exists\sigma.\ (\mathsf{while}\ 1\ \mathsf{skip}, \cdot) \Rightarrow_{\mathrm{B}} \sigma$

Proof. We prove that $\exists\sigma.\ (\mathsf{while}\ 1\ \mathsf{skip}, \cdot) \Rightarrow_{\mathrm{B}} \sigma$ implies falsity. The proof proceeds by eliminating the existential in the premise, and by induction on the structure of $\Rightarrow_{\mathrm{B}}$. □

We can, however, construct a finite derivation tree for our factorial program:

$$(\mathit{fac}\ 4, \cdot) \Rightarrow_{\mathrm{B}} \{c \mapsto 0, r \mapsto 24\}$$

The following theorem proves the correspondence between inductive big-step derivation trees and derivation trees for sequences of small-step transitions:

Theorem 2. *$(c, \sigma) \rightarrow^* (\mathsf{skip}, \sigma')$ iff $(c, \sigma) \Rightarrow_{\mathrm{B}} \sigma'$.*

Proof. The small-to-big direction follows by induction on $\rightarrow^*$ using Lemma 3. The big-to-small direction follows by induction on $\Rightarrow_{\mathrm{B}}$, using the transitivity of $\rightarrow^*$ and Lemma 4. □

Lemma 3. *If $(c, \sigma) \rightarrow (c', \sigma')$ and $(c', \sigma') \Rightarrow_{\mathrm{B}} \sigma''$ then $(c, \sigma) \Rightarrow_{\mathrm{B}} \sigma''$.*

$$\boxed{(c,\sigma) \Rightarrow_{\mathrm{B}} \sigma'}$$

$$\frac{}{(\mathsf{skip},\sigma) \Rightarrow_{\mathrm{B}} \sigma}\text{B-Skip} \qquad \frac{x \notin \mathrm{dom}(\sigma)}{(\mathsf{alloc}\ x,\sigma) \Rightarrow_{\mathrm{B}} \sigma[x \mapsto \mathsf{null}]}\text{B-Alloc}$$

$$\frac{x \in \mathrm{dom}(\sigma) \qquad (e,\sigma) \Rightarrow_{\mathrm{E}} v}{(x := e,\sigma) \Rightarrow_{\mathrm{B}} \sigma[x \mapsto v]}\text{B-Assign} \qquad \frac{(c_1,\sigma) \Rightarrow_{\mathrm{B}} \sigma' \qquad (c_2,\sigma') \Rightarrow_{\mathrm{B}} \sigma''}{(c_1;c_2,\sigma) \Rightarrow_{\mathrm{B}} \sigma''}\text{B-Seq}$$

$$\frac{(e,\sigma) \Rightarrow_{\mathrm{E}} v \qquad v \neq 0 \qquad (c_1,\sigma) \Rightarrow_{\mathrm{B}} \sigma'}{(\mathsf{if}\ e\ c_1\ c_2,\sigma) \Rightarrow_{\mathrm{B}} \sigma'}\text{B-If} \qquad \frac{(e,\sigma) \Rightarrow_{\mathrm{E}} 0 \qquad (c_2,\sigma) \Rightarrow_{\mathrm{B}} \sigma'}{(\mathsf{if}\ e\ c_1\ c_2,\sigma) \Rightarrow_{\mathrm{B}} \sigma'}\text{B-IfZ}$$

$$\frac{\begin{array}{c}(e,\sigma) \Rightarrow_{\mathrm{E}} v \qquad v \neq 0 \qquad (c,\sigma) \Rightarrow_{\mathrm{B}} \sigma' \\ (\mathsf{while}\ e\ c,\sigma') \Rightarrow_{\mathrm{B}} \sigma''\end{array}}{(\mathsf{while}\ e\ c,\sigma) \Rightarrow_{\mathrm{B}} \sigma''}\text{B-While} \qquad \frac{(e,\sigma) \Rightarrow_{\mathrm{E}} 0}{(\mathsf{while}\ e\ c,\sigma) \Rightarrow_{\mathrm{B}} \sigma}\text{B-WhileZ}$$

Figure 3: Big-step semantics for commands

$$\boxed{(c,\sigma) \overset{\infty}{\Rightarrow}}$$

$$\mathsf{co}\frac{(c_1,\sigma) \overset{\infty}{\Rightarrow}}{(c_1;c_2,\sigma) \overset{\infty}{\Rightarrow}}\text{D-Seq1} \qquad \mathsf{co}\frac{(c_1,\sigma) \Rightarrow_{\mathrm{B}} \sigma' \qquad (c_2,\sigma') \overset{\infty}{\Rightarrow}}{(c_1;c_2,\sigma) \overset{\infty}{\Rightarrow}}\text{D-Seq2}$$

$$\mathsf{co}\frac{(e,\sigma) \Rightarrow_{\mathrm{E}} v \qquad v \neq 0 \qquad (c_1,\sigma) \overset{\infty}{\Rightarrow}}{(\mathsf{if}\ e\ c_1\ c_2,\sigma) \overset{\infty}{\Rightarrow}}\text{D-If} \qquad \mathsf{co}\frac{(e,\sigma) \Rightarrow_{\mathrm{E}} 0 \qquad (c_2,\sigma) \overset{\infty}{\Rightarrow}}{(\mathsf{if}\ e\ c_1\ c_2,\sigma) \overset{\infty}{\Rightarrow}}\text{D-IfZ}$$

$$\mathsf{co}\frac{(e,\sigma) \Rightarrow_{\mathrm{E}} v \qquad v \neq 0 \qquad (c,\sigma) \overset{\infty}{\Rightarrow}}{(\mathsf{while}\ e\ c,\sigma) \overset{\infty}{\Rightarrow}}\text{D-WhileBody} \qquad \mathsf{co}\frac{\begin{array}{c}(e,\sigma) \Rightarrow_{\mathrm{E}} v \qquad v \neq 0 \qquad (c,\sigma) \Rightarrow_{\mathrm{B}} \sigma' \\ (\mathsf{while}\ e\ c,\sigma') \overset{\infty}{\Rightarrow}\end{array}}{(\mathsf{while}\ e\ c,\sigma) \overset{\infty}{\Rightarrow}}\text{D-While}$$

Figure 4: Big-step semantics for diverging commands (extending Figure 3)

Proof. Straightforward proof by induction on $\rightarrow$. □

Lemma 4. $(c_1,\sigma) \rightarrow^* (c_1',\sigma')$ *implies* $(c_1;c_2,\sigma) \rightarrow^* (c_1';c_2,\sigma')$

Proof. Straightforward proof by induction on $\rightarrow^*$. □

3.2. Coinductive Big-Step Divergence Predicate

The rules in Figure 4 coinductively define a big-step divergence predicate. Like '$\overset{\infty}{\rightarrow}$', the '$\overset{\infty}{\Rightarrow}$' predicate has no axioms so the rules describe exactly the set of all diverging computations. Divergence arises in a single branch of a derivation tree, while other branches may converge. Recalling our program from Section 2:

$$\textit{while-assign} \ \equiv\ \mathsf{alloc}\ \mathrm{x}; \mathrm{x} := 0; \mathsf{while}\ 1\ (\mathrm{x} := +(\mathrm{x},1))$$

Here, alloc x and x := 0 converge and the while command diverges. The big-step divergence predicate uses the standard inductive big-step relation for converging branches. Consequently, we need *both* the rules in Figures 3 and 4 to express divergence. This gives rise to a multitude of rules for each construct, and leads to duplication of premises between the rules. For example, the set of finite and infinite derivation trees whose conclusion is a sequential composition requires three rules: B-Seq, D-Seq1, and D-Seq2. Here, the premise

$(c_1, \sigma) \Rightarrow_{\mathrm{B}} \sigma'$ is duplicated between the rules, giving rise to the so-called duplication problem with big-step semantics. Theorem 5 proves the correspondence between $\overset{\infty}{\to}$ and $\overset{\infty}{\Rightarrow}$.

Theorem 5. $(c, \sigma) \overset{\infty}{\to}$ *iff* $(c, \sigma) \overset{\infty}{\Rightarrow}$

Proof (classical). The small-to-big direction follows by coinduction using Lemma 6 and Lemma 7 (which relies on classical arguments) for case analysis. The other direction follows by coinduction and Lemma 8. □

Lemma 6. *Either* $(c, \sigma) \to^* (\mathsf{skip}, \sigma')$ *or* $(c, \sigma) \overset{\infty}{\to}$.

Proof (classical). By the law of excluded middle and classical reasoning. □

Lemma 7. *If* $(c_1, \sigma) \to^* (c_1', \sigma')$ *and* $(c_1; c_2, \sigma) \overset{\infty}{\to}$*, then* $(c_1'; c_2, \sigma') \overset{\infty}{\to}$.

Proof. Straightforward proof by induction on $\to^*$, using the fact that $\to$ is deterministic. □

Lemma 8. *If* $(c, \sigma) \overset{\infty}{\Rightarrow}$ *then* $\exists c', \sigma'.\ \Big((c, \sigma) \to (c', \sigma') \ \wedge\ (c', \sigma') \overset{\infty}{\Rightarrow}\Big)$.

Proof. Straightforward proof by structural induction on the command c, using Theorem 2 in the sequential composition case. □

3.3. Pretty-Big-Step Semantics

The idea behind pretty-big-step semantics is to break big-step rules into intermediate rules, so that each rule fully evaluates a single sub-term and then continues the evaluation. Continuing evaluation may involve either further computation, or propagating any divergence or abrupt termination that arose during evaluation of a sub-term. Following Charguéraud [8], we introduce so-called *semantic constructors* for commands and *outcomes* for indicating convergence or divergence:

$$\mathit{SemCmd} \ni C ::= c \mid \mathsf{assign2}\ x\ v \mid \mathsf{seq2}\ o\ c \mid \mathsf{if2}\ v\ c\ c \mid \mathsf{while2}\ v\ e\ c \mid \mathsf{while3}\ o\ e\ c$$

$$\mathit{Outcome} \ni o ::= \mathsf{conv}\ \sigma \mid \mathsf{div}$$

The added constructors are used to distinguish whether evaluation should continue.

For example, defining '$\Downarrow$' as our pretty-big-step relation, the rules defining sequential composition are now expressed using the semantic constructor $\mathsf{seq2}$:

$$\frac{(c_1, \sigma) \Downarrow o_1 \qquad (\mathsf{seq2}\ o_1\ c_2, \sigma) \Downarrow o}{(c_1; c_2, \sigma) \Downarrow o}\ \text{P-Seq1} \qquad \frac{(c_2, \sigma) \Downarrow o}{(\mathsf{seq2}\ (\mathsf{conv}\sigma)\ c_2, \sigma_0) \Downarrow o}\ \text{P-Seq2}$$

Each of these two rules is a pretty-big-step rule: reading the rules in a bottom-up manner, P-Seq1 evaluates a single sub-term c_1, plugs the result of evaluation into the semantic constructor $\mathsf{seq2}$, and evaluates that term. The rule P-Seq2 in turn checks that the result of evaluating the first sub-term did not result in divergence, and goes on to evaluate c_2. An additional so-called *abort rule* is required for propagating divergence if it occurs in the first branch of a sequential composition:

$$\frac{}{(\mathsf{seq2}\ \mathsf{div}\ c_2, \sigma) \Downarrow \mathsf{div}}\ \text{P-Seq-Abort}$$

Such abort rules are required for all semantic constructors. As Charguéraud [8] remarks, these are tedious to both read and write, but are straightforward to generate automatically.

The pretty-big-step rules in Figure 5 have a *dual* interpretation: such rules define two separate relations, one *inductive*, the other *coinductive*. We use du on the left of rules to indicate relations with dual interpretations. We use $\Downarrow$ to refer to the inductive interpretation, and $\overset{\mathsf{co}}{\Downarrow}$ to refer to the coinductive interpretation. We refer to the union of the two interpretations of the relation defined by a set of pretty-big-step rules by $\overset{\mathsf{du}}{\Downarrow}$. Crucially, these relations are based on the same set of rules. In practice, this means that the coinductively defined relation subsumes the inductively defined relation, as shown by Lemma 9.

$$\boxed{(C,\sigma) \Downarrow o}$$

$$\mathsf{du}\,\frac{}{(\mathsf{skip},\sigma)\Downarrow \mathsf{conv}\ \sigma}\ \text{P-Skip} \qquad \mathsf{du}\,\frac{x \notin \mathrm{dom}(\sigma)}{(\mathsf{alloc}\ x,\sigma)\Downarrow (\mathsf{conv}\ \sigma[x \mapsto \mathsf{null}])}\ \text{P-Alloc}$$

$$\mathsf{du}\,\frac{(e,\sigma) \Rightarrow_{\mathrm{E}} v \qquad (\mathsf{assign2}\ x\ v,\sigma)\Downarrow o}{(x := e,\sigma)\Downarrow o}\ \text{P-Assign1} \qquad \mathsf{du}\,\frac{x \in \mathrm{dom}(\sigma)}{(\mathsf{assign2}\ x\ v,\sigma)\Downarrow \mathsf{conv}\ \sigma[x \mapsto v]}\ \text{P-Assign2}$$

$$\mathsf{du}\,\frac{(c_1,\sigma)\Downarrow o_1 \qquad (\mathsf{seq2}\ o_1\ c_2,\sigma)\Downarrow o}{(c_1; c_2,\sigma)\Downarrow o}\ \text{P-Seq1} \qquad \mathsf{du}\,\frac{(c,\sigma)\Downarrow o}{(\mathsf{seq2}\ (\mathsf{conv}\ \sigma)\ c,\sigma_0)\Downarrow o}\ \text{P-Seq2}$$

$$\mathsf{du}\,\frac{\begin{array}{c}(e,\sigma) \Rightarrow_{\mathrm{E}} v\\ (\mathsf{if2}\ v\ c_1\ c_2,\sigma)\Downarrow o\end{array}}{(\mathsf{if}\ e\ c_1\ c_2,\sigma)\Downarrow o}\ \text{P-If} \qquad \mathsf{du}\,\frac{v \neq 0 \qquad (c_1,\sigma)\Downarrow o_1}{(\mathsf{if2}\ v\ c_1\ c_2,\sigma)\Downarrow o_1}\ \text{P-If2} \qquad \mathsf{du}\,\frac{(c_2,\sigma)\Downarrow o_2}{(\mathsf{if2}\ 0\ c_1\ c_2,\sigma)\Downarrow o_2}\ \text{P-IfZ2}$$

$$\mathsf{du}\,\frac{(e,\sigma) \Rightarrow_{\mathrm{E}} v \qquad (\mathsf{while2}\ v\ e\ c,\sigma)\Downarrow o}{(\mathsf{while}\ e\ c,\sigma)\Downarrow o}\ \text{P-While} \qquad \mathsf{du}\,\frac{\begin{array}{c}v \neq 0\\ (c,\sigma)\Downarrow o \qquad (\mathsf{while3}\ o\ e\ c,\sigma)\Downarrow o'\end{array}}{(\mathsf{while2}\ v\ e\ c,\sigma)\Downarrow o'}\ \text{P-While2}$$

$$\mathsf{du}\,\frac{}{(\mathsf{while2}\ 0\ e\ c,\sigma)\Downarrow \mathsf{conv}\ \sigma}\ \text{P-WhileZ2} \qquad \mathsf{du}\,\frac{(\mathsf{while}\ e\ c,\sigma)\Downarrow o}{\mathsf{while3}\ (\mathsf{conv}\ \sigma)\ e\ c,\sigma_0)\Downarrow o}\ \text{P-While3}$$

$$\mathsf{du}\,\frac{}{(\mathsf{seq2}\ \mathsf{div}\ c_2,\sigma)\Downarrow \mathsf{div}}\ \text{P-Seq-Abort} \qquad \mathsf{du}\,\frac{}{(\mathsf{while3}\ \mathsf{div}\ e\ c,\sigma)\Downarrow \mathsf{div}}\ \text{P-While-Abort}$$

Figure 5: Pretty-big-step semantics for commands

Lemma 9. *If* $(C,\sigma) \Downarrow o$ *then* $(C,\sigma) \overset{\mathsf{co}}{\Downarrow} o$.

Proof. Straightforward proof by induction on $\Downarrow$. ☐

However, the coinductive interpretation is less useful for proving properties about converging programs, since converging and diverging programs cannot be syntactically distinguished in the coinductive interpretation. For example, we can prove that $\mathsf{while}\ 1\ \mathsf{skip}$ coevaluates to anything:

Example 10. *For any* o, $(\mathsf{while}\ 1\ \mathsf{skip},\cdot) \overset{\mathsf{co}}{\Downarrow} o$.

Proof. Straightforward proof by coinduction. ☐

An important property of the rules in Figure 5 is that divergence is only derivable under the coinductive interpretation.

Lemma 11. $(c,\sigma) \Downarrow o$ *implies* $o \neq \mathsf{div}$.

Proof. Straightforward proof by induction on $\Downarrow$. ☐

Pretty-big-step semantics can be used to reason about terminating programs on a par with traditional big-step relations, as shown by Theorem 12.

Theorem 12. $(c,\sigma) \Rightarrow_{\mathrm{B}} \sigma'$ *iff* $(c,\sigma) \Downarrow \mathsf{conv}\ \sigma'$.

Proof. Each direction is proven by straightforward induction on $\Rightarrow_{\mathrm{B}}$ and $\Downarrow$ respectively. The $\Downarrow$-to-$\Rightarrow_{\mathrm{B}}$ direction uses Lemma 11. ☐

Pretty-big-step semantics can be used to reason about diverging programs on a par with traditional big-step divergence predicates, as shown by Theorem 13.

Theorem 13. *$(c, \sigma) \overset{\infty}{\Rightarrow}$ iff $(c, \sigma) \overset{\mathsf{co}}{\Downarrow} \mathsf{div}$.*

Proof (classical). Each direction is proved by coinduction. The $\overset{\infty}{\Rightarrow}$-to-$\overset{\mathsf{co}}{\Downarrow}$ direction uses Lemma 12. The other direction uses Lemma 14 (which relies on classical arguments) and Lemma 15. □

Lemma 14. *If $(c, \sigma) \overset{\mathsf{co}}{\Downarrow} o$ and $\neg ((c, \sigma) \Downarrow o)$ then $(c, \sigma) \overset{\mathsf{co}}{\Downarrow} \mathsf{div}$*

Proof (classical). By coinduction, using Lemma 9 and the law of excluded middle for case analysis on $(c, \sigma) \Downarrow (o, \sigma')$. □

Lemma 15. *If $(c, \sigma) \Downarrow \mathsf{conv}\ \sigma'$ and $(c, \sigma) \overset{\mathsf{co}}{\Downarrow} \mathsf{conv}\ \sigma''$ then $\sigma' = \sigma''$.*

Proof. Straightforward proof by induction on $\Downarrow$. □

Our pretty-big-step semantics uses 18 rules (counting rules for $\Rightarrow_{\mathrm{E}}$) with 16 premises (counting judgments about both $\Rightarrow_{\mathrm{E}}$ and $\overset{\mathsf{du}}{\Downarrow}$), none of which are duplicates. In contrast, the union of the rules for inductive standard big-step rules in Section 3.1 and the divergence predicate in Section 3.2 uses 17 rules with 25 premises of which 6 are duplicates. Pretty-big-step semantics solves the duplication problem, albeit the solution comes at the cost of introducing 5 extra semantic constructors and breaking the rules in the inductive interpretation up into multiple rules, which in this case adds an extra rule compared to the original big-step rules with duplication.

4. State-Based Big-Step Semantics

In this section we present a novel lightweight approach to representing divergence. It is lightweight in the sense that we do not have to introduce new relations or rules for existing constructs; and it is generic in the sense that the extension can be applied to rules to make them express both diverging and converging programs in a way that is independent of underlying constructs and auxiliary entities. The approach can be used to extend standard inductively defined rules to allow them to express divergence on a par with standard divergence predicates and pretty-big-step rules.

4.1. The While-Language and its State-Based Big-Step Semantics

We show how augmenting the standard inductive big-step rules from Figures 1 and 3 with *status flags* allows us to express and reason about divergence on a par with traditional big-step divergence predicates. Status flags indicate either convergence ($\downarrow$) or divergence ($\uparrow$), ranged over by:

$$\mathit{Status} \ni \delta ::= \downarrow \mid \uparrow$$

Threading status flags through the conclusion and premises of the standard big-step rules in left-to-right order gives the rules in Figure 6. In all rules, the status flag is threaded through rules such that the conclusion source always starts in a $\downarrow$ state.

Like pretty-big-step semantics, the rules in Figure 6 have a *dual* interpretation: both inductive and coinductive. We use $\Rightarrow_{\mathrm{G}}$ to refer to the relation given by the inductive interpretation of the rules, and $\overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}}$ to refer to the relation given by the coinductive interpretation. When needed, we refer to the union of these interpretations by $\overset{\mathsf{du}}{\Rightarrow}_{\mathrm{G}}$.

Figure 6 threads status flags through the rules for the $\Rightarrow_{\mathrm{E}}$ relation as well as the standard big-step relation for commands, $\Rightarrow_{\mathrm{G}}$. But expressions cannot actually diverge. Our motivation for threading the flag

$$\boxed{(e, \sigma, \delta) \Rightarrow_{\text{GE}} v, \delta'}$$

$$\text{du}\,\frac{}{(v, \sigma, \downarrow) \Rightarrow_{\text{GE}} v, \downarrow}\,\text{GE-Val} \qquad \text{du}\,\frac{x \in \text{dom}(\sigma)}{(x, \sigma, \downarrow) \Rightarrow_{\text{GE}} \sigma(x), \downarrow}\,\text{GE-Var} \qquad \text{du}\,\frac{\begin{array}{c}(e_1, \sigma, \downarrow) \Rightarrow_{\text{GE}} n_1, \delta \\ (e_2, \sigma, \delta) \Rightarrow_{\text{GE}} n_2, \delta'\end{array}}{(e_1 \oplus e_2, \sigma, \downarrow) \Rightarrow_{\text{GE}} \oplus(n_1, n_2), \delta'}\,\text{GE-Bop}$$

$$\text{du}\,\frac{}{(e, \sigma, \uparrow) \Rightarrow_{\text{GE}} v, \uparrow}\,\text{GE-Div}$$

$$\boxed{(c, \sigma, \delta) \Rightarrow_{\text{G}} \sigma', \delta'}$$

$$\text{du}\,\frac{}{(\mathsf{skip}, \sigma, \downarrow) \Rightarrow_{\text{G}} \sigma, \downarrow}\,\text{G-Skip} \qquad \text{du}\,\frac{x \notin \text{dom}(\sigma)}{(\mathsf{alloc}\ x, \sigma, \downarrow) \Rightarrow_{\text{G}} \sigma[x \mapsto \mathsf{null}], \downarrow}\,\text{G-Alloc}$$

$$\text{du}\,\frac{x \in \text{dom}(\sigma) \qquad (e, \sigma, \downarrow) \Rightarrow_{\text{GE}} v, \delta}{(x := e, \sigma, \downarrow) \Rightarrow_{\text{G}} \sigma[x \mapsto v], \delta}\,\text{G-Assign} \qquad \text{du}\,\frac{(c_1, \sigma, \downarrow) \Rightarrow_{\text{G}} \sigma', \delta \qquad (c_2, \sigma', \delta) \Rightarrow_{\text{G}} \sigma'', \delta'}{(c_1; c_2, \sigma, \downarrow) \Rightarrow_{\text{G}} \sigma'', \delta'}\,\text{G-Seq}$$

$$\text{du}\,\frac{\begin{array}{c} v \neq 0 \\ (e, \sigma, \downarrow) \Rightarrow_{\text{GE}} v, \delta \qquad (c_1, \sigma, \delta) \Rightarrow_{\text{G}} \sigma', \delta'\end{array}}{(\mathsf{if}\ e\ c_1\ c_2, \sigma, \downarrow) \Rightarrow_{\text{G}} \sigma', \delta'}\,\text{G-If} \qquad \text{du}\,\frac{(e, \sigma, \downarrow) \Rightarrow_{\text{GE}} 0, \delta \qquad (c_2, \sigma, \delta) \Rightarrow_{\text{G}} \sigma', \delta'}{(\mathsf{if}\ e\ c_1\ c_2, \sigma, \downarrow) \Rightarrow_{\text{G}} \sigma', \delta'}\,\text{G-IfZ}$$

$$\text{du}\,\frac{\begin{array}{cc}(e, \sigma, \downarrow) \Rightarrow_{\text{GE}} v, \delta & v \neq 0 \\ (c, \sigma, \delta) \Rightarrow_{\text{G}} \sigma', \delta' & (\mathsf{while}\ e\ c, \sigma', \delta') \Rightarrow_{\text{G}} \sigma'', \delta''\end{array}}{(\mathsf{while}\ e\ c, \sigma, \downarrow) \Rightarrow_{\text{G}} \sigma'', \delta''}\,\text{G-While} \qquad \text{du}\,\frac{(e, \sigma, \downarrow) \Rightarrow_{\text{GE}} 0, \delta}{(\mathsf{while}\ e\ c, \sigma, \downarrow) \Rightarrow_{\text{G}} \sigma, \delta}\,\text{G-WhileZ}$$

$$\text{du}\,\frac{}{(c, \sigma, \uparrow) \Rightarrow_{\text{G}} \sigma', \uparrow}\,\text{G-Div}$$

Figure 6: State-based big-step semantics for commands and expressions with divergence

through anyway is that it anticipates future language extensions which may permit expressions to diverge, such as allowing user-defined functions to be called. It also allows us to correctly propagate divergence in rules where evaluation of expressions does not necessarily occur as the first premise in rules. We discuss this point in Section 5.2.

How do these rules allow reasoning about divergence? In the G-Seq rule in Figure 6, the first premise may diverge to produce $\uparrow$ in place of δ in the first premise. If this is the case, any subsequent computation is irrelevant. To inhibit subsequent computation we introduce *divergence rules* E-Div and G-Div, also in Figure 6. The divergence rules serve a similar purpose as abort rules in pretty-big-step semantics: it propagates divergence as it arises and inhibits further evaluation.

A technical curiosity of the divergence rule is that it allows evaluation to return an *arbitrary* store. In other words, states record and propagate irrelevant information. Although this is an unusual way of propagating semantic information in big-step semantics, the intuition behind them follows how big-step SOS traditionally propagates divergence and abrupt termination. Recall that big-step and pretty-big-step rules discard the structure of the current program term, and only record that divergence occurs. Witness, for example, the big-step and pretty-big-step rules for propagating exceptions from the introduction of this article:

$$\frac{(c_1, \sigma) \overset{\infty}{\Rightarrow}}{(c_1; c_2, \sigma) \Rightarrow \sigma''}\,\text{B-}\infty\text{-Seq1} \qquad \frac{\mathsf{abort}(o)}{(\mathsf{seq2}\ o\ c_2, \sigma) \Downarrow o}\,\text{P-Seq-Abort}$$

These rules "forget" the structure of whatever other effects a diverging computation produces. Indeed, it makes little difference that it does, since the coinductive interpretation is not deterministic, as Example 10 proves. Similarly, state-based big-step semantics retains only the important structure of the state, but ignores the irrelevant parts: for converging computations, all parts of the state are important, whereas for divergent or abruptly terminated states, only the fact that we are abruptly terminating is important.

A key property of generic divergence is that the conclusions of our rules always start in a state with the convergent flag $\downarrow$. It follows that, under an inductive interpretation, we cannot construct derivations that result in a divergent state '$\uparrow$'. Lemma 16 proves that we cannot use the inductively defined relation to prove divergence.

Lemma 16. *$(c, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} \sigma', \delta$ implies $\delta \neq \uparrow$.*

Proof. Straightforward proof by induction on $\Rightarrow_{\mathrm{G}}$. $\square$

Theorem 17 proves that adding status flags and the divergence rule does not change the inductive meaning of the standard inductive big-step relation.

Theorem 17. *$(c, \sigma) \Rightarrow_{\mathrm{B}} \sigma'$ iff $(c, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} \sigma', \downarrow$.*

Proof. The $\Rightarrow_{\mathrm{B}}$-to-$\Rightarrow_{\mathrm{G}}$ follows by straightforward induction. The $\Rightarrow_{\mathrm{G}}$-to-$\Rightarrow_{\mathrm{B}}$ direction follows by straightforward induction and Lemma 16. $\square$

As for the coinductive interpretation of '$\Rightarrow_{\mathrm{G}}$', Theorem 18 proves that adding status flags allows us to prove divergence on a par with traditional big-step divergence predicates.

Theorem 18. *$(c, \sigma) \overset{\infty}{\Rightarrow}$ iff $(c, \sigma, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}} \sigma', \uparrow$.*

Proof (classical). The $\overset{\infty}{\Rightarrow}$-to-$\overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}}$ direction follows by straightforward coinduction, using Lemma 19, Lemma 20, and the law of excluded middle for case analysis on $\Rightarrow_{\mathrm{G}}$. $\square$

Lemma 19. *If $(c, \sigma, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}} \sigma', \delta$ and $\neg\,((c, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} \sigma, \delta)$ then $(c, \sigma, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}} \sigma', \uparrow$*

Proof (classical). By coinduction and the law of excluded middle for case analysis on $\Rightarrow_{\mathrm{G}}$. $\square$

The relationship between the inductive and coinductive interpretation of the rules for '$\Rightarrow_{\mathrm{G}}$' is similar to the relationships observed in connection with pretty-big-step semantics. The coinductive interpretation also subsumes the inductive interpretation, as proven in Lemma 20.

Lemma 20. *If $(c, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} \sigma', \downarrow$ then $(c, \sigma, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}} \sigma', \downarrow$*

Proof. Straightforward proof by induction on $\Rightarrow_{\mathrm{G}}$. $\square$

As with pretty-big-step semantics (Section 3.3), the coinductive interpretation is less useful for proving properties about converging programs, since converging and diverging programs cannot be distinguished in the coinductive interpretation. For example, we can prove that while 1 skip coevaluates to anything:

Example 21. *For any σ' and δ, $(\mathsf{while}\ 1\ \mathsf{skip}, \cdot, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}} \sigma', \delta$.*

Proof. Straightforward proof by coinduction. $\square$

Comparing generic divergence (Figure 6) with pretty-big-step (Figure 5), we see that our rules contain no duplication, and use just 13 rules with 13 premises, whereas pretty-big-step semantics uses 18 rules with 16 premises. While we use fewer rules than pretty-big-step, and introducing divergence does not introduce duplicate premises, the state-based big-step rules in Figure 6 actually do contain some duplicate premises in the rules G-If, G-IfZ, G-While, and G-WhileZ: each of these rules evaluate the same expression e. This duplication could have been avoided by introducing an intermediate for for if and while, similar to pretty-big-step.

State-based big-step SOS is equivalent to but more concise than traditional big-step and pretty-big-step. For the simple While-language, rules with state-based divergence is no more involved to work with than with traditional approaches, as illustrated by examples in our Coq proofs available online at: `http://www.plancomps.org/jlamp2015/`.

4.2. Divergence Rules are Necessary

From Example 21 we know that some diverging programs result in a $\downarrow$ state. But do we really need the $\uparrow$ flag and divergence rule? Here, we answer this question affirmatively. Consider the following program:

$$(\mathsf{while}\ 1\ \mathsf{skip}); \mathsf{alloc}\ \mathrm{x}; \mathrm{x} := +(\mathrm{x}, 0)$$

Proving that this program diverges in this case depends crucially on the G-Div allowing us to propagate divergence. The while-command diverges whereas the last sub-commands of the program contain a stuck computation: the variable x has the null value when it is dereferenced. The derivation trees we can construct must use the G-Div rule as follows:

$$\dfrac{(\mathsf{while}\ 1\ \mathsf{skip}, \cdot, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}} (\cdot, \uparrow) \qquad \dfrac{}{(\mathsf{alloc}\ \mathrm{x}; \mathrm{x} := +(\mathrm{x}, 0), \cdot, \uparrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}} (\cdot, \uparrow)}\ \text{G-Div}}{((\mathsf{while}\ 1\ \mathsf{skip}); \mathsf{alloc}\ \mathrm{x}; \mathrm{x} := +(\mathrm{x}, 0), \cdot, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}} (\cdot, \uparrow)}\ \text{G-Seq}$$

Example 22. *For any* σ*,* $((\mathsf{while}\ 1\ \mathsf{skip}); \mathsf{alloc}\ \mathrm{x}; \mathrm{x} := +(\mathrm{x}, 0), \cdot, \downarrow) \overset{\mathsf{co}}{\Rightarrow} (\sigma, \uparrow)$*. In contrast,* $\neg\ \exists\sigma.\ ((\mathsf{while}\ 1\ \mathsf{skip}); \mathsf{alloc}\ \mathrm{x}; \mathrm{x} := +(\mathrm{x}, 0), \cdot, \downarrow) \overset{\mathsf{co}}{\Rightarrow} (\sigma, \downarrow)$

Leroy and Grall [3] observed a similar point about the coinductive interpretation of the rules for the λ-calculus with contants, i.e., that it is non-deterministic, and that it does not contain computations that diverge and then get stuck, nor computations that diverge towards infinite values. Here, we have shown (Theorem 17 and 18) that coevaluation of state-based big-step semantics is equally expressive as traditional divergence predicates and, transitively (by Theorem 2 and 5), small-step semantics.

5. Beyond the While-Language

State-based big-step semantics is able to reason about divergence on a par with small-step semantics for the simple While-language. In this section we illustrate that the approach also scales to deal with other language features, such as exceptions and non-deterministic input. To this end, we present a novel proof method for relating small-step and big-step semantics with sources of non-determinism. Finally, we discuss potential pitfalls and limitations.

5.1. Non-deterministic input

Previous approaches to relating big-step and small-step SOS focus mainly on relating finite computations [4; 18; 19]. The main counter-example appears to be Leroy and Grall's study of coinductive big-step semantics [3], which considers a deterministic language. Their proof for relating diverging computations in small-step and big-step semantics relies crucially on this fact. We consider the extension of our language with an expression form for non-deterministic interactive input, and prove that the extension preserves the equivalence of small-step semantics and state-based big-step semantics. Our proof provides a novel means of relating small-step and big-step SOS for divergent computations involving non-determinism at the leaves of derivation trees.

Consider the extension of our language with the following expression form which models interactive input:

$$\mathit{Expr} \ni e ::= \ldots \mid \mathsf{input}$$

Its single expression evaluation rule is:

$$\dfrac{}{(\mathsf{input}, \sigma, \downarrow) \Rightarrow_{\mathrm{GE}} v, \downarrow}$$

Adding this construct clearly makes our language non-deterministic, since the value to the right of $\Rightarrow_{\mathrm{GE}}$ is free. If we extend expression evaluation for small-step correspondingly, is the big-step state-based semantics still equivalent to the small-step semantics?

If we extend expression evaluation for the small-step transition relation correspondingly, the answer should intuitively be “yes”: we have not changed the semantics of commands, so most of the proofs of the properties about the relationship between small-step and big-step semantics from Section 2 carry over unchanged, since they rely on the expression evaluation relation being equivalent between the two semantics.

But the proof Theorem 5 crucially relies on Lemma 7, which in turn holds only for deterministic transition relations and expression evaluation relations. The violated property is the following:

If $(c_1, \sigma) \rightarrow^* (c_1', \sigma')$ and $(c_1; c_2, \sigma) \overset{\infty}{\rightarrow}$, then $(c_1'; c_2, \sigma') \overset{\infty}{\rightarrow}$.

Here, if $\rightarrow$ is non-deterministic, it may be that c_1 makes a sequence of transition steps to a c_1' which is stuck, whereas there may exist a derivation such that $c_1; c_2$ diverges.

Example 23. *Consider the program state:*

$$(\overbrace{\mathsf{if\ input}\ \mathrm{x}\ \mathsf{skip}}^{c_1}; \overbrace{\mathsf{while}\ 1\ \mathsf{skip}}^{c_2}, \cdot)$$

If input *returns a non-zero value, evaluation gets stuck. Otherwise, evaluation diverges. Thus it holds that* $(c_1, \cdot,) \rightarrow^* (\mathrm{x}, \cdot)$ *and* $(c_1; c_2, \cdot) \overset{\infty}{\rightarrow}$*, but not* $(\mathrm{x}; c_2, \cdot) \overset{\infty}{\rightarrow}$.

What we *can* prove about possibly-terminating computations that rely on sequential composition instead is the following:

Lemma 24. *If* $(c_1; c_2, \sigma) \overset{\infty}{\rightarrow}$ *and there is no* σ' *such that* $(c_1, \sigma) \rightarrow^* (\mathsf{skip}, \sigma')$*, then it must be the case that* $(c_1, \sigma) \overset{\infty}{\rightarrow}$.

Proof. The proof is by coinduction, using the goal as coinduction hypothesis. The proof follows by inversion on the first hypothesis, from which we derive that either $c_1 = \mathsf{skip}$, which leads to a contradiction, or there exist c_1', σ_1 for which $(c_1, \sigma) \rightarrow (c_1', \sigma_1)$. By the second hypothesis, it must also hold that there is no σ' such that $(c_1', \sigma_1) \rightarrow^* (\mathsf{skip}, \sigma')$. From these facts, the goal follows by applying Trans∞, the small-step rule S-Seq, and the coinduction hypothesis. □

Using Lemma 24, we can relate infinite sequences of small-step transitions to infinite state-based big-step derivations as follows.

Lemma 25. *If* $(c, \sigma) \overset{\infty}{\rightarrow}$ *then* $(c, \sigma, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}} \sigma', \uparrow$ *for any* σ'.

Proof (classical). The proof is by guarded coinduction, using the goal as the coinduction hypothesis. The critical cases are those for sequential composition and ‘while’: these cases use the law of excluded middle for case analysis on whether c_1 converges or not. If it does, the goal follows by Lemma 20 (which carries over unchanged). If it does not, the goal follows by Lemma 24 above. □

The proof of the other direction, i.e., relating infinite big-step derivations to infinite sequences of small-step transitions can now be proven.

Theorem 26. $(c, \sigma) \overset{\infty}{\rightarrow}$ *iff for any* σ', $(c, \sigma, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}} \sigma', \uparrow$.

Proof (classical). The small-to-big direction is proven in Lemma 24 above. The other direction follows by coinduction and Lemma 27 below. □

Lemma 27. *If, for any* σ'' $(c, \sigma) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}} \sigma'', \uparrow$*, then* $\exists c', \sigma'.\ \Big((c, \sigma) \rightarrow (c', \sigma') \ \wedge \ (c', \sigma') \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}} \sigma'', \uparrow\Big)$.

Proof. The proof carries over from Lemma 8, and relies on Theorem 28 below. □

Theorem 28. $(c, \sigma) \rightarrow^* (\mathsf{skip}, \sigma')$ *iff* $(c, \sigma) \Rightarrow_{\mathrm{B}} \sigma'$.

Proof. The proof straightforwardly carries over from Theorem 2. □

5.2. Exceptions

We extend our language with exceptions. We add exceptions to our language by augmenting the syntactic sort for status flags:

$$\mathit{Status} \ni \delta ::= \ldots \mid \mathsf{exc}(v)$$

We also augment the syntactic sort for commands with a '$\mathsf{throw}\ v$' construct for throwing a value v as an exception:

$$\mathit{Cmd} \ni c ::= \mathsf{throw}\ v$$

The rules for the construct are given by a single rule for throwing exception, and two rules for propagating the exception for both command and expression evaluation:

$$\frac{}{(\mathsf{throw}\ v, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} \sigma', \mathsf{exc}(v)}\ \text{G-Throw} \qquad \frac{}{(c, \sigma, \mathsf{exc}(v)) \Rightarrow_{\mathrm{G}} \sigma', \mathsf{exc}(v)}\ \text{G-Exc}$$

$$\frac{}{(e, \sigma, \mathsf{exc}(v)) \Rightarrow_{\mathrm{GE}} v', \mathsf{exc}(v)}\ \text{GE-Exc}$$

Using these rules, exceptions are propagated similarly to how divergence is propagated in rules.

The G-Throw and G-Exc rules above admit arbitrary σ's on the right-hand side of the arrow. Admitting arbitrary stores to be propagated in connection with divergence was motivated in part by the fact that divergent computations are non-deterministic anyway (see, e.g., Example 21). But computations that throw exceptions are guaranteed to converge, so do we really need it here?

For the simple language considered here, we do not strictly need to relate exceptional states to arbitrary other states. We choose to do so in order to match the traditional big-step rules for exceptions recalled in the introduction:

$$\frac{(c_1, \sigma) \Rightarrow \mathsf{exc}(v)}{(c_1; c_2, \sigma) \Rightarrow \mathsf{exc}(v)}\ \text{B-Exc-Seq1} \qquad \frac{(c_1, \sigma) \Rightarrow \sigma' \quad (c_2, \sigma') \Rightarrow \mathsf{exc}(v)}{(c_1; c_2, \sigma) \Rightarrow \mathsf{exc}(v)}\ \text{B-Exc-Seq2}$$

In these rules, as in the state-based G-Seq rule, the stores resulting from abruptly terminated states are irrelevant.

One might like to remember the store in rules when exceptions are thrown. There are at least two ways to do this:

1. we can make the state-based exception rules remember the state of the store, i.e., we can replace the abrupt termination rules G-Throw, G-Exc, and GE-Exc by:

$$\frac{}{(\mathsf{throw}\ v, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} \sigma, \mathsf{exc}(v)}\ \text{G-Throw} \qquad \frac{}{(c, \sigma, \mathsf{exc}(v)) \Rightarrow_{\mathrm{G}} \sigma, \mathsf{exc}(v)}\ \text{G-Exc}$$

$$\frac{}{(e, \sigma, \mathsf{exc}(v)) \Rightarrow_{\mathrm{GE}} v', \mathsf{exc}(v)}\ \text{GE-Exc}$$

   or

2. we can make the exception status itself record the store, i.e.:

$$\mathit{Status} \ni \delta ::= \ldots \mid \mathsf{exc}(v, \sigma)$$

While the first approach seems intuitively simplest, the second may is more flexible in some settings: consider a variant of the While-language where exceptions can arise during expression evaluation, where expression evaluation may affect the store, and where we allow variables to be passed around as first-class objects similar to references in Standard ML [20]. The signature of expression evaluation and the assignment rule in such a language would look like:

$$\boxed{(e, \sigma, \delta) \Rightarrow_{\mathrm{GE}} v, \sigma, \delta'}$$

$$\frac{(e_1, \sigma, \downarrow) \Rightarrow_{\mathrm{GE}} x, \sigma', \delta \quad (e_2, \sigma', \delta) \Rightarrow_{\mathrm{GE}} v, \sigma'', \delta' \quad x \in \mathrm{dom}(\sigma'')}{(e_1 := e_2, \sigma, \downarrow) \Rightarrow_{\mathrm{GE}} v, \sigma''[x \mapsto v], \delta'}$$

Given the rule and language described above, if e_1 or e_2 abruptly terminates the rule still insists that $x \in \sigma$. Thus, it becomes essential that abrupt termination allows us to return an arbitrary store, such that we can synthesise a store σ'' for which $x \in \text{dom}(\sigma'')$. If we didn't evaluatoin might get stuck instead of propagating abrupt termination.

The need to synthesise semantic information that is not the result of any actual computation is a definite drawback of the approach. We emphasise that:

- in spite of this, state-based big-step rules do allow abrupt termination and divergence to be propagated correctly; and
- the state-based approach is equally applicable to the more flexible pretty-big-step style, which would allow tedious abort-rules to be replaced by a single state-based abort rule instead.[4]

6. Implicit Generic Divergence

Previous section presented a novel encoding of divergence. In this section, we first recall how I-MSOS [10] provides a framework for semantics-preserving language extension. Next, we present an extension of I-MSOS that allows us to automatically transform traditional big-step SOS signatures and rules to express divergence on a par with small-step semantics.

Recall the inductive big-step rule for sequential composition:

$$\frac{(c_1, \sigma) \Rightarrow_{\text{B}} \sigma' \qquad (c_2, \sigma') \Rightarrow_{\text{B}} \sigma''}{(c_1; c_2, \sigma) \Rightarrow_{\text{B}} \sigma''}\,\text{B-Seq}$$

This rule threads a store from conclusion source through the premises to the conclusion target. Adding a new auxiliary entity, such as a status flag, requires us to thread the entity through in essentially the same way. I-MSOS allows us to omit auxiliary entities in rules where the auxiliary entity is merely propagated, like the B-Seq rule. Consider the following I-MSOS signature and rule:

$$\boxed{(\ c\ , \sigma) \Rightarrow_{\text{I}} \sigma'}$$

$$\mathsf{du}\,\frac{c_1 \Rightarrow_{\text{I}} () \qquad c_2 \Rightarrow_{\text{I}} ()}{c_1; c_2 \Rightarrow_{\text{I}} ()}\,\text{I-Seq}$$

The auxiliary entity that is highlighted in the signature says that the entity is implicitly and automatically propagated in rules that do not explicitly mention it. In rules with multiple premises, I-MSOS supports multiple ways of threading entities through the premises. The simplest is to thread them through premises in left-to-right or right-to-left order. Here, we let I-MSOS rules thread auxiliary entities through premises in left-to-right order. Thus, the rule I-Seq corresponds exactly to B-Seq.

Using I-MSOS, rules can be specified independently and combined without having the meaning of the rules change. Specifically, for any derivation tree that can be constructed *before* introducing a new auxiliary entity by means of I-MSOS, a derivation tree with the same structure can be constructed *after*, too.

The I-MSOS rules and signatures in Figure 7 correspond to the inductive rules in Figure 3. In order to express state-based divergence, we need to extend I-MSOS. Our extensions are as follows:

- *Left-hand side default values:* a non-variable occurring in a highlighted position on the left-hand side of an arrow in an I-MSOS signature denotes a *default value.* Unless another value is explicitly given in the rule, the conclusion left-hand side auxiliary entity matches that value. The auxiliary entity is threaded through rules in the usual fashion [10, Section 2.3].

[4]This approach was previous explored by the authors in [21, Section 3.1].

$$\boxed{(\; e \;, \sigma) \Rightarrow_{\mathrm{IE}} v}$$

$$\frac{}{v \Rightarrow_{\mathrm{IE}} v}\,\text{IE-Val} \qquad \frac{x \in \mathrm{dom}(\sigma)}{(x, \sigma) \Rightarrow_{\mathrm{IE}} \sigma(x)}\,\text{IE-Var} \qquad \frac{e_1 \Rightarrow_{\mathrm{IE}} n_1 \quad e_2 \Rightarrow_{\mathrm{IE}} n_2}{e_1 \oplus e_2 \Rightarrow_{\mathrm{IE}} \oplus(n_1, n_2)}\,\text{IE-Bop}$$

$$\boxed{(\; c \;, \sigma) \Rightarrow_{\mathrm{I}} \sigma'}$$

$$\mathsf{du}\frac{}{\mathsf{skip} \Rightarrow_{\mathrm{I}} ()}\,\text{I-Skip} \qquad \mathsf{du}\frac{x \notin \mathrm{dom}(\sigma)}{(\mathsf{alloc}\ x, \sigma) \Rightarrow_{\mathrm{I}} \sigma[x \mapsto \mathsf{null}]}\,\text{I-Alloc}$$

$$\mathsf{du}\frac{x \in \mathrm{dom}(\sigma) \quad (e, \sigma) \Rightarrow_{\mathrm{IE}} v}{(x := e, \sigma) \Rightarrow_{\mathrm{I}} \sigma[x \mapsto v]}\,\text{I-Assign} \qquad \mathsf{du}\frac{c_1 \Rightarrow_{\mathrm{I}} () \quad c_2 \Rightarrow_{\mathrm{I}} ()}{c_1; c_2 \Rightarrow_{\mathrm{I}} ()}\,\text{I-Seq}$$

$$\mathsf{du}\frac{e \Rightarrow_{\mathrm{IE}} v \quad v \neq 0 \quad c_1 \Rightarrow_{\mathrm{I}} ()}{\mathsf{if}\ e\ c_1\ c_2 \Rightarrow_{\mathrm{I}} ()}\,\text{I-If} \qquad \mathsf{du}\frac{e \Rightarrow_{\mathrm{IE}} 0 \quad c_2 \Rightarrow_{\mathrm{I}} ()}{\mathsf{if}\ e\ c_1\ c_2 \Rightarrow_{\mathrm{I}} ()}\,\text{I-IfZ}$$

$$\mathsf{du}\frac{e \Rightarrow_{\mathrm{IE}} v \quad v \neq 0 \quad c \Rightarrow_{\mathrm{I}} () \quad \mathsf{while}\ e\ c \Rightarrow_{\mathrm{I}} ()}{\mathsf{while}\ e\ c \Rightarrow_{\mathrm{I}} ()}\,\text{I-While} \qquad \mathsf{du}\frac{e \Rightarrow_{\mathrm{IE}} 0}{\mathsf{while}\ e\ c \Rightarrow_{\mathrm{I}} ()}\,\text{I-WhileZ}$$

Figure 7: Big-step I-MSOS for expressions and commands with implicit propagation

For example, in the signature judgment $(\; c \;, \downarrow) \Rightarrow \delta$, $\downarrow$ is a non-variable, whereas δ is a variable. Applying this signature judgment as described above to the I-Seq I-MSOS rule from Figure 7 gives the following SOS rule:

$$\frac{(c_1, \downarrow) \Rightarrow_{\mathrm{I}} \delta \quad (c_2, \delta) \Rightarrow_{\mathrm{I}} \delta'}{(c_1; c_2, \downarrow) \Rightarrow_{\mathrm{I}} \delta'}$$

Considering another example, applying the same judgment signature to the I-Skip axiom from Figure 7 gives the SOS axiom:

$$\frac{}{(\mathsf{skip}, \downarrow) \Rightarrow \downarrow}$$

Here, we follow I-MSOS in that, unless an auxiliary entity is explicitly mentioned and changed in an I-MSOS axiom, the axiom does not exhibit observable side-effects; hence the occurrence of the $\downarrow$ flag in both source and target in the axiom above.

- *Mode of interpretation:* the default interpretation for rules is given by the annotation on the relation symbol. A relation symbol '$\rightsquigarrow$' that has no annotation means that rules, unless otherwise indicated, have an inductive interpretation; '$\overset{\mathsf{co}}{\rightsquigarrow}$' means that the default interpretation is coinductive; and '$\overset{\mathsf{du}}{\rightsquigarrow}$' means that the default interpretation is dual.
- *Naming:* we use highlighted subscripts to name relations. For example, $\rightsquigarrow_{\mathrm{A}\ \mathrm{B}}$ extends the rules for $\rightsquigarrow_{\mathrm{A}}$, and calls the extended relation $\rightsquigarrow_{\mathrm{AB}}$.

Using these extensions, we give the following signature for the rules in Figure 7:

$$\boxed{(\; c \;, \downarrow) \overset{\mathsf{du}}{\Rightarrow}_{\mathrm{I}\ \mathrm{G}} \delta}$$

We also augment the rules in Figure 7 by the divergence rule from Section 4.1:

$$\mathsf{du}\frac{}{(c, \sigma, \uparrow) \Rightarrow_{\mathrm{IG}} (\sigma', \uparrow)}\,\text{IG-Div}$$

The resulting set of rules are identical to those in Figure 6. Using I-MSOS, we have automatically generated rules that solve the duplication problem and allow us to reason about diverging programs on a par with traditional approaches from the literature.

7. Related Work

Several papers have explored how to represent divergence in big-step semantics. Leroy and Grall [3] survey different approaches to representing divergence in coinductive big-step semantics, including divergence predicates, trace-based semantics, and taking the coinductive interpretation of standard big-step rules. They conclude that traditional divergence predicates are the most well-behaved, but increase the size of specifications by around 40%. The trace-based semantics of Leroy and Grall relies on concatenating infinite traces for accumulating the full trace of rules with multiple premises. Nakata and Uustalu [22] propose a more elegant approach to accumulating traces based, on 'peel' rules. Their approach is closely related to the *partiality monad*, introduced by Capretta, and used by several authors to give functional representations of big-step semantics, including Danielsson [23] and Abel and Chapman [24] (who call it the *delay monad*). In the partiality monad, functions either return a finitely delayed result or an infinite trace of delays. Piróg and Gibbons [25] study the category theoretic foundations of the resumption monad. Related to the resumption monad is the interactive I/O monad by Hancock and Setzer [26] for modeling the behaviour of possibly-diverging interactive programs.

While it is possible to specify and reason about operational semantics by means of the partiality monad, it relies on the ability to express mixed inductive/coinductive functions in order to use it in proof assistants. Agda [27] provides native support for such function definitions, while Coq does not. Nakata and Uustalu [22; 28] show that it is possible to express and reason about a functional representation of a trace-based semantics in Coq using a purely coinductive style. This works well for the simple While-language, but preliminary experiments suggest that such guarded coinductive functions may be more subtle to implement in Coq. Trace-based semantics work provides a strong foundation for reasoning about possibly-diverging programs, but unfortunately, a modern proof assistant like Coq is not up to the task of expressing and reasoning about these in a fully satisfactory manner: Coq's support for coinductive proofs is limited to guarded coinduction, which makes many otherwise simple proofs unnecessarily involved. For example, proving a goal of the following form is not possible by guarded coinduction alone:

$$P \implies \exists \tau (c, \sigma) \overset{\text{co}}{\Rightarrow} \tau$$

Here, P is some proposition, τ is a trace, and $\implies$ is implication. Proving such impliciations can be done by manually constructing a witness function for τ which is not always trivial. Charguéraud [8] directs a similar criticism at Coq's lack of support for coinduction, and cites this as one of the motivations for the pretty-big-step style.

Nakata and Uustalu also give coinductive big-step semantics for *resumptions* [28] (denoting the external behaviour of a communicating agent in concurrency theory [29]) as well as interleaving and concurrency [30]. These lines of work provide a more general way of giving and reasoning about all possible outcomes of a program than the committed-choice non-determinism that we considered in Section 5.1. It is, however, also somewhat more involved to specify and work with in a proof assistant like Coq.

Moggi [31] suggested monads as a means to modularity in denotational semantics [32]. In a similar vein, Modular SOS [9] provides a means to modularity in operational semantics. The state-based approach to divergence presented here is a variant of the abrupt termination technique used in [9]. That article represents abrupt termination as emitted signals. In a small-step semantics, this enables the encoding of abrupt termination by introducing a top-level handler that matches on emitted signals: if the handler observes an emitted signal, the program abruptly terminates. Exceptions as emitted signals could also be used for expressing abrupt termination in big-step semantics. However, this would entail wrapping each premise in a handler, thereby cluttering rules. Subsequent work [21] observed that encoding abrupt termination as a stateful flag instead scales better to big-step rules by avoiding such explicit handlers. State-based divergence was used in [33; 34] for giving a semantics for the untyped λ-calculus.

This work differs from [21; 33; 34] in several ways: unlike [21] it considers both inductive and coinductive big-step semantics and it is based on SOS, but shows how to extend the approach to I-MSOS. Whereas [34] provided a case study based on extending Leroy and Grall's proofs about a substitution-based λ-calculus, this work considers the While-language, and provides more in-depth analyses and techniques for using the approach to give and reason about programming language semantics in practice. [33] proposed the state-based approach as a straightforward way of augmenting a semantics such that it is useful for reasoning about divergence, but did not provide proofs of the equivalence with traditional big-step semantics nor small-step semantics. Lastly, [21] mainly considers pretty-big-step, whereas the state-based big-step semantics considered here have a more traditional big-step flavour (although, as remarked in Section 5.2, this entails some potential pitfalls that could be avoided by using pretty-big-step instead).

The question of which semantic style (small-step or big-step) is better for proofs is a moot point. Big-step semantics are held to be more convenient for certain proofs, such as compiler correctness proofs. Leroy and Grall [3] cite compiler correctness proofs as a main motivation for using coinductive big-step semantics as opposed to small-step semantics: using small-step semantics complicates the correctness proof. Indeed, Hutton and Wright [5] and Hutton and Bahr [35] also use big-step semantics for their compiler correctness proofs. However, in the certified C compiler *CompCert*, Leroy [36] uses a small-step semantics and sophisticated notions of bisimulation for its compiler correctness proofs.

In their paper [37] introducing the syntactic approach to type safety, Wright and Felleisen survey type safety proofs based on denotational and big-step semantics, and conclude that small-step semantics are a better fit for proving type safety by progress and preservation lemmas. Harper and Stone [38] direct a similar criticism at the big-step style. Big-step semantics can, however, be used for strong type safety on a par with small-step semantics. Leroy and Grall [3] show how to coinductively prove progress using coinductive divergence predicates, whereby type safety can be proven, provided one also proves a big-step preservation lemma, which is usually unproblematic. Another approach to big-step type safety consists in making the big-step semantics *total* by providing explicit error rules for cases where the semantics goes wrong. Type safety is then proven by showing that well-typed programs cannot go wrong. A non-exhaustive list of examples that use explicit error rules includes: Cousot's work on types as abstract interpretations [39]; Danielsson's work on operational semantics using the partiality monad [23]; and Charguéraud's work on pretty-big-step semantics [8], which provides a nice technique for encoding explicit error rules more conveniently. A third option for proving type safety using a big-step relation is to encode the big-step semantics as a small-step abstract machine [40; 41; 42], whereby the standard small-step type safety proof technique applies. A preliminary study [33] of a variant of Cousot's types as abstract interpretations suggests that abstract interpretation can be used to prove big-step type safety without the explicit error rules Cousot uses in his original presentation [39].

8. Conclusion

We presented a novel approach to augmenting standard big-step semantics to express divergence on a par with small-step and traditional big-step semantics. Our approach to representing divergence uses fewer rules than existing approaches of similar expressiveness (e.g., traditional big-step and pretty-big-step), and can be generated automatically by extending the I-MSOS framework. We also considered how to extend our semantics with interactive input, and provided a slight generalisation of the traditional proof method for relating diverging small-step and big-step semantics.

Our experiments show that state-based semantics provides a novel, lightweight, and promising approach to concise big-step SOS specification of programming languages involving divergence and abrupt termination.

Acknowledgements. Thanks to the anonymous reviewers, Neil Sculthorpe, Paolo Torrini, and Ulrich Berger for their helpful suggestions for improving this article. This work was supported by an EPSRC grant (EP/I032495/1) to Swansea University in connection with the *PLanCompS* project (`www.plancomps.org`).

References

[1] G. D. Plotkin, A structural approach to operational semantics, J. Log. Algebr. Program. 60-61 (2004) 17–139. doi:10.1016/j.jlap.2004.05.001.

[2] G. Kahn, Natural semantics, in: STACS'87, Vol. 247 of LNCS, Springer, 1987, pp. 22–39. doi:10.1007/BFb0039592.

[3] X. Leroy, H. Grall, Coinductive big-step operational semantics, Information and Computation 207 (2) (2009) 284–304. doi:10.1016/j.ic.2007.12.004.

[4] T. Nipkow, G. Klein, Concrete Semantics: With Isabelle/HOL, Springer, 2014.

[5] G. Hutton, J. Wright, What is the meaning of these constant interruptions?, JFP 17 (2007) 777–792. doi:10.1017/S0956796807006363.

[6] O. Danvy, L. R. Nielsen, Refocusing in reduction semantics, BRICS Research Series RS-04-26, Dept. of Comp. Sci., Aarhus University (2004).

[7] C. Bach Poulsen, P. D. Mosses, Generating specialized interpreters for Modular Structural Operational Semantics, in: LOPSTR'13, Vol. 8901 of LNCS, Springer, 2014.

[8] A. Charguéraud, Pretty-big-step semantics, in: M. Felleisen, P. Gardner (Eds.), ESOP'14, Vol. 7792 of LNCS, Springer, 2013, pp. 41–60. doi:10.1007/978-3-642-37036-6_3.

[9] P. D. Mosses, Modular structural operational semantics, J. Log. Algebr. Program. 60-61 (2004) 195–228. doi:10.1016/j.jlap.2004.03.008.

[10] P. D. Mosses, M. J. New, Implicit propagation in structural operational semantics, ENTCS 229 (4) (2009) 49–66. doi:10.1016/j.entcs.2009.07.073.

[11] J. C. Reynolds, Definitional interpreters for higher-order programming languages, in: ACM Annual Conference, ACM, 1972, pp. 717–740. doi:10.1145/800194.805852.

[12] P. Cousot, R. Cousot, Inductive definitions, semantics and abstract interpretations, in: POPL'92, ACM, 1992, pp. 83–94. doi:10.1145/143165.143184.

[13] B. C. Pierce, Types and programming languages, MIT Press, 2002.

[14] D. Sangiorgi, Introduction to Bisimulation and Coinduction, Cambridge University Press, 2011.

[15] T. Coquand, G. Huet, The calculus of constructions, Information and Computation 76 (23) (1988) 95–120. doi:10.1016/0890-5401(88)90005-3.

[16] B. C. Pierce, Advanced Topics in Types and Programming Languages, The MIT Press, 2004.

[17] J. P. Seldin, On the proof theory of Coquand's calculus of constructions, Annals of Pure and Applied Logic 83 (1) (1997) 23–101. doi:10.1016/S0168-0072(96)00008-5.

[18] B. C. Pierce, C. Casinghino, M. Greenberg, C. Hriţcu, V. Sjöberg, B. Yorgey, Software Foundations, 2013. URL http://cis.upenn.edu/~bcpierce/sf

[19] Ş. Ciobâcă, From small-step semantics to big-step semantics, automatically, in: E. B. Johnsen, L. Petre (Eds.), IFM'13, Vol. 7940 of LNCS, Springer, 2013, pp. 347–361. doi:10.1007/978-3-642-38613-8_24.

[20] R. Milner, M. Tofte, D. MacQueen, The Definition of Standard ML, MIT Press, 1997.

[21] C. Bach Poulsen, P. D. Mosses, Deriving pretty-big-step semantics from small-step semantics, in: ESOP'14, Vol. 8410 of LNCS, Springer, 2014, pp. 270–289. doi:10.1007/978-3-642-54833-8_15.

[22] K. Nakata, T. Uustalu, Trace-based coinductive operational semantics for while, in: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (Eds.), TPHOLs'09, Vol. 5674 of LNCS, Springer, 2009, pp. 375–390. doi:10.1007/978-3-642-03359-9_26.

[23] N. A. Danielsson, Operational semantics using the partiality monad, in: ICFP'12, ACM, 2012, pp. 127–138. doi:10.1145/2364527.2364546.

[24] A. Abel, J. Chapman, Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types, in: P. Levy, N. Krishnaswami (Eds.), MSFP'14, Vol. 153 of ENTCS, Open Publishing Association, 2014, pp. 51–67. doi:10.4204/EPTCS.153.4.

[25] M. Piróg, J. Gibbons, The coinductive resumption monad, ENTCS 308 (2014) 273–288. doi:10.1016/j.entcs.2014.10.015.

[26] P. Hancock, A. Setzer, Interactive programs in dependent type theory, in: P. Clote, H. Schwichtenberg (Eds.), CSL'00, Vol. 1862 of LNCS, Springer, 2000, pp. 317–331. doi:10.1007/3-540-44622-2_21.

[27] A. Bove, P. Dybjer, U. Norell, A brief overview of Agda – a functional language with dependent types, in: TPHOLs'09, Springer, 2009, pp. 73–78. doi:10.1007/978-3-642-03359-9_6.

[28] K. Nakata, T. Uustalu, Resumptions, weak bisimilarity and big-step semantics for while with interactive I/O: an exercise in mixed induction-coinduction, in: L. Aceto, P. Sobocinski (Eds.), SOS'10, Vol. 32 of EPTCS, 2010, pp. 57–75. doi:10.4204/EPTCS.32.5.

[29] R. Milner, Processes: A mathematical model of computing agents, in: Logic Colloquium '73, Studies in logic and the foundations of mathematics, North-Holland Pub. Co., 1975, pp. 157–153.

[30] T. Uustalu, Coinductive big-step semantics for concurrency, in: N. Yoshida, W. Vanderbauwhede (Eds.), PLACES'13, Vol. 137 of EPTCS, 2013, pp. 63–78. doi:10.4204/EPTCS.137.6.

[31] E. Moggi, Notions of computation and monads, Inf. Comput. 93 (1) (1991) 55–92. doi:10.1016/0890-5401(91)90052-4.

[32] P. Cenciarelli, E. Moggi, A syntactic approach to modularity in denotational semantics, in: Category Theory and Computer Science, 1993.

[33] C. Bach Poulsen, P. D. Mosses, P. Torrini, Imperative polymorphism by store-based types as abstract interpretations, in: PEPM'15, ACM, 2015, pp. 3–8. doi:10.1145/2678015.2682545.

[34] C. Bach Poulsen, P. D. Mosses, Divergence as state in coinductive big-step semantics, presented at NWPT'14 (2014).

[35] P. Bahr, G. Hutton, Calculating correct compilers, Journal of Functional Programming 25. doi:10.1017/S0956796815000180.

[36] X. Leroy, Formal certification of a compiler back-end or: Programming a compiler with a proof assistant, in: POPL '06, ACM, 2006, pp. 42–54. doi:10.1145/1111037.1111042.

[37] A. K. Wright, M. Felleisen, A syntactic approach to type soundness, Inf. Comput. 115 (1) (1994) 38–94. doi:10.1006/inco.1994.1093.

[38] R. Harper, C. Stone, A type-theoretic interpretation of Standard ML, in: G. Plotkin, C. Stirling, M. Tofte (Eds.), Proof, Language, and Interaction, MIT Press, Cambridge, MA, USA, 2000, pp. 341–387.

[39] P. Cousot, Types as abstract interpretations, in: POPL'97, ACM, 1997, pp. 316–331. doi:10.1145/263699.263744.

[40] M. S. Ager, D. Biernacki, O. Danvy, J. Midtgaard, A functional correspondence between evaluators and abstract machines, in: PPDP '03, ACM, 2003, pp. 8–19. doi:10.1145/888251.888254.

[41] O. Danvy, K. Millikin, On the equivalence between small-step and big-step abstract machines: A simple application of lightweight fusion, Inf. Process. Lett. 106 (3) (2008) 100–109. doi:10.1016/j.ipl.2007.10.010.

[42] R. J. Simmons, I. Zerny, A logical correspondence between natural semantics and abstract machines, in: PPDP'13, ACM, 2013, pp. 109–119. doi:10.1145/2505879.2505899.