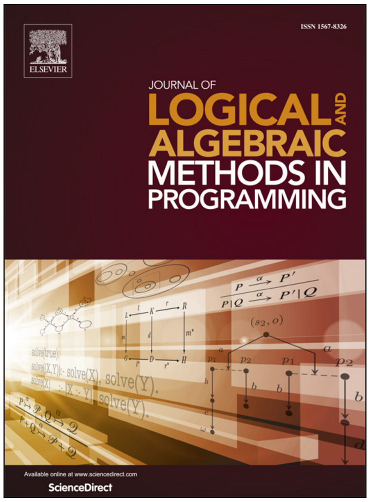
**Paper:**

Bach Poulsen, C. & Mosses, P. (2016). Flag-based big-step semantics. *Journal of Logical and Algebraic Methods in Programming*

http://dx.doi.org/10.1016/j.jlamp.2016.05.001

# Flag-based big-step semantics

Casper Bach Poulsen, Peter D. Mosses



PII: S2352-2208(16)30031-1
DOI: http://dx.doi.org/10.1016/j.jlamp.2016.05.001
Reference: JLAMP 118

To appear in: *Journal of Logical and Algebraic Methods in Programming*

Received date: 26 January 2015
Revised date: 30 March 2016
Accepted date: 9 May 2016

Please cite this article in press as: C. Bach Poulsen, P.D. Mosses, Flag-based big-step semantics, *J. Log. Algebraic Methods Program.* (2016), http://dx.doi.org/10.1016/j.jlamp.2016.05.001

## Highlights

- We present a novel lightweight approach to representing divergence and abrupt termination in structural operational semantics.
- Our approach to divergence alleviates the duplication problem in big-step semantics and uses fewer rules and premises than traditional approaches in the literature.
- We illustrate how threading of the flag arguments through evaluation formulae could be automated. This facilitates deriving flag-based big-step rules from standard big-step SOS rules.
- We present a novel proof method for proving the equivalence of diverging computations in small-step and big-step SOS with non-deterministic interactive input.

# Flag-Based Big-Step Semantics

Casper Bach Poulsen[1,*], Peter 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 approach to representing divergence and abrupt termination in big-step semantics using status flags. This 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 serious *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 $c_2$ is an *infinite* computation, i.e., *diverges*? 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
[1] Present address: Programming Languages Group, Delft University of Technology, Mekelweg 4, 2628 CD Delft, Netherlands

*Preprint submitted to Journal of Logical and Algebraic Methods in Programming* *May 18, 2016*

Here, the first premise in B-$\infty$-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:[2]

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

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-$\infty$-Seq2, and B-Exc-Seq2).

Charguéraud [8] introduced the novel *pretty-big-step* style of big-step SOS. Pretty-big-step rules avoid the duplication problem by breaking 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 introduce an intermediate expression constructor, $\mathsf{seq2}$, and use *outcomes* $o$ to range over either convergence at a store $\sigma$ ($\mathsf{conv}\ \sigma$), divergence ($\mathsf{div}$), 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 $\mathsf{abort}$ 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 has 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 rule 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 reuse the idea from pretty-big-step semantics of interpreting the same set of rules both inductively and coinductively, and adopt and adapt the technique (due to Klin [9, p. 216]) for modular specification of abrupt termination in small-step Modular SOS [9]. The idea is to make program states and result states record an additional *status flag* (ranged over by $\delta$) which indicates that the current state is either convergent ($\downarrow$), divergent ($\uparrow$), or abruptly terminated ('$\mathsf{exc}(v)$'):

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

Here, derivations continue only so long as they are in a convergent state, indicated by $\downarrow$. In order to propagate divergence or abrupt termination, we use pretty-big-step inspired abort rules:

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

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 $\sigma'$ in F-Div and F-Exc. Using these status flags, we can prove that terms diverge similarly to using either traditional big-step divergence rules or pretty-big-step semantics. For example, using pretty-big-step, the proposition $(c, \sigma) \Downarrow \mathsf{div}$ is coinductively proved when $(c, \sigma)$ has an infinite derivation tree; similarly, using the $\uparrow$ status flag, the proposition, for any $\sigma'$, $(c, \sigma, \downarrow) \Rightarrow \sigma', \uparrow$ is coinductively proved when $(c, \sigma, \downarrow)$ has an infinite derivation tree.

We call this style of rules *flag-based big-step semantics*. Flag-based big-step semantics:

[2] 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. The duplication problem occurs in any case for constructs with more than two premises.

- supports propagating exceptions without the intermediate expression forms found in pretty-big-step rules (such as seq2);[3]
- uses fewer rules and premises than both the traditional and the pretty-big-step approach;
- 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 (Sect. 2). Next, we recall traditional approaches to representing possibly-diverging computations in big-step semantics, and how to prove equivalence between semantics for these approaches (Sect. 3). Thus equipped, we make the following contributions:

- We present a novel approach to representing divergence and abrupt termination (Sect. 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. Our approach uses fewer rules and premises than both small-step semantics and pretty-big-step semantics.
- We consider how the approach scales to more interesting language features (Sect. 5), including non-deterministic interactive input. A problem in this connection is that the traditional proof method for relating diverging computations in small-step and big-step SOS works 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.
- The explicit use of flags in our approach makes the rules somewhat tedious to read and write. We suggest leaving the flag arguments implicit in almost all rules (Sect. 6). The way we specify this is similar to Implicitly-Modular SOS (I-MSOS) [10].

Our experiments show that flag-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 previous approaches in Sect. 7 and conclude in Sect. 8.

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

We use a simple While-language as a running example. Its abstract 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 |
| $\oplus \in \{+, -, *\}$ | Binary operations<br>on natural numbers |

[3] There is, however, nothing to prevent us from using the flag-based approach for propagating divergence in pretty-big-step rules. Section 5 discusses why this could be attractive for certain applications.

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

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

Figure 1: Big-step semantics for expressions

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

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

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

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

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

Figure 2: Small-step semantics for commands

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 to which variable $x$ is mapped (if any) in the store $\sigma$. The notation $\sigma[x \mapsto v]$ denotes the update of store $\sigma$ with value $v$ at variable $x$. We use $\text{dom}(\sigma)$ to denote the domain of a map, and $\{x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots\}$ to denote the map from $x_1$ 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.

### *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 and the signature of the evaluation relation are given in Fig. 1.[4] A judgment $(e, \sigma) \Rightarrow_{\text{E}} v$ says that evaluating $e$ in the store $\sigma$ 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 $\sigma$. A judgment $(c, \sigma) \to (c', \sigma')$ asserts the possibility of a transition from the state $(c, \sigma)$ to the state $(c', \sigma')$.

[4] Following Reynolds [11], this relation is *trivial*, and expression 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 further in Sect.5.

### *2.3. Finite Computations*

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

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

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

The following factorial function is an example of a program with a finite sequence of transitions for any natural number $n$:

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

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:

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

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:

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

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

Here and throughout this article, we use '$\mathsf{co}$' on the left of rules to indicate that the relation is coinductively defined by that set of rules.[5] 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 $\overset{\infty}{\to}$, and since the rule Trans$\infty$ cannot be used to construct finite derivation trees, the relation 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 $\mathsf{while}\ 1\ \mathsf{skip}$ diverges by using $(\mathsf{while}\ 1\ \mathsf{skip}, \cdot) \overset{\infty}{\to}$ 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:

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

[5] 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.

There also exists an infinite derivation tree whose conclusion is:

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

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

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

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 $\{\mathrm{x} \mapsto 1\}$ that does not match this coinduction hypothesis:

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

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

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

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 online at: `http://www.plancomps.org/flag-based-big-step/`.

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 the previous section (always extending the big-step semantics for expressions defined in Fig. 1).

### *3.1. Inductive Big-Step Semantics*

The big-step rules in Fig. 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 $\sigma$ terminates with a final store $\sigma'$. This corresponds to arriving at a state $(\mathsf{skip}, \sigma)$ by analogy with the small-step semantics in Sect. 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:

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

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

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

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

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

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

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

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)

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

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

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

*Proof.* Straightforward proof by induction on $\to$. □

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

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

### 3.2. *Coinductive Big-Step Divergence Predicate*

The rules in Fig. 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 Sect. 2:

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

Here, $\mathsf{alloc}\ \mathrm{x}$ and $\mathrm{x} := 0$ converge, but the $\mathsf{while}$ command diverges. The big-step divergence predicate uses the standard inductive big-step relation for converging branches. Consequently, we need the rules in *both* Figures 3 and 4 to express divergence. This increases the number 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 $\overset{\infty}{\rightarrow}$ and $\overset{\infty}{\Rightarrow}$.

**Theorem 5.** $(c, \sigma) \overset{\infty}{\rightarrow}$ *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* $(\exists \sigma'.\ (c, \sigma) \rightarrow^{*} (\mathsf{skip}, \sigma'))$ *or* $(c, \sigma) \overset{\infty}{\rightarrow}$.

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

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

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

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

*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 propagation of 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, the following rules define sequential composition for a pretty-big-step relation with the signature '$(C, \sigma) \Downarrow o$', using the semantic constructor $\mathsf{seq2}$:

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

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 was convergence with some store $\sigma$, and goes on to evaluate $c_2$ under that store. 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 both to read and write, but are straightforward to generate automatically.

The pretty-big-step rules in Fig. 5 have a *dual* interpretation: such rules define two separate relations, one *inductive*, the other *coinductive*. We use '$\mathsf{du}$' on the left of rules to indicate relations with dual interpretations. We use $\Downarrow$ to refer to the inductive interpretation, and $\Downarrow^{\mathsf{co}}$ 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 $\Downarrow^{\mathsf{du}}$. 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.

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

*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* $o$, $(\mathsf{while}\ 1\ \mathsf{skip}, \cdot) \Downarrow^{\mathsf{co}} o$.

*Proof.* Straightforward proof by coinduction. □

An important property of the rules in Fig. 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 proved 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* $(c, \sigma) \Downarrow^{\mathsf{co}} \mathsf{div}$.

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

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

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

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

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

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

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

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

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

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

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

Figure 5: Pretty-big-step semantics for commands

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

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

Our pretty-big-step semantics uses 18 rules (including rules for $\Rightarrow_{\mathrm{E}}$) with 16 premises (counting judgments about both $\Rightarrow_{\mathrm{E}}$ and $\Downarrow^{\mathsf{du}}$), none of which are duplicates. In contrast, the union of the rules for inductive standard big-step rules in Sect. 3.1 and the divergence predicate in Sect. 3.2 uses 17 rules with 25 premises of which 6 are duplicates. Pretty-big-step semantics solves the duplication problem, albeit 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. Flag-Based Big-Step Semantics

In this section we present a novel approach to representing divergence and abrupt termination in big-step semantics, which does not require introducing new relations, and relies on fewer, simpler rules for propagation of divergence and/or abrupt termination. The approach can be used to augment standard inductively defined rules to allow them to express divergence on a par with standard divergence predicates or pretty-big-step rules.

### *4.1. The While-Language and its Flag-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 (↓) or divergence (↑), ranged over by:

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

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

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

$$\textsf{du}\,\dfrac{}{(e,\sigma,\uparrow) \Rightarrow_{\text{FE}} v, \uparrow}\,\text{FE-Div}$$

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

$$\textsf{du}\,\dfrac{}{(\textsf{skip},\sigma,\downarrow) \Rightarrow_{\text{F}} \sigma, \downarrow}\,\text{F-Skip} \qquad \textsf{du}\,\dfrac{x \notin \operatorname{dom}(\sigma)}{(\textsf{alloc}\ x,\sigma,\downarrow) \Rightarrow_{\text{F}} \sigma[x \mapsto \textsf{null}], \downarrow}\,\text{F-Alloc}$$

$$\textsf{du}\,\dfrac{x \in \operatorname{dom}(\sigma) \qquad (e,\sigma,\downarrow) \Rightarrow_{\text{FE}} v, \delta}{(x := e,\sigma,\downarrow) \Rightarrow_{\text{F}} \sigma[x \mapsto v], \delta}\,\text{F-Assign} \qquad \textsf{du}\,\dfrac{(c_1,\sigma,\downarrow) \Rightarrow_{\text{F}} \sigma', \delta \qquad (c_2,\sigma',\delta) \Rightarrow_{\text{F}} \sigma'', \delta'}{(c_1; c_2,\sigma,\downarrow) \Rightarrow_{\text{F}} \sigma'', \delta'}\,\text{F-Seq}$$

$$\textsf{du}\,\dfrac{\begin{array}{c}v \neq 0 \\ (e,\sigma,\downarrow) \Rightarrow_{\text{FE}} v, \delta \qquad (c_1,\sigma,\delta) \Rightarrow_{\text{F}} \sigma', \delta'\end{array}}{(\textsf{if}\ e\ c_1\ c_2,\sigma,\downarrow) \Rightarrow_{\text{F}} \sigma', \delta'}\,\text{F-If} \qquad \textsf{du}\,\dfrac{(e,\sigma,\downarrow) \Rightarrow_{\text{FE}} 0, \delta \qquad (c_2,\sigma,\delta) \Rightarrow_{\text{F}} \sigma', \delta'}{(\textsf{if}\ e\ c_1\ c_2,\sigma,\downarrow) \Rightarrow_{\text{F}} \sigma', \delta'}\,\text{F-IfZ}$$

$$\textsf{du}\,\dfrac{\begin{array}{cc}(e,\sigma,\downarrow) \Rightarrow_{\text{FE}} v, \delta & v \neq 0 \\ (c,\sigma,\delta) \Rightarrow_{\text{F}} \sigma', \delta' & (\textsf{while}\ e\ c,\sigma',\delta') \Rightarrow_{\text{F}} \sigma'', \delta''\end{array}}{(\textsf{while}\ e\ c,\sigma,\downarrow) \Rightarrow_{\text{F}} \sigma'', \delta''}\,\text{F-While} \qquad \textsf{du}\,\dfrac{(e,\sigma,\downarrow) \Rightarrow_{\text{FE}} 0, \delta}{(\textsf{while}\ e\ c,\sigma,\downarrow) \Rightarrow_{\text{F}} \sigma, \delta}\,\text{F-WhileZ}$$

$$\textsf{du}\,\dfrac{}{(c,\sigma,\uparrow) \Rightarrow_{\text{F}} \sigma', \uparrow}\,\text{F-Div}$$

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

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

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

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

How do these rules support reasoning about divergence? In the F-Seq rule in Fig. 6, the first premise may diverge to produce ↑ in place of $\delta$ in the first premise. If this is the case, any subsequent computation is irrelevant. To inhibit subsequent computation we introduce *divergence rules* FE-Div and F-Div, also in Fig. 6. The divergence rules serve a similar purpose to abort rules in pretty-big-step semantics: they propagate divergence as it arises and inhibit further evaluation.

A technical curiosity of the divergence rule F-Div is that it allows evaluation to return an *arbitrary* store. In other words, states record and propagate irrelevant information. This is a somewhat unusual way of propagating semantic information in big-step semantics; however, the intuition behind it follows how divergence and abrupt termination are traditionally propagated in big-step SOS. 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 Sect. 1:

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

These rules "forget" the structure of whatever other effects a diverging computation produces. Similarly, flag-based big-step semantics retains the relevant structure of configurations, but allows us to choose arbitrary values for the irrelevant parts: for converging computations, all parts of the configuration are relevant, whereas for divergent or abruptly terminated configurations, only the fact that we are abruptly terminating is important.

A key property of flag-based divergence is that the conclusions of 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.** *$(c, \sigma, \downarrow) \Rightarrow_{\mathrm{F}} \sigma', \delta$ implies $\delta \neq \uparrow$.*

*Proof.* Straightforward proof by induction on $\Rightarrow_{\mathrm{F}}$. □

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 $(c, \sigma, \downarrow) \Rightarrow_{\mathrm{F}} \sigma', \downarrow$.*

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

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

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

*Proof (classical).* The $\overset{\infty}{\Rightarrow}$-to-$\overset{\mathsf{co}}{\Rightarrow}_{\mathrm{F}}$ direction follows by straightforward coinduction, using Theorem 17. The other direction follows from Lemma 19, Lemma 20, and the law of excluded middle for case analysis on whether branches converge or not. □

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

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

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

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

*Proof.* Straightforward proof by induction on $\Rightarrow_{\mathrm{F}}$. □

Theorem 17 proves that we can choose *any* store $\sigma'$ when constructing a proof of divergence for some $c$ and $\sigma$. Lemma 21 summarises this observation, by proving that the choice of $\sigma'$ is, in fact, irrelevant.

**Lemma 21.** *For any $c$ and any store $\sigma$, $(\forall\sigma'.(c, \sigma, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{F}} (\sigma', \downarrow))$ iff $(\exists\sigma'.\ (c, \sigma, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{F}} (\sigma', \downarrow))$.*

*Proof.* The ∀-to-∃ direction is trivial. The other direction follows by straightforward coinduction. □

As with pretty-big-step semantics (Sect. 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 $\mathsf{while}\ 1\ \mathsf{skip}$ coevaluates to anything:

**Example 22.** *For any $\sigma'$ and $\delta$, $(\mathsf{while}\ 1\ \mathsf{skip}, \cdot, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{F}} \sigma', \delta$.*

*Proof.* Straightforward proof by coinduction. □

Comparing flag-based divergence (Fig. 6) with pretty-big-step (Fig. 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 flag-based big-step rules in Fig. 6 actually do contain some duplicate premises in the rules F-If, F-IfZ, F-While, and F-WhileZ: each of these rules evaluate the same expression $e$. This duplication could have been avoided by introducing a constructor for intermediate expressions for ‘$\mathsf{if}$’ and ‘$\mathsf{while}$’, similar to pretty-big-step.

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

### *4.2. Divergence Rules are Necessary*

From Example 22 we know that some diverging programs result in a ↓ state. But do we really need the ↑ 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 depends crucially on F-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 $\mathsf{null}$ value when it is dereferenced, so this program will attempt to add $\mathsf{null}$ and 0, which is meaningless. The derivation trees we can construct must use the F-Div rule as follows:

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

**Example 23.** *For any $\sigma$, $((\mathsf{while}\ 1\ \mathsf{skip}); \mathsf{alloc}\ \mathrm{x}; \mathrm{x} := \mathrm{x} + 0, \cdot, \downarrow) \overset{\mathsf{co}}{\Rightarrow} (\sigma, \uparrow)$. 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 $\lambda$-calculus with constants, 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 flag-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

Flag-based big-step semantics supports reasoning 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 (which is analogous to Theorem 5 in Sect. 3.2 of this article) for relating diverging computations in small-step and big-step semantics relies crucially on determinism. 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 flag-based big-step semantics. Our proof provides a novel approach to 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:

$$\frac{}{(\mathsf{input}, \sigma, \downarrow) \Rightarrow_{\mathrm{FE}} v, \downarrow}$$

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

The answer to this question 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 Sect. 2 carry over unchanged, since they rely on the expression evaluation relation being equivalent between the two semantics.

But the proof of 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 $c_1$ can make a sequence of transition steps to a $c_1'$ which is stuck, whereas there may exist some alternative sequence of transition steps for $c_1$ such that $c_1; c_2$ diverges.

**Example 24.** *Consider the program state where* x *is an unbound variable:*

$$(\overbrace{\mathsf{if}\ \mathsf{input}\ (\mathrm{x} := 0)\ \mathsf{skip}}^{c_1};\overbrace{\mathsf{while}\ 1\ \mathsf{skip}}^{c_2}, \cdot)$$

*If* $\mathsf{input}$ *returns a non-zero value, evaluation gets stuck, since the command '*$\mathrm{x} := 0$*' is meaningless in the empty store ('*$\cdot$*'). Otherwise, evaluation diverges. Thus it holds that* $(c_1, \cdot) \rightarrow^* (\mathrm{x} := 0, \cdot)$ *and* $(c_1; c_2, \cdot) \overset{\infty}{\rightarrow}$*, but not* $(\mathrm{x} := 0; c_2, \cdot) \overset{\infty}{\rightarrow}$*.*

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

**Lemma 25.** *If* $(c_1; c_2, \sigma) \overset{\infty}{\rightarrow}$ *and* $\neg(\exists \sigma'.\ (c_1, \sigma) \rightarrow^* (\mathsf{skip}, \sigma'))$*, 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 exists a configuration $(c_1', \sigma_1)$ for which $(c_1, \sigma) \rightarrow (c_1', \sigma_1)$. By the second hypothesis, it must also hold that there is no $\sigma'$ such that $(c_1', \sigma_1) \rightarrow^* (\mathsf{skip}, \sigma')$. From these facts, the goal follows by applying Trans$\infty$, the small-step rule S-Seq, and the coinduction hypothesis. □

Using Lemma 25, we can relate infinite sequences of small-step transitions to infinite flag-based big-step derivations as follows.

**Lemma 26.** *For any $\sigma'$, if $(c, \sigma) \overset{\infty}{\to}$ then $(c, \sigma, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{F}} \sigma', \uparrow$.*

*Proof (classical).* The proof is by guarded coinduction, using the goal as the coinduction hypothesis. The critical cases are those for sequential composition and '$\mathsf{while}$': these cases use the law of excluded middle for case analysis on whether $c$ 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 25 above. □

The proof of the other direction, i.e., relating infinite big-step derivations to infinite sequences of small-step transitions is proved using Lemma 28:

**Theorem 27.** *$(c, \sigma) \overset{\infty}{\to}$ iff for any $\sigma'$, $(c, \sigma, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{F}} \sigma', \uparrow$.*

*Proof (classical).* The small-to-big direction is proved in Lemma 25 above. The other direction follows by coinduction and Lemma 28 below. □

**Lemma 28.** *For any $\sigma''$, if $(c, \sigma) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{F}} \sigma'', \uparrow$, then $\exists c', \sigma'.\ \Big((c, \sigma) \to (c', \sigma') \ \wedge\ (c', \sigma') \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{F}} \sigma'', \uparrow\Big)$.*

*Proof.* The proof carries over from Lemma 8, and relies on Theorem 29 below. □

**Theorem 29.** *$(c, \sigma) \to^* (\mathsf{skip}, \sigma')$ 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:

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

We let exceptions record both a thrown value and the state of the store when the exception occurs. We also augment the syntactic sort for commands with a '$\mathsf{throw}\ v$' construct for throwing a value $v$ as an exception, and a '$\mathsf{catch}\ c\ c$' construct for handling exceptions:

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

The rules for these constructs are given by a single rule for throwing an exception, and two rules for propagating exceptions for command and expression evaluation:

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

$$\frac{}{(c, \sigma, \mathsf{exc}(v, \sigma_0)) \Rightarrow_{\mathrm{F}} \sigma', \mathsf{exc}(v, \sigma_0)}\ \text{F-Exc} \qquad \frac{}{(e, \sigma, \mathsf{exc}(v, \sigma_0)) \Rightarrow_{\mathrm{FE}} v', \mathsf{exc}(v, \sigma_0)}\ \text{FE-Exc}$$

Using these rules, exceptions are propagated similarly to divergence.

The following rules specify the semantics of '$\mathsf{catch}\ c\ c$':

$$\frac{(c_1, \sigma, \downarrow) \Rightarrow_{\mathrm{F}} \sigma', \delta \qquad \delta \neq \mathsf{exc}(\_, \_)}{(\mathsf{catch}\ c_1\ c_2, \sigma, \downarrow) \Rightarrow_{\mathrm{F}} \sigma', \delta}\ \text{F-Catch} \qquad \frac{(c_1, \sigma, \downarrow) \Rightarrow_{\mathrm{F}} \sigma', \mathsf{exc}(v, \sigma_0) \qquad (c_2, \sigma_0, \downarrow) \Rightarrow_{\mathrm{F}} \sigma'', \delta}{(\mathsf{catch}\ c_1\ c_2, \sigma, \downarrow) \Rightarrow_{\mathrm{F}} \sigma'', \delta}\ \text{F-Catch-Some}$$

Here, F-Catch handles the case where no exception arises during evaluation of $c_1$. F-Catch-Some detects that an exception has occurred and proceeds to evaluate the handler, $c_2$, in the store recorded in the exception.[6]

[6] The handler $c_2$ here discards the value of the exception. It is straightforward to give flag-based semantics for handlers with patterns that match thrown exception values, e.g., by adapting rules from [20, Sect. 3.3] or [21, Fig. 2].

### *5.3. The Necessity of Choosing Arbitrary Stores*

The F-Throw and F-Exc rules above admit arbitrary $\sigma'$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 22). 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. But 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 values similarly to references in Standard ML [22]. The signature of expression evaluation and the assignment rule in such a language could be:

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

$$\frac{(e_1, \sigma, \downarrow) \Rightarrow_{\text{FE}} x, \sigma', \delta \qquad (e_2, \sigma', \delta) \Rightarrow_{\text{FE}} v, \sigma'', \delta' \qquad x \in \text{dom}(\sigma'')}{(e_1 := e_2, \sigma, \downarrow) \Rightarrow_{\text{FE}} v, \sigma''[x \mapsto v], \delta'}\text{F-RefAsgn}$$

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

A further consequence of a rule like F-RefAsgn above is that the property summarised in Lemma 21 no longer holds: we can no longer prove that a program that diverges has an arbitrary store. For some applications this is unimportant; for others, this state of affairs is unfortunate. For example, a proof assistant like Coq does not support coinductive reasoning about existentially quantified goals in a 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:

$$\text{If } P \text{ then } \exists \sigma'.\ (c, \sigma, \downarrow) \overset{\text{co}}{\Rightarrow} (\sigma', \uparrow)$$

where $P$ is some proposition.

While the need to synthesise semantic information that is not the result of any actual computation may be unattractive for certain applications, flag-based big-step rules do allow abrupt termination and divergence to be propagated correctly. Furthermore, the flag-based approach is also applicable to the more flexible pretty-big-step style: applying a flag-based propagation strategy to pretty-big-step rules would alleviate the need for abort-rules for each semantic expression constructor, since propagation is handled by a single flag-based abort rule instead.[7]

## 6. Implicit Flag-Based Divergence

As illustrated in the previous sections, the flag-based approach can reduce the number of rules and premises required for specifying divergence or abrupt termination in big-step semantics. However, explicit threading of the flag arguments $\delta$ and $\delta'$ through the evaluation formulae in all rules is somewhat tedious, and might discourage adoption of the approach.

In this section, we introduce new notation for signatures. Using such signatures, the flag arguments can be left implicit in almost all rules. For instance, we can specify the While language with the signatures specified below simply by adding the divergence rules FE-Div and F-Div from Fig. 6 to the original big-step rules given in Figs. 1 and 3.

Recall the original signatures used for the big-step semantics of While:

$$\boxed{(e, \sigma) \Rightarrow_{\text{E}} v} \qquad \boxed{(c, \sigma) \Rightarrow_{\text{B}} \sigma'}$$

[7] This approach was explored in previous work [23, Sect. 3.1] by the authors.

Our new signatures for the flag-based semantics of While are as follows:

$$\boxed{(e,\sigma\colorbox{lightgray}{$,\delta{:}{-}{\downarrow}$})\Rightarrow_{\mathrm{FE}} v\colorbox{lightgray}{$,\delta'$}} \qquad \boxed{(c,\sigma\colorbox{lightgray}{$,\delta{:}{-}{\downarrow}$})\Rightarrow_{\mathrm{F}} \sigma'\colorbox{lightgray}{$,\delta'$}}$$

The highlighting of the arguments $\delta$ and $\delta'$ specifies formally that they can either be omitted or made explicit (uniformly) when the evaluation relation concerned is used in a rule. The notation '$\delta{:}{-}{\downarrow}$' indicates that $\downarrow$ is the *default* value of the flag $\delta$.[8] Rules written using just the non-highlighted parts of the signatures abbreviate flag-based rules as follows:

- When a rule without explicit flags has no evaluation premises, it abbreviates the flag-based rule where the flags of both the source and target of the conclusion are $\downarrow$. For example,

$$\frac{}{(\mathsf{skip},\sigma)\Rightarrow_{\mathrm{F}}\sigma}\,\text{F-Skip}\quad\text{abbreviates}\quad\mathsf{du}\,\frac{}{(\mathsf{skip},\sigma,\downarrow)\Rightarrow_{\mathrm{F}}\sigma,\downarrow}\,\text{F-Skip.}$$

- When a rule without explicit flags has $n$ evaluation premises, it abbreviates the flag-based rule where the flags of the source of the first evaluation premise and of the source of the conclusion are $\downarrow$; the flags of the target of evaluation premise $i$ and of the source of evaluation premise $i+1$ are $\delta_i$ (for $1 \le i < n$); and the flags of the target of evaluation premise $n$ and the conclusion are $\delta'$. For example,

$$\frac{(c_1,\sigma)\Rightarrow_{\mathrm{F}}\sigma' \qquad (c_2,\sigma')\Rightarrow_{\mathrm{F}}\sigma''}{(c_1;c_2,\sigma)\Rightarrow_{\mathrm{F}}\sigma''}\,\text{F-Seq}\quad\text{abbreviates}\quad\mathsf{du}\,\frac{(c_1,\sigma,\downarrow)\Rightarrow_{\mathrm{F}}\sigma',\delta_1 \qquad (c_2,\sigma',\delta_1)\Rightarrow_{\mathrm{F}}\sigma'',\delta'}{(c_1;c_2,\sigma,\downarrow)\Rightarrow_{\mathrm{F}}\sigma'',\delta'}\,\text{F-Seq.}$$

Replacing the signatures in Figs. 1 and 3 by those given above, the big-step rules there abbreviate all the flag-based rules given in Fig. 6 (up to renaming of flag variables), so all that is needed is to add the two explicit divergence rules (FE-Div and F-Div).

*The Definition of Standard ML* [22] introduced the technique of letting arguments of evaluation relations remain implicit in big-step rules, calling it "the store convention". I-MSOS [10] proposed the use of highlighting to specify formally which arguments can be omitted, and defined I-MSOS by translation to MSOS [9], but the notation provided there does not support the intended use of the default $\downarrow$ in flag-based big-step rules. It should however be possible to generalise I-MSOS to support flags, and thereby allow further arguments to be omitted. For instance, the I-MSOS signature $\colorbox{lightgray}{$($}\,c\,\colorbox{lightgray}{$,\sigma)$}\Rightarrow_{\mathrm{F}}\colorbox{lightgray}{$($}\,\colorbox{lightgray}{$\sigma'$}\,\colorbox{lightgray}{$)$}$ allows the $\sigma$ and $\sigma'$ arguments to be omitted as follows:

$$\frac{}{\mathsf{skip}\Rightarrow_{\mathrm{F}}(\,)}\,\text{F-Skip}\qquad\frac{c_1\Rightarrow_{\mathrm{F}}(\,)\qquad c_2\Rightarrow_{\mathrm{F}}(\,)}{c_1;c_2\Rightarrow_{\mathrm{F}}(\,)}\,\text{F-Seq}$$

The highlighting in the following signature could therefore specify that the above rules abbreviate flag-based MSOS rules:

$$\boxed{\colorbox{lightgray}{$($}\,c\,\colorbox{lightgray}{$,\sigma,\delta{:}{-}{\downarrow})$}\Rightarrow_{\mathrm{F}}(\,\colorbox{lightgray}{$\sigma',\delta'$}\,)}$$

Further development of the details of implicit flag-based I-MSOS is out of the scope of this paper, and left to future work.

## 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.

[8] '$\delta{:}{-}{\downarrow}$' is reminiscent of Prolog: it indicates that $\downarrow$ is the condition for $\delta$ to allow evaluation to proceed normally.

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 [24] 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 [25] and Abel and Chapman [26] (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 [27] study the category theoretic foundations of the resumption monad. Related to the resumption monad is the interactive I/O monad by Hancock and Setzer [28] 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 recursive/corecursive functions in order to use it in proof assistants. Agda [29] provides native support for such function definitions, while Coq does not. Nakata and Uustalu [24; 30] 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 can be subtle to implement in Coq. Propagating divergence between premises in trace-based big-step semantics is also slightly more involved than the approach taken in pretty-big-step semantics and by us: propagating divergence between premises in trace-based semantics entails a coinductive proof for each premise (proving that the trace is infinite). In contrast, propagation in both pretty-big-step semantics and our approach relies on simple case analysis (i.e., via abort rules or inspecting the status flag).

Trace-based semantics provides a strong foundation for constructively reasoning about possibly-diverging programs, but unfortunately, as discussed earlier, 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 involving existentials unnecessarily involved. For example, proving a goal of the following form is not possible by guarded coinduction alone:

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

Here, $P$ is some proposition, and $\tau$ is a trace. Proving such propositions can be done by manually constructing a witness function for $\tau$, which is not always trivial. Charguéraud [8] directs a similar criticism at Coq's lack of support for coinduction.

Hur et al. [31] present a novel means of doing coinductive proofs. The idea is to use a *parameterised greatest fixed-point* for proofs by coinduction. Their Coq library, *Paco*, provides an implementation of the method, which supports proofs by coinduction using a more flexible notion of guardedness than the naive syntactic guardedness check which Coq implements. This line of work provides a promising direction for more tractable coinductive proofs.

Looking beyond Coq, the support for features pertaining to coinduction seems more progressive in the Agda proof assistant. Indeed, Danielsson [25] leverages some of this support by using mutually recursive/corecursive functions (which Coq does not support) in his big-step functional operational semantics for the $\lambda$-calculus. Also working in Agda, Abel and Chapman [26] use *sized types* [32] for their proofs by coinduction.

Nakata and Uustalu also give coinductive big-step semantics for *resumptions* [30] (denoting the external behaviour of a communicating agent in concurrency theory [33]) as well as interleaving and concurrency [34]. 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 Sect. 5.1. It is, however, also somewhat more involved to specify and work with.

Owens et al. [35] argue that step-indexed semantics, i.e., using a counter which is decremented with each recursive call, is an under-utilised technique for representing big-step semantics with possible divergence. By augmenting a big-step semantics with counters, it becomes possible to represent it as a function in logic such that it is guaranteed to terminate: when the clock runs out, a special "time-out" value is produced. This permits reasoning about divergence, since the set of diverging programs is exactly the set of programs for which there does not exist a finitely-valued counter such that the program successfully terminates without timing out. This provides an alternative that avoids the duplication problem with big-step semantics but still supports reasoning about diverging computations, albeit somewhat more indirectly than representing infinite computations as infinite derivation trees.

Moggi [36] suggested monads as a means to obtain modularity in denotational semantics [37]. In a similar vein, Modular SOS [9] provides a means to obtain modularity in operational semantics. The flag-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 [23] observed that encoding abrupt termination as a stateful flag instead scales better to big-step rules by avoiding such explicit handlers. Flag-based divergence was used in [38] for giving a semantics for the untyped $\lambda$-calculus. The first author's thesis [39, Cpt. 4 and 5] describes a rule format for which small-step and pretty-big-step rules are provably equivalent using flag-based abrupt termination and divergence.

Our work differs from [23; 38; 39] in several ways: unlike [23] it considers both inductive and coinductive big-step semantics and it is based on SOS. In our work with Torrini [38], we proposed the flag-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. Another difference from this paper is that both [23; 39] mainly considers pretty-big-step, whereas the flag-based big-step style considered here has a more traditional big-step flavour. The first author's thesis [39] also uses the generalised coinductive proof technique described in Sect. 5.1 for relating divergent small-step and pretty-big-step semantics with limited non-determinism. Whereas [39] applies the technique to a variant of MSOS in pretty-big-step style, this paper shows that the technique also scales to flag-based big-step SOS.

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 [40] also use big-step semantics for their compiler correctness proofs. However, in *CompCert*, Leroy [41] uses a small-step semantics and sophisticated notions of bisimulation for its compiler correctness proofs.

Wright and Felleisen introduce the syntactic approach to type safety in [42]. They survey type safety proofs based on denotational and big-step semantics, and conclude that small-step semantics is a better fit for proving type safety by progress and preservation lemmas. Harper and Stone [43] 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 proved, 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 proved 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 [44]; Danielsson's work on operational semantics using the partiality monad [25]; 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 [45; 46; 47], whereby the standard small-step type safety proof technique applies. A preliminary study [38] 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 [44].

## 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 the flag arguments of the evaluation relations can be generated automatically. We also considered how to extend our semantics with interactive input, and provided a generalisation of the traditional proof method for relating diverging small-step and big-step semantics.

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

*Acknowledgements.* We thank the anonymous reviewers for their constructive suggestions for improving this paper. Thanks also to Neil Sculthorpe, Paolo Torrini, and Ulrich Berger for their helpful comments on a previous version of 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, Inf. Comput. 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?, J. Functional Program. 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, pp. 220–236. doi:10.1007/978-3-319-14125-1_13.

[8] A. Charguéraud, Pretty-big-step semantics, in: 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 '72, ACM Annual Conference, ACM, New York, NY, USA, 1972, pp. 717–740. doi:10.1145/800194.805852.

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

[13] B. C. Pierce, Types and Programming Languages, MIT Press, Cambridge, MA, USA, 2002.

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

[15] T. Coquand, G. Huet, The calculus of constructions, Inf. Comput. 76 (2–3) (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, Cambridge, MA, USA, 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, electronic textbook.
URL http://cis.upenn.edu/~bcpierce/sf

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

[20] M. Churchill, P. D. Mosses, N. Sculthorpe, P. Torrini, Reusable components of semantic specifications, in: Trans. Aspect-Oriented Software Development XII, Vol. 8989 of LNCS, Springer, 2015, pp. 132–179. doi:10.1007/978-3-662-46734-3_4.

[21] M. Churchill, P. D. Mosses, Modular bisimulation theory for computations and values, in: FoSSaCS'13, Vol. 7794 of LNCS, Springer, 2013, pp. 97–112. doi:10.1007/978-3-642-37075-5_7.

[22] R. Milner, M. Tofte, R. Harper, D. MacQueen, The Definition of Standard ML, MIT Press, Cambridge, MA, USA, 1997.

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

[24] K. Nakata, T. Uustalu, Trace-based coinductive operational semantics for While, in: TPHOLs'09, Vol. 5674 of LNCS, Springer, 2009, pp. 375–390. doi:10.1007/978-3-642-03359-9_26.

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

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

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

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

[29] 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.

[30] K. Nakata, T. Uustalu, Resumptions, weak bisimilarity and big-step semantics for While with interactive I/O: An exercise in mixed induction-coinduction, in: SOS'10, Vol. 32 of EPTCS, 2010, pp. 57–75. doi:10.4204/EPTCS.32.5.

[31] C. Hur, G. Neis, D. Dreyer, V. Vafeiadis, The power of parameterization in coinductive proof, in: POPL'13, ACM, New York, NY, USA, 2013, pp. 193–206. doi:10.1145/2429069.2429093.

[32] A. Abel, MiniAgda: Integrating sized and dependent types, in: PAR'10, Vol. 43 of EPTCS, 2010, pp. 14–28. doi:10.4204/EPTCS.43.2.

[33] 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.

[34] T. Uustalu, Coinductive big-step semantics for concurrency, in: PLACES'13, Vol. 137 of EPTCS, 2013, pp. 63–78. doi:10.4204/EPTCS.137.6.

[35] S. Owens, M. O. Myreen, R. Kumar, Y. K. Tan, Functional big-step semantics, in: ESOP'16, Vol. 9632 of LNCS, Springer, 2016, to appear.

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

[37] P. Cenciarelli, E. Moggi, A syntactic approach to modularity in denotational semantics, in: Category Theory and Computer Science, Proc. 5th Intl. Conf., 1993.

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

[39] C. Bach Poulsen, Extensible transition system semantics, Ph.D. thesis, Swansea University, to appear (2016).

[40] P. Bahr, G. Hutton, Calculating correct compilers, J. Functional Program. 25 (2015) e14 (47 pages). doi:10.1017/S0956796815000180.

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

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

[43] 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.

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

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

[46] 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.

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