Transformations Between Structural Specifications of Programming Language 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, big-step, and pretty-big-step. The big-step style also has several variants (inductive, coinductive, trace-based), which are intended for different purposes. Two significant issues are that these variants need to be specified completely independently, and that representing divergence and abrupt termination gives rise to some annoying duplication between rules.

We present a novel variant of the big-step style. It avoids the duplication problem, and uses fewer rules and premises for representing divergence than previous approaches in the literature. Using implicitly-modular structural operational semantics (I-MSOS), we also show how to automatically transform standard big-step rules into our variant, as well as into a trace-based variant.

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

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 has lower interpretive overhead, making it more suitable as a basis for efficient interpreters [3; 4]. Different styles can also be used for specifying different fragments of the same language.

Style is not the only representational choice when specifying rules. For example, inductively defined relations are usually chosen for finite computations; coinductively defined divergence predicates [5] for infinite computations; trace-based semantics [6; 7] for possibly-infinite computations; etc. In small-step semantics, such variants are typically expressed in terms of the *same* small-step relation, whereby there is no need to reformulate the small-step rules themselves. In contrast, different variants of big-step rules are usually specified independently, and new rules are required each time a new variant is needed. Formulating variants of the same rules is both tedious and error-prone. Furthermore, a commonly-held belief is that big-step

*Corresponding author

semantics suffers from an inherent *duplication problem* [8] when expressing diverging computations. For these reasons, big-step semantics is often abandoned in favour of the small-step style.

Here, we present a novel encoding of divergence and abrupt termination that eliminates the duplication problem. We also show how to automatically transform between variants of big-step rules. This solves two significant problems often associated with big-step semantics. Our contributions fall into two categories: *methodological* and *technical*. Methodologically, we present a novel encoding of divergence that supports concise representation of divergence in big-step semantics. Our encoding supports reasoning about possibly-infinite computations on a par with small-step semantics, and eliminates the big-step duplication problem for diverging and abruptly terminating computations. Our approach is *generic* in the sense that it does not depend on underlying language constructs or language paradigms. We then show *technically* how to transform standard big-step rules to obtain interesting variants of the semantics (e.g., generic divergence and trace-based semantics) without manual reformulation or introduction of new rules for existing constructs. We use the Coq proof assistant [9] for providing proofs of correspondence between different representations of the same semantics, and compare proofs for small-step and (transformed) big-step variants.

Throughout the article we use a simple while-language as our running example. We first 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 provide proofs of relationships between some of the recalled variants (Sect. 3). Thus equipped, we make the following contributions:

- We present a novel generic approach to representing divergence and abrupt termination: *generic divergence* (Sect. 4). Generic divergence alleviates the duplication problem in big-step semantics. For all examples considered by the authors so far, including applicative and imperative languages, generic divergence straightforwardly allows expressing divergence in a way that does not depend on the underlying language or paradigm. Comparing with alternatives in the literature, such as *pretty-big-step* semantics [8], our approach is equally expressive, but uses fewer rules and premises. This shows that it is a misconception that big-step semantics is inherently bad at dealing with divergence and abrupt termination.

- 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 framework suffices for transforming standard big-step rules into variants with generic divergence (Sect. 5) and trace-based semantics (Sect. 6) alike. This alleviates the manual labour involved in giving different variants of the same big-step semantics.

- We show how our extension of I-MSOS facilitates transformations that allow us to *calculate* [11] a *definite assignment* analysis using the proof of safety of the analysis as a guiding principle (Sect. 7). Comparing small-step and big-step versions of such safety proofs we conclude that, in the case of definite assignment, big-step has a slight edge over small-step.

Our experiments show that transformational techniques as embodied in I-MSOS are a viable tool for deriving interesting variants of structural specifications of programming languages, including our novel approach to concisely representing divergence using big-step semantics.

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

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

Figure 1: Big-step semantics for expressions

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

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

$Var \ni x ::= \mathrm{x} \mid \mathrm{y} \mid \ldots$	Variables
$\mathbb{N} \ni n ::= 0 \mid 1 \mid \ldots$	Natural numbers
$Cmd \ni c ::= \mathsf{skip} \mid \mathsf{alloc}\ x \mid x := e \mid c; c \mid \mathsf{ifnz}\ e\ c\ c \mid \mathsf{whilenz}\ e\ c$	Commands
$Val \ni v ::= \mathsf{null} \mid n$	Values
$Expr \ni e ::= v \mid x \mid bop(e,e)$	Expressions
$bop \in \{+,-,*\}$	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 $\sigma \in Var \xrightarrow{\text{fin}} Val$ 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 map σ' such that $\sigma'(x) = v$, and $\sigma'(x') = \sigma(x')$ for x' different from 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. The remainder of this section introduces a mixed big-step and small-step semantics for this language, and introduces conventions we shall use.

2.1. Big-Step Expression Evaluation Relation

Big-step semantics is a natural choice for specifying the semantics for expressions which, 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. 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 $(c,\sigma) \to (c',\sigma')$ asserts the possibility of a transition from the state (c,σ) to the state (c',σ').

2.3. Finite Computations

In order to express computation using a small-step semantics, one iterates the transition relation. The '$\to^*$' relation is the reflexive-transitive closure of the transition relation for commands, which covers the set of all finite transition sequences:

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

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

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

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

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

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

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

Figure 2: Small-step semantics for commands

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

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

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 co on the left of rules to indicate that the relation is coinductively defined by that set of rules.[1] We assume that the reader is familiar with the basics of coinduction (for introductions see, e.g., [6; 12; 13]).

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 covers 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 whilenz 1 skip diverges by using $(\mathsf{whilenz}\ 1\ \mathsf{skip}, \cdot) \overset{\infty}{\to}$ as our *coinduction hypothesis*. In the

[1] This notation for coinductively defined relations is a variation of Cousot and Cousot's [5] notation for distinguishing inductively and coinductively (or *positively* and *negatively*) defined relations.

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:

$$\dfrac{\dfrac{(1,\cdot) \Rightarrow_{\mathrm{E}} 1 \quad 1 \neq 0}{(\mathsf{whilenz}\ 1\ \mathsf{skip},\cdot) \to (\mathsf{skip};\mathsf{whilenz}\ 1\ \mathsf{skip},\cdot)} \quad \dfrac{\dfrac{}{(\mathsf{skip};\mathsf{whilenz}\ 1\ \mathsf{skip},\cdot) \to (\mathsf{whilenz}\ 1\ \mathsf{skip},\cdot)} \quad \dfrac{\vdots}{(\mathsf{whilenz}\ 1\ \mathsf{skip},\cdot) \overset{\infty}{\to}}\ \text{CIH}}{(\mathsf{skip};\mathsf{whilenz}\ 1\ \mathsf{skip},\cdot) \overset{\infty}{\to}}}{(\mathsf{whilenz}\ 1\ \mathsf{skip},\cdot) \overset{\infty}{\to}}$$

There also exists an infinite derivation tree whose conclusion is:

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

Proving this is only slightly more involved: we first prove the following straightforward lemma by coinduction:

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

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 [14; 15], 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 [16] 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 [6] 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'. Being inductively defined, the rules cannot express diverging programs.

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

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

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

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

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

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

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

Figure 3: Big-step semantics for commands

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

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

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

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

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 Figure 4 coinductively define a big-step divergence predicate. Like '$\overset{\infty}{\to}$', 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:

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

Here, alloc x and x := 0 converge and the whilenz 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 to duplication of premises between the rules. For example, the set of convergent and divergent 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 $\to^*$. □

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. Each intermediate rule either continues evaluation or propagates 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:

$$\mathit{SemCmd} \ni C ::= c \mid \mathsf{assign2}\ x\ v \mid \mathsf{seq2}\ o\ c \mid \mathsf{ifnz2}\ v\ c\ c \mid \mathsf{whilenz2}\ v\ e\ c \mid \mathsf{whilenz3}\ o\ e\ c$$

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

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 seq2:

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

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 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:[2]

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

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. Crucial is that 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* $(C, \sigma) \Downarrow (o, \sigma')$ *then* $(C, \sigma) \overset{\mathsf{co}}{\Downarrow} (o, \sigma')$.

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 $\mathsf{whilenz}\ 1\ \mathsf{skip}$ coevaluates to anything:

Example 10. $(\mathsf{whilenz}\ 1\ \mathsf{skip}, \cdot) \overset{\mathsf{co}}{\Downarrow} (o, \sigma')$ *for any* o *and* σ'.

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. $(c, \sigma) \Downarrow (o, \sigma')$ *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 $\Downarrow$. 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* $(c, \sigma) \overset{\mathsf{co}}{\Downarrow} (\mathsf{div}, \sigma')$.

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. □

[2]These abort rules allow an arbitrary store σ'. This design decision eases our proofs. Specifically, Lemma 14 does not hold without allowing arbitrary stores in abort rules. We conjecture that stronger abort rules could have been used, but remark that the structure of the store for diverging programs is irrelevant for most practical purposes.

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

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

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

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

$$\mathsf{du}\,\frac{(e, \sigma) \Rightarrow_{\mathrm{E}} v \qquad (\mathsf{ifnz2}\ v\ c_1\ c_2, \sigma) \Downarrow (o, \sigma')}{(\mathsf{ifnz}\ e\ c_1\ c_2, \sigma) \Downarrow (o, \sigma')}\ \text{P-If} \quad \mathsf{du}\,\frac{v \neq 0 \qquad (c_1, \sigma) \Downarrow (o_1, \sigma')}{(\mathsf{ifnz2}\ v\ c_1\ c_2, \sigma) \Downarrow (o_1, \sigma')}\ \text{P-If2} \quad \mathsf{du}\,\frac{(c_2, \sigma) \Downarrow (o_2, \sigma')}{(\mathsf{ifnz2}\ 0\ c_1\ c_2, \sigma) \Downarrow (o_2, \sigma')}\ \text{P-IfZ2}$$

$$\mathsf{du}\,\frac{(e, \sigma) \Rightarrow_{\mathrm{E}} v \qquad (\mathsf{whilenz2}\ v\ e\ c, \sigma) \Downarrow (o, \sigma')}{(\mathsf{whilenz}\ e\ c, \sigma) \Downarrow (o, \sigma')}\ \text{P-While} \qquad \mathsf{du}\,\frac{v \neq 0 \qquad (c, \sigma) \Downarrow (o, \sigma') \qquad (\mathsf{whilenz3}\ o\ e\ c, \sigma') \Downarrow (o', \sigma'')}{(\mathsf{whilenz2}\ v\ e\ c, \sigma) \Downarrow (o', \sigma'')}\ \text{P-While2}$$

$$\mathsf{du}\,\frac{}{(\mathsf{whilenz2}\ 0\ e\ c, \sigma) \Downarrow (\mathsf{conv}, \sigma)}\ \text{P-WhileZ2} \qquad \mathsf{du}\,\frac{(\mathsf{whilenz}\ e\ c, \sigma) \Downarrow (o, \sigma')}{(\mathsf{whilenz3}\ \mathsf{conv}\ e\ c, \sigma) \Downarrow (o, \sigma')}\ \text{P-While3}$$

$$\mathsf{du}\,\frac{}{(\mathsf{seq2}\ \mathsf{div}\ c_2, \sigma) \Downarrow (\mathsf{div}, \sigma')}\ \text{P-Seq-Abort} \quad \mathsf{du}\,\frac{}{(\mathsf{whilenz3}\ \mathsf{div}\ e\ c, \sigma) \Downarrow (\mathsf{div}, \sigma')}\ \text{P-While-Abort}$$

Figure 5: Pretty-big-step semantics for commands

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

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 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 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.

3.4. Trace-Based Coinductive Big-Step Semantics

The approaches seen so far have essentially required us to decide which interpretation to use ahead of evaluating a program: either we use induction in the hopes that the derivation tree is finite, or we try to use coinduction to construct a proof that a program diverges. For proofs, one typically resorts to using the law of excluded middle for doing case analyses on whether a program terminates or not, which brings us outside the realm of constructivity.

Nakata and Uustalu [17] argue that constructive reasoning forces one to be conscious about the principles one depends on in arguments, and that "a need for unexpectedly strong principles is often a sign of some imperfect design choice in the setup of a theory". The idea behind trace-based coinductive big-step semantics [6; 7; 17] is to gradually accumulate a trace of semantic information, such as how the state of the store develops as a program is evaluated. For programs that terminate, the final element of the trace is the store resulting from evaluation. For programs that do not terminate, the trace is infinite. Traces provide enough structure that we can reason about them constructively without assuming whether they are finite or not.

Following Nakata and Uustalu [7], the rules in Figure 6 coinductively define two relations: $\Rightarrow_{\mathrm{NU}}$ is a coinductive big-step relation that produces a trace, whereas $\overset{*}{\Rightarrow}_{\mathrm{NU}}$ is the coinductive prefix-closure of $\Rightarrow_{\mathrm{NU}}$. The rules rely on traces, coinductively defined as follows:

$$\mathit{Trace} \ni \tau ::= \langle\sigma\rangle \mid \sigma :: \tau$$

The $\overset{*}{\Rightarrow}_{\mathrm{NU}}$ relation is responsible for 'peeling off' prefixes of a trace, so as to look for the tail of the trace, which contains the most recently added store, if it exists (e.g., if the trace is finite). If, for example, the first branch of T-Seq diverges, τ will be infinite and the second premise of the rule will be an infinite branch of applications of the T-Peel rule. This effectively inhibits further evaluation.

A pleasant property of coinductive trace-based big-step semantics is that it allows us to prove that our language is deterministic, even for diverging programs. Formally, what we prove is that all traces that start from the same state are *bisimilar*, where bisimilarity '$\approx$' is defined as follows:

$$\mathsf{co}\frac{}{\langle\sigma\rangle \approx \langle\sigma\rangle}\,\text{Bisim1} \qquad \mathsf{co}\frac{\tau \approx \tau'}{\sigma :: \tau \approx \sigma :: \tau'}\,\text{Bisim2}$$

Following Nakata and Uustalu [7; 17] we can prove that the language is deterministic.

Proposition 16. *If* $(c,\sigma) \Rightarrow_{\mathrm{NU}} \tau$ *and* $(c,\sigma) \Rightarrow_{\mathrm{NU}} \tau'$ *then* $\tau \approx \tau'$.

Proof. The proof is by structural induction on c. The cases for the T-Seq and T-While rules rely on lemmas using coinduction to prove that the conclusion follows from the given assumptions. □

$\boxed{(c,\sigma) \Rightarrow_{\text{NU}} \tau}$ $\boxed{(c,\tau) \overset{*}{\Rightarrow}_{\text{NU}} \tau}$

$$\textsf{co}\frac{}{(\textsf{skip},\sigma) \Rightarrow_{\text{NU}} \langle\sigma\rangle}\ \text{T-Skip} \qquad \textsf{co}\frac{x \notin \text{dom}(\sigma)}{(\textsf{alloc}\ x,\sigma) \Rightarrow_{\text{NU}} \sigma :: \langle\sigma[x \mapsto \textsf{null}]\rangle}\ \text{T-Alloc}$$

$$\textsf{co}\frac{(e,\sigma) \Rightarrow_{\text{E}} v \quad x \in \text{dom}(\sigma)}{(x := e,\sigma) \Rightarrow_{\text{NU}} \sigma :: \langle\sigma[x \mapsto v]\rangle}\ \text{T-Assign} \qquad \textsf{co}\frac{(c_1,\sigma) \Rightarrow_{\text{NU}} \tau \quad (c_2,\tau) \overset{*}{\Rightarrow}_{\text{NU}} \tau'}{(c_1; c_2,\sigma) \Rightarrow_{\text{NU}} \tau'}\ \text{T-Seq}$$

$$\textsf{co}\frac{(e,\sigma) \Rightarrow_{\text{E}} v \quad v \neq 0 \quad (c_1,\sigma :: \langle\sigma\rangle) \overset{*}{\Rightarrow}_{\text{NU}} \tau}{(\textsf{ifnz}\ e\ c_1\ c_2,\sigma) \Rightarrow_{\text{NU}} \tau}\ \text{T-If} \qquad \textsf{co}\frac{(e,\sigma) \Rightarrow_{\text{E}} 0 \quad (c_2,\sigma :: \langle\sigma\rangle) \overset{*}{\Rightarrow}_{\text{NU}} \tau}{(\textsf{ifnz}\ e\ c_1\ c_2,\sigma) \Rightarrow_{\text{NU}} \tau}\ \text{T-IfZ}$$

$$\textsf{co}\frac{(e,\sigma) \Rightarrow_{\text{E}} v \quad v \neq 0 \quad (c,\sigma :: \langle\sigma\rangle) \overset{*}{\Rightarrow}_{\text{NU}} \tau \quad (\textsf{whilenz}\ e\ c,\tau) \overset{*}{\Rightarrow}_{\text{NU}} \tau'}{(\textsf{whilenz}\ e\ c,\sigma) \Rightarrow_{\text{NU}} \tau'}\ \text{T-While}$$

$$\textsf{co}\frac{(e,\sigma) \Rightarrow_{\text{E}} 0}{(\textsf{whilenz}\ e\ c,\sigma) \Rightarrow_{\text{NU}} \sigma :: \langle\sigma\rangle}\ \text{T-WhileZ}$$

$$\textsf{co}\frac{(c,\sigma) \Rightarrow_{\text{NU}} \tau}{(c,\langle\sigma\rangle) \overset{*}{\Rightarrow}_{\text{NU}} \tau}\ \text{T-Run} \qquad \textsf{co}\frac{(c,\tau) \overset{*}{\Rightarrow}_{\text{NU}} \tau'}{(c,\sigma :: \tau) \overset{*}{\Rightarrow}_{\text{NU}} \sigma :: \tau'}\ \text{T-Peel}$$

Figure 6: Coinductive trace-based big-step semantics for commands

The following theorem states that the trace-based semantics $\Rightarrow_{\text{NU}}$ coincides with the big-step semantics given by $\Rightarrow_{\text{B}}$ and $\overset{\infty}{\Rightarrow}$.

Theorem 17. $(c,\sigma) \Rightarrow_{\text{NU}} \tau$ *iff* $(\exists\sigma'.\ (c,\sigma) \Rightarrow_{\text{B}} \sigma') \ \vee\ (c,\sigma) \overset{\infty}{\Rightarrow}$.

Proof (classical). The proof relies on a functional encoding of the trace-based semantics in order to reason existentially about traces. Using $f \in \mathit{Cmd} \to \mathit{Store} \to \mathit{Trace}$ to denote the functional encoding, and $\mathit{last}(\tau,\sigma)$ to denote the inductively defined relation checking that the final store in a trace τ is σ, the proof is split into four lemmas:

- $(c,\sigma) \Rightarrow_{\text{NU}} \tau$ and τ is finite, then $\exists\sigma', (c,\sigma) \Rightarrow_{\text{B}} \sigma' \ \wedge\ \mathit{last}(\tau,\sigma')$;
- $(c,\sigma) \Rightarrow_{\text{B}} \sigma'$ implies $(c,\sigma) \Rightarrow_{\text{NU}} (f\ c\ \sigma)$;
- $(c,\sigma) \Rightarrow_{\text{NU}} \tau$ and τ is infinite, then $(c,\sigma) \overset{\infty}{\Rightarrow}$; and
- $(c,\sigma) \overset{\infty}{\Rightarrow}$ implies $(c,\sigma) \Rightarrow_{\text{NU}} (f\ c\ \sigma)$.

Using these lemmas, the big-to-trace direction follows straightforwardly. The trace-to-big direction uses the law of excluded middle for case analysis on the finiteness of traces. □

Coinductive trace-based semantics adds structure to a relation that is useful for proving properties about diverging programs. The extra structure is, however, unnecessary when all we require is a simple distinction between programs that diverge or converge. Next, we present a lightweight and generic approach to representing divergence.

$$\mathsf{du}\frac{}{(c,\sigma,\downarrow) \to^\star (c,\sigma,\downarrow)}\ \text{G-Refl} \qquad \mathsf{du}\frac{(c,\sigma) \to (c',\sigma') \qquad (c',\sigma',\downarrow) \to^\star (c'',\sigma'',\delta)}{(c,\sigma,\downarrow) \to^\star (c'',\sigma'',\delta)}\ \text{G-Trans}$$

Figure 7: Reflexive-transitive closure with generic divergence

4. A Generic Approach to Divergence

The approaches surveyed so far have required us to introduce new relations (e.g., big-step divergence predicates in Sect. 3.2, and the prefix-closure of the trace-based relation in Sect. 3.4), or introduce new rules for existing constructs (e.g., pretty-big-step semantic constructors in Sect. 3.3). In this section we present a novel approach to representing divergence that is both lightweight and generic. It is lightweight in 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. The approach works equally well for small-step and big-step semantics.

4.1. Small-Step Semantics

The idea is to use a flag to indicate divergence:

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

Here, $\downarrow$ indicates convergence and $\uparrow$ divergence.

We show how to extend the inductively defined reflexive-transitive closure $\to^*$ from Sect. 2.3 to express both divergence and convergence. The extension involves simply threading an auxiliary entity ranging over the *Div* flag through the conclusion and premises. As with pretty-big-step semantics, we give the resulting closure a *dual* interpretation (i.e., it defines both an inductive and coinductive relation). Figure 7 summarises the resulting rules for $\to^\star$. This simple extension suffices to prove and reason about divergence without a separate $\overset{\infty}{\to}$ relation. 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 '$\uparrow$'.

Proposition 18. $(c,\sigma,\downarrow) \to^\star (c',\sigma',\delta)$ *implies* $\delta \neq \uparrow$.

Proof. By induction on $\to^\star$. □

Theorem 19 proves that the inductive interpretation of $\to^\star$ corresponds exactly to the standard reflexive-transitive closure.

Theorem 19. $(c,\sigma,\downarrow) \to^\star (c',\sigma',\downarrow)$ *iff* $(c,\sigma) \to^* (c',\sigma')$

Proof. Each direction is proven by straightforward induction on $\to^\star$ and $\to^*$. □

Analogously, Theorem 20 proves that the coinductive interpretation is able to express divergence on a par with the traditional small-step divergence predicate from Sect. 2.4.

Theorem 20. $(c,\sigma,\downarrow) \overset{\mathsf{co}}{\to}^\star (c',\sigma',\uparrow)$ *iff* $(c,\sigma) \overset{\infty}{\to}$.

Proof. Each direction is proven by straightforward rule coinduction on $\overset{\mathsf{co}}{\to}^\star$ and $\overset{\infty}{\to}$. □

In summary, generic divergence allows us to use a single set of rules with a dual interpretation to represent both convergence and divergence.

4.2. Big-Step Semantics

We show how augmenting the standard inductive big-step rules from Figures 1 and 3 with divergence flags allows us to express and reason about divergence on a par with traditional big-step divergence predicates. Threading divergence flags through the conclusion and premises of the standard big-step rules in left-to-right order gives the rules in Figure 8. In all rules, the divergence flag is threaded through rules such that the conclusion source always starts in a $\downarrow$ state. Since expressions cannot diverge, their definition is the same as in Figure 1.

In the G-Seq rule in Figure 8, 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 the *divergence rule* G-Div, also in Figure 8. The divergence rule serves the same purpose as abort rules in pretty-big-step semantics: it propagates divergence as it arises and inhibits further evaluation. The divergence rule allows evaluation to return an arbitrary store. This does not weaken the expressiveness as compared with small-step or big-step divergence predicates, as we shortly prove in Theorem 22. We consider the converging case first.

Theorem 21 proves that adding divergence flags and the divergence rule does not change the inductive meaning of the standard inductive big-step relation.

Theorem 21. $(c, \sigma) \Rightarrow_{\mathrm{B}} \sigma'$ *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 24 (given below). □

Theorem 22 proves that adding divergence flags allows us to prove divergence on a par with traditional big-step divergence predicates.

Theorem 22. $(c, \sigma) \overset{\infty}{\Rightarrow}$ *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 23, Lemma 25, and the law of excluded middle for case analysis on $\Rightarrow_{\mathrm{G}}$. □

Lemma 23. *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 $\Rightarrow_{\mathrm{G}}$. □

Lemma 24 proves that we cannot use the inductively defined relation to prove divergence.

Lemma 24. $(c, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} (\sigma', \delta)$ *implies* $\delta \neq \uparrow$.

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

Since the rules have a dual interpretation, the coinductive interpretation subsumes the inductive interpretation, as proven in Lemma 25.

Lemma 25. *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 $\Rightarrow_{\mathrm{G}}$. □

However, 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 whilenz 1 skip coevaluates to anything:

Example 26. $(\mathsf{whilenz}\ 1\ \mathsf{skip}, \cdot, \downarrow) \overset{\mathsf{co}}{\Rightarrow}_{\mathrm{G}} (\sigma', \delta)$ *for any* σ' *and* δ.

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

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

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

$$\mathsf{du}\frac{(e, \sigma, \downarrow) \Rightarrow_{\mathrm{E}} v \qquad v \neq 0 \quad (c_1, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} (\sigma', \delta)}{(\mathsf{ifnz}\ e\ c_1\ c_2, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} (\sigma', \delta)}\ \text{G-If} \qquad \mathsf{du}\frac{(e, \sigma, \downarrow) \Rightarrow_{\mathrm{E}} 0 \qquad (c_2, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} (\sigma', \delta)}{(\mathsf{ifnz}\ e\ c_1\ c_2, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} (\sigma', \delta)}\ \text{G-IfZ}$$

$$\mathsf{du}\frac{(e, \sigma, \downarrow) \Rightarrow_{\mathrm{E}} v \qquad v \neq 0 \quad (c, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} (\sigma', \delta) \quad (\mathsf{whilenz}\ e\ c, \sigma', \delta) \Rightarrow_{\mathrm{G}} (\sigma'', \delta')}{(\mathsf{whilenz}\ e\ c, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} (\sigma'', \delta')}\ \text{G-While} \qquad \mathsf{du}\frac{(e, \sigma, \downarrow) \Rightarrow_{\mathrm{E}} 0}{(\mathsf{whilenz}\ e\ c, \sigma, \downarrow) \Rightarrow_{\mathrm{G}} (\sigma, \downarrow)}\ \text{G-WhileZ}$$

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

Figure 8: Big-step semantics for commands and expressions with generic divergence

Proof. Straightforward proof by coinduction. □

Comparing generic divergence (Figure 8) 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. Our approach suffices to express divergence on a par with the approaches considered in Sects. 2 and 3. Using rules with generic divergence in proofs is no more involved than traditional approaches either, as illustrated by examples in our Coq proofs available online at: `http://www.plancomps.org/jlamp2015/`.

4.3. Divergence Rules are Necessary

From Example 26 we know that some diverging programs result in a ↓ state. A natural question is: do we really need the ↑ flag and divergence rule? Here, we answer this question affirmatively. Consider the following program:

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

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 type error: 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:

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

Example 27. *For any* σ, $((\mathsf{whilenz}\ 1\ \mathsf{skip}); \mathsf{alloc}\ \mathrm{x}; \mathrm{x} := +(\mathrm{x}, 0), \cdot, \downarrow) \overset{\mathsf{co}}{\Rightarrow} (\sigma, \uparrow)$. *In contrast,* $\neg\ \exists\sigma.\ ((\mathsf{whilenz}\ 1\ \mathsf{skip}); \mathsf{alloc}\ \mathrm{x}; \mathrm{x} := +(\mathrm{x}, 0), \cdot, \downarrow) \overset{\mathsf{co}}{\Rightarrow} (\sigma, \downarrow)$

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

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

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

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

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

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

Figure 9: Big-step I-MSOS for expressions and commands with implicit propagation

5. Implicit Generic Divergence

The previous section presented a novel encoding of divergence. In this section, we first recall how I-MSOS 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 rule for sequential composition:

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

This rule threads a store from conclusion source through the premises to the conclusion target. Adding a new auxiliary entity, such as a divergence 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:

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

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

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 and in the rest of this article, we let I-MSOS rules thread auxiliary entities through premises in left-to-right order. Thus, 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 9 correspond to the inductive rules in Figure 3. In order to express divergence generically, 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.

  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 9 gives the following SOS rule:

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

  Considering another example, applying the same judgment signature to the I-Skip axiom from Figure 9 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.

- *Right-hand side default values:* a non-variable occurring in a highlighted position on the right-hand side of an arrow in an I-MSOS signature denotes a default value for *axioms*. Unless another value is explicitly given in an axiom, the right-hand side auxiliary entity matches the value. For non-axioms, the right-hand side is a variable that has been threaded through premises following the standard convention for propagation between premises.

  For example, in the signature judgment $(c,\sigma\ ,\langle\sigma\rangle\ )\ \Rightarrow\ (\ \sigma'\ ,\sigma :: \langle\sigma'\rangle)$, '$\sigma :: \langle\sigma'\rangle$' is a non-variable. Applying this signature judgment as described above to the I-Seq I-MSOS rule from Figure 9 gives the following SOS rule:

$$\frac{(c_1,\sigma,\langle\sigma\rangle)\Rightarrow_{\mathrm{G}}(\sigma',\tau') \qquad (c_2,\sigma',\tau)\Rightarrow_{\mathrm{G}}(\sigma'',\tau')}{(c_1;c_2,\sigma,\langle\sigma\rangle)\Rightarrow_{\mathrm{G}}(\sigma'',\tau')}$$

  Considering another example, applying the same judgment signature to the I-Alloc rule from Figure 9 gives the rule:

$$\frac{x\notin\mathrm{dom}(\sigma)}{(\mathsf{skip},\sigma,\langle\sigma\rangle)\Rightarrow(\sigma[x\mapsto\mathsf{null}],\sigma :: \langle\sigma[x\mapsto\mathsf{null}]\rangle)}$$

- *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 9:

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

We also augment the rules in Figure 9 by the divergence rule from Sect. 4.2:

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

The resulting set of rules are identical to those in Figure 8. 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.

6. Implicit Trace-Based Semantics

We show how to transform I-MSOS big-step rules using the I-MSOS transformations introduced in previous section to obtain trace-based semantics à la Nakata and Uustalu [7]. We also illustrate how this allows us to reason about properties of infinite programs in our example language.

6.1. Stateful Traces as State

Using I-MSOS with the extension introduced in previous section, giving a trace-based semantics is a mere matter of giving the right signature for the rules in Figure 9. Using the same notion of trace as in Sect. 3.4, we give the following signature to the I-MSOS rules in Figure 9:

$$(c, \sigma, \langle\sigma\rangle) \overset{\mathsf{co}}{\Rightarrow}_{I\ T} (\sigma', \sigma :: \langle\sigma'\rangle)$$

We also add a rule for peeling off prefixes:

$$\mathsf{co}\frac{(c, \sigma, \tau) \Rightarrow_{\text{IT}} (\sigma', \tau')}{(c, \sigma, \sigma_0 :: \tau) \Rightarrow_{\text{IT}} (\sigma', \sigma_0 :: \tau')}\ \text{IT-Peel}$$

The resulting rules are summarised in Figure 10. The following propositions state the correspondence between the automatically generated rules in Fig. 10 and the trace-based rules à la Nakata and Uustalu given in Figure 6.

Theorem 28. *Using $f \in Cmd \to Store \to Trace$ to denote the functional encoding of $\Rightarrow_{\text{NU}}$, and $g \in Cmd \to Store \to Trace$ to denote the functional encoding of $\Rightarrow_{\text{IT}}$, it holds that $(c, s) \Rightarrow_{\text{NU}} (f\ c\ s)$ iff $(c, s, \langle s\rangle) \Rightarrow_{\text{IT}} (\sigma', (g\ c\ s))$.*

Proof. We relate $\Rightarrow_{\text{IT}}$ to standard big-step semantics following the proof structure outlined in Theorem 17, from which our goal follows as a corollary. □

6.2. Trace-Based Definite Assignment

Trace-based semantics are useful for reasoning about program properties of diverging programs. Variables that have been allocated but not assigned a value are mapped to the special null value in stores. This gives rise to potential undefined behaviour. For example, $(+(x, 1), \sigma[x \mapsto \mathsf{null}])$ does not have a well-defined semantics using our rules. *Definite assignment* is a program property that rules out this possibility: it says that only assigned variables are read. This property is not straightforwardly expressible for diverging programs using traditional divergence predicates or generic divergence. It is, however, expressible using trace-based semantics.

To express definite assignment, we define traces coinductively to record sets of variables:

$$R, A \subseteq Var \qquad RATrace \ni T \ ::= \ \langle (R, A)\rangle \ \mid \ (R, A) :: T$$

The idea is that we accumulate information about which variables have definitely been assigned to (denoted by the set A), and which variables have been read (denoted by the set R). Using this notion of trace, we give the following signature to the rules in Figure 9:

$$(c, \langle (R, A)\rangle) \overset{\mathsf{co}}{\Rightarrow}_{\text{I A}} (R, A) :: T \qquad (e, \langle (R, A)\rangle) \Rightarrow_{\text{IE A}} (v, \langle (R', A)\rangle)$$

We also replace the rule for reading variables by one that records that the variable has been read:

$$\frac{x \in \text{dom}(\sigma)}{(x, \sigma, \langle (R, A)\rangle) \Rightarrow_{\text{IEA}} (\sigma(x), \langle (R \cup \{x\}, A)\rangle}\ \text{IEA-Var}$$

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

$$\textsf{co}\frac{}{(\textsf{skip}, \sigma, \langle\sigma\rangle) \Rightarrow_{\text{IT}} (\sigma, \sigma :: \langle\sigma\rangle)}\ \text{IT-Skip} \qquad \textsf{co}\frac{x \notin \text{dom}(\sigma) \qquad \sigma' = \sigma[x \mapsto \textsf{null}]}{(\textsf{alloc}\ x, \sigma, \langle\sigma\rangle) \Rightarrow_{\text{IT}} (\sigma', \sigma :: \langle\sigma'\rangle)}\ \text{IT-Alloc}$$

$$\textsf{co}\frac{x \in \text{dom}(\sigma) \qquad (e, \sigma) \Rightarrow_{\text{E}} v \qquad \sigma' = \sigma[x \mapsto v]}{(x := e, \sigma, \langle\sigma\rangle) \Rightarrow_{\text{IT}} (\sigma', \sigma :: \langle\sigma'\rangle)}\ \text{IT-Assign} \qquad \textsf{co}\frac{(c_1, \sigma, \langle\sigma\rangle) \Rightarrow_{\text{IT}} (\sigma', \tau) \qquad (c_2, \sigma', \tau) \Rightarrow_{\text{IT}} (\sigma'', \tau')}{(c_1; c_2, \sigma, \langle\sigma\rangle) \Rightarrow_{\text{IT}} (\sigma'', \tau')}\ \text{IT-Seq}$$

$$\textsf{co}\frac{(e, \sigma) \Rightarrow_{\text{E}} v \qquad v \neq 0 \qquad (c_1, \sigma, \langle\sigma\rangle) \Rightarrow_{\text{IT}} (\sigma', \tau)}{(\textsf{ifnz}\ e\ c_1\ c_2, \sigma, \langle\sigma\rangle) \Rightarrow_{\text{IT}} (\sigma', \tau)}\ \text{IT-If} \qquad \textsf{co}\frac{(e, \sigma) \Rightarrow_{\text{E}} 0 \qquad (c_2, \sigma, \langle\sigma\rangle) \Rightarrow_{\text{IT}} (\sigma', \tau)}{(\textsf{ifnz}\ e\ c_1\ c_2, \sigma, \langle\sigma\rangle) \Rightarrow_{\text{IT}} (\sigma', \tau)}\ \text{IT-IfZ}$$

$$\textsf{co}\frac{(e, \sigma) \Rightarrow_{\text{E}} v \qquad v \neq 0 \qquad (c, \sigma, \langle\sigma\rangle) \Rightarrow_{\text{IT}} (\sigma', \tau) \qquad (\textsf{whilenz}\ e\ c, \sigma', \tau) \Rightarrow_{\text{IT}} (\sigma'', \tau')}{(\textsf{whilenz}\ e\ c, \sigma, \langle\sigma\rangle) \Rightarrow_{\text{IT}} (\sigma'', \tau')}\ \text{IT-While} \qquad \textsf{co}\frac{(e, \sigma) \Rightarrow_{\text{IT}} 0}{(\textsf{whilenz}\ e\ c, \sigma, \langle\sigma\rangle) \Rightarrow_{\text{IT}} (\sigma, \sigma :: \langle\sigma\rangle)}\ \text{IT-WhileZ}$$

$$\textsf{co}\frac{(c, \sigma, \tau) \Rightarrow_{\text{IT}} (\sigma', \tau')}{(c, \sigma, \sigma_0 :: \tau) \Rightarrow_{\text{IT}} (\sigma', \sigma_0 :: \tau')}\ \text{IT-Peel}$$

Figure 10: I-MSOS-generated coinductive big-step trace-based SOS for commands

Similarly, we replace the rule for assignment by one that records which variables have been assigned:

$$\frac{x \in \text{dom}(\sigma) \qquad (e, \sigma, \langle(R, A)\rangle) \Rightarrow_{\text{IEA}} (v, \langle(R', A)\rangle}{(x := e, \sigma, \langle(R, A)\rangle) \Rightarrow_{\text{IA}} (\sigma[x \mapsto v], (R, A) :: (R', A) :: \langle(R', A \cup \{x\})\rangle)}\ \text{IA-Assign}$$

Finally, we add a rule for peeling off traces:

$$\frac{(c, T) \Rightarrow_{\text{IA}} T'}{(c, (R, A) :: T) \Rightarrow_{\text{IA}} (R, A) :: T'}\ \text{IA-Peel}$$

The resulting rules are summarised in Appendix A. We can now define what it means for a trace to satisfy definite assignment by the predicate $\textsf{defasn}$ for traces:

$$\textsf{co}\frac{R \subseteq A \qquad \textsf{defasn}\, T}{\textsf{defasn}\, ((R, A) :: T)}\ \text{DefAsn1} \qquad \textsf{co}\frac{R \subseteq A}{\textsf{defasn}\, \langle(R, A)\rangle}\ \text{DefAsn2}$$

The set of programs that satisfy definite assignment are those that produce traces satisfying this definition of definite assignment. This set of programs is a proper subset of the set of programs that do not go wrong. For example, $\textsf{alloc}\ \text{x}; \textsf{alloc}\ \text{y}; \text{x} := \text{y}$ does not satisfy definite assignment, but does not go wrong either: the program reads the variable y before it is assigned, but produces the store $\{\text{x} \mapsto \textsf{null}, \text{y} \mapsto \textsf{null}\}$.

Trace-based definite assignment requires us to first construct a possibly-infinite trace and subsequently prove that the trace satisfies definite assignment. For practical purposes, it is convenient to determine that a program satisfies this property *statically*, that is without running the program. In next section we show how to *calculate* a static definite assignment analysis for our while-language.

7. Static Definite Assignment

In earlier sections we showed how to solve problems typically associated with big-step semantics. In this section, we motivate the usefulness of big-step semantics by comparing big-step and small-step proofs. We

do so by considering the problem of *calculating* safe analyses from big-step semantics. Inspired by Hutton and Bahr's work on calculating correct compilers [11], we use the proof of safety as a guiding principle for calculating the analysis (by so-called *constructive induction* [18]).

7.1. Definite Assignment as Abrupt Termination

In previous section we saw how to definite assignment could be defined by means of possibly-infinite traces. For calculating analyses, it is convenient to work with an inductively defined relation. We instrument our evaluation relation to make definite assignment a property that we can reason about inductively.

An equivalent way of expressing definite assignment is to make all programs terminate abruptly when they do *not* satisfy definite assignment. Instead of using traces, we introduce an exception flag:

$$\mathit{Except} \ni \varepsilon ::= \mathsf{ok} \mid \mathsf{err}$$

Using $A \subseteq \mathit{Var}$ for sets of variables that have definitely been assigned, we give the following signature to the rules from Figure 9:

$$\boxed{(\ c\ , A) \overset{\mathsf{du}}{\Rightarrow}_{\mathrm{I\ B}} A'} \quad \boxed{(\ c\ , \mathsf{ok}) \overset{\mathsf{du}}{\Rightarrow}_{\mathrm{I\ B}} \varepsilon} \qquad \boxed{(\ e\ , A) \Rightarrow_{\mathrm{IE\ B}} v} \quad \boxed{(\ e\ , \mathsf{ok}) \Rightarrow_{\mathrm{IE\ B}} (\ v\ , \varepsilon)}$$

We also replace the rule for reading variables by two I-MSOS rules:

$$\frac{x \in \mathrm{dom}(\sigma) \qquad x \in A}{(x, \sigma, A) \Rightarrow_{\mathrm{IEB}} \sigma(x)}\ \text{IEB-Var} \qquad \frac{x \in \mathrm{dom}(\sigma) \qquad x \notin A}{(x, \sigma, A, \mathsf{ok}) \Rightarrow_{\mathrm{IEB}} (v, \mathsf{err})}\ \text{IEB-Var-Err}$$

Here, IEB-Var-Err errs if we attempt to read from a variable that has not definitely been assigned. Similarly, we replace the rule for assignment by:

$$\frac{x \in \mathrm{dom}(\sigma) \qquad (e, \sigma, A) \Rightarrow_{\mathrm{IEB}} v}{(x := e, \sigma, A) \Rightarrow_{\mathrm{IB}} (\sigma[x \mapsto v], A' \cup \{x\})}\ \text{IB-Assign}$$

Finally, in order to propagate the abrupt termination, we add exception rules, similar to the divergence rules from Sect. 4:

$$\mathsf{du}\,\frac{}{(c, \sigma, A, \mathsf{err}) \Rightarrow_{\mathrm{IB}} (\sigma', A', \mathsf{err})}\ \text{IB-Err} \qquad \frac{}{(e, \sigma, A, \mathsf{err}) \Rightarrow_{\mathrm{IEB}} (v, \mathsf{err})}\ \text{IEB-Err}$$

The resulting SOS rules are summarised in Appendix B. Now, the programs that satisfy definite assignment are those for which $(c, \sigma, A, \mathsf{ok}) \Rightarrow_{\mathrm{IB}} (\sigma', A', \varepsilon)$ implies that $\varepsilon \neq \mathsf{err}$. In other words, it is the set of programs which, if they terminate, do not terminate abruptly.

7.2. Calculating Static Definite Assignment

Definite assignment as abrupt termination is convenient as a guiding principle for calculating a static definite assignment analysis. The analysis we are looking for is one which tracks the set of variables that has definitely been assigned. Our analysis will use judgments of the form $(c, A) \triangleright A'$ to assert that, given a command c and a set of variables A that have definitely been assigned, evaluating command c does not entail reading any variable that has not been assigned. We want the following theorem to hold:

Theorem 29. *If* $(c, A) \triangleright A'$ *and* $(c, \sigma, A, \mathrm{OK}) \Rightarrow_{\mathrm{IB}} (\sigma', A'', \varepsilon)$*, then* $\varepsilon = \mathrm{OK}$

We calculate definite assignment using this property as our guiding principle. The idea is to do the proof by induction on $\Rightarrow_{\mathrm{IB}}$, and use the proof as a guide for inferring a set of rules for $\triangleright$ that is safe-by-construction. Using the proof as a guide allows us to gradually infer which properties $\triangleright$ and $\Rightarrow_{\mathrm{IB}}$ should satisfy in order to complete the proof. The proof sketch and lemmas given below were used to infer the rules given in Figure 11.

$$\mathit{CmdOrExpr} \ni b ::= c \mid e \qquad \boxed{(b, A) \triangleright A}$$

$$\frac{}{(\mathsf{skip}, A) \triangleright A}\ \text{A-Skip} \qquad \frac{}{(\mathsf{alloc}\ x, A) \triangleright A}\ \text{A-Alloc} \qquad \frac{(e, A) \triangleright A}{(x := e, A) \triangleright A \cup \{x\}}\ \text{A-Assign}$$

$$\frac{(c_1, A) \triangleright A_1 \qquad (c_2, A_1) \triangleright A_2}{(c_1; c_2, A) \triangleright A_2}\ \text{A-Seq} \qquad \frac{(e, A) \triangleright A \qquad (c_1, A) \triangleright A_1 \qquad (c_2, A) \triangleright A_2}{(\mathsf{ifnz}\ e\ c_1\ c_2, A) \triangleright A}\ \text{A-If}$$

$$\frac{(e, A) \triangleright A \qquad (c, A) \triangleright A'}{(\mathsf{whilenz}\ e\ c, A) \triangleright A}\ \text{A-While} \qquad \frac{}{(v, A) \triangleright A}\ \text{A-Val} \qquad \frac{x \in A}{(x, A) \triangleright A}\ \text{A-Var}$$

$$\frac{(e_1, A) \triangleright A \qquad (e_2, A) \triangleright A}{(\mathit{bop}(e_1, e_2), A) \triangleright A}\ \text{A-Bop}$$

Figure 11: Static definite assignment

Proof of Theorem 29. By induction on $\Rightarrow_{\text{IB}}$. The proof relies on three lemmas: one saying that the size of the set A increases monotonically under $\Rightarrow_{\text{IB}}$ (Lemma 30); one saying that weakening (adding more variables to) the assumptions about which variables have been assigned does not inhibit the analysis (Lemma 31); and a preservation lemma which says that the static analysis safely under-approximates the actual set of assigned variables (Lemma 32). □

Lemma 30. $(c, \sigma, A, \text{OK}) \Rightarrow_{\text{IB}} (\sigma', A', \varepsilon)$ *implies* $A \subseteq A'$.

Proof. Straightforward proof by induction on $\Rightarrow_{\text{IB}}$. □

Lemma 31. *If* $(c, A) \triangleright A'$ *and* $A \subseteq A_0$*, then* $\exists A'_0.\ (c, A_0) \triangleright A'_0$.

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

Lemma 32. *If* $(c, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IB}} (\sigma', A', \varepsilon)$ *and* $(c, A) \triangleright A''$ *then* $A'' \subseteq A'$.

Proof. Straightforward proof by induction on $\Rightarrow_{\text{IB}}$. □

The static analysis in Figure 11 is naïve and could be improved. For example, A-If can safely be replaced by the following rule which takes the intersection of the inferred assigned variables of each branch of a conditional:

$$\frac{(e, A) \triangleright A \qquad (c_1, A) \triangleright A_1 \qquad (c_2, A) \triangleright A_2}{(\mathsf{ifnz}\ e\ c_1\ c_2, A) \triangleright (A_1 \cap A_2)}\ \text{A-If}'$$

Using Theorem 29 as a guiding principle, it is easy to construct trivial analyses that are safe by construction: e.g., simply reject all programs. The art is to find analyses that accept more programs. Here, we are mainly concerned with comparing big-step and small-step proofs, and the calculated naïve analysis in Figure 11 suffices for this purpose.

7.3. Definite Assignment Safety using Small-Step

Similar to the guiding principle given in previous section, we can devise a statement that allows us to use the same approach to calculate static definite assignment. We extend our small-step semantics to accumulate a set of assigned variables and abruptly terminate when a variable is read that is not in the set of assigned variables, following the approach described in Sect. 7.1. The rules are summarised in Appendix C. For this small-step relation, we use Theorem 33 below as a guiding principle. We summarise its proof along with lemmas that the proof relies on:

Theorem 33. *If* $(c, A) \triangleright A'$ *and* $(c, \sigma, A, \text{OK}) \rightarrow_{\text{SB}}^{*} (\sigma', A'', \varepsilon)$*, then* $\varepsilon = \text{OK}$

Proof of Theorem 33. By induction on $\rightarrow_{\text{SB}}^{*}$ using Lemma 34 and Lemma 35. □

Lemma 34. *If* $(c, \sigma, A, \mathsf{ok}) \rightarrow_{\text{SB}} (c', \sigma', A', \varepsilon')$ *and* $(c, A) \triangleright A''$*, then* $\exists A'''.\ (c', A') \triangleright A''' \ \wedge \ A'' \subseteq A'''$.

Proof. By induction on $\rightarrow_{\text{SB}}$. It requires lemmas about weakening (Lemma 31) and the monotonic increase of the set of assigned variables under $\triangleright$ (Lemma 36). □

Lemma 35. *If* $(c, A) \triangleright A'$ *and* $(c, \sigma, A, \text{OK}) \rightarrow_{\text{SB}} (c', \sigma', A'', \varepsilon)$*, then* $\varepsilon = \text{OK}$

Proof. Straightforward proof by induction on $\rightarrow_{\text{SB}}$. □

Lemma 36. $(c, A) \triangleright A'$ *implies* $A \subseteq A'$

Proof. Proof by induction on $\triangleright$. □

7.4. Comparing Big-Step and Small-Step

The big-step proof allows us to prove the desired property directly. It uses fewer lemmas and fewer cases than the small-step proof. The small-step proof works equally well as a guiding principle, but requires more lemmas, and the preservation lemma (Lemma 34) is slightly more involved than the big-step proof.

8. Related Work

In their paper introducing the traditional approach to representing divergence in big-step semantics via coinductive divergence predicates, Cousot and Cousot [5, p. 90] remark: "The quest for a unique general-purpose semantics for programming languages has failed. A better approach is to establish correspondences between various semantics at different levels of abstraction." In that paper, they start from a coinductive trace-based big-step SOS and use abstract interpretation [19] to derive standard big-step rules, divergence predicates, and small-step semantics. We address the same issue identified by the Cousots, but in a different way: we start from an inductive big-step semantics and show how to automatically transform it to obtain novel variants that are of equal expressiveness as divergence predicates and trace-based semantics.

Several papers have explored how to represent divergence in big-step semantics. Leroy and Grall [6] 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 propose a more elegant approach to accumulating traces based on the 'peel' rules, as recalled in Sect. 3.4. Subsequent work by Nakata and Uustalu has shown how this approach scales to coinductive big-step semantics for *resumptions* [17] (denoting the external behaviour of a communicating agent in concurrency theory [20]) as well as interleaving and concurrency [21].

Big-step semantics is close in spirit to denotational semantics. Traditionally, divergence in denotational semantics is represented as the greatest fixed-point of a semantic domain [22]. Several monads have also been suggested for more refined notions. One such example is the *partiality monad* [23; 24] (also known as the *delay monad* [25]). In the partiality monad, functions either return a finitely delayed result or an infinite trace of delays. Piróg and Gibbons [26] study the category theoretic foundations of the resumption monad. Related to the resumption monad is the interactive I/O monad by Hancock and Setzer [27] for modeling the behaviour of possibly-diverging interactive programs.

Moggi [28] suggested monads as a means to modularity in denotational semantics [29]. In a similar vein, Modular SOS [30] provides a means to modularity in operational semantics. The approach to generic divergence presented here is a variant of the abrupt termination technique used in [30]. 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 [31] observed that encoding abrupt termination as a stateful flag instead scales better to big-step rules by avoiding such explicit handlers. Here, we have extended that approach to divergence and trace-based semantics using I-MSOS.

The question of which semantic style 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 [6] 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 [32] and Hutton and Bahr [11] also use big-step semantics for their compiler correctness proofs. However, in the certified C compiler *CompCert*, Leroy [33] uses a small-step semantics and sophisticated notions of bisimulation for its compiler correctness proofs.

In their paper [34] 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 [35] 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 [6] 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. (This essentially corresponds to the approach we took to proving the safety of definite assignment in Sect. 7.) A non-exhaustive list of examples that use explicit error rules includes: Cousot's work on types as abstract interpretations [36]; Danielsson's work on operational semantics using the partiality monad [24]; 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 [37; 38; 39], whereby the standard small-step type safety proof technique applies. A preliminary study [40] 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 [36].

Our example definite assignment analysis was slightly easier to prove using big-step semantics. Klein and Nipkow's [41] proof of safety for a more sophisticated definite assignment analysis for a Java-like language is also based on big-step semantics.

Generic divergence was used in [40; 42] for giving a semantics for the untyped λ-calculus. This article expands those works by considering a while-language instead, showing how to use I-MSOS to automatically transform inductive big-step rules to incorporate generic divergence and traces, and how such transformations are useful for verifying program properties.

9. Conclusion

We presented a novel approach to augmenting standard big-step semantics to express divergence on a par with traditional approaches. Our approach to representing divergence uses fewer rules than existing approaches of similar expressiveness, and can be generated automatically by extending the I-MSOS framework. This extension also allows us to automatically generate trace-based semantics in the style of Nakata and Uustalu [7].

We also considered how to calculate static analyses based on big-step semantics, and compared big-step and small-step proofs. Our calculation relies on a mechanical instrumentation of the semantics, and a safety property about the instrumented semantics. Using this safety principle, we showed how to calculate a static definite assignment analysis. The proof based on big-step semantics is simpler than the corresponding proof for the small-step semantics. This provides evidence that automatically generated variants of big-step semantics are useful in practice.

Our experiments show that transformational techniques as embodied in I-MSOS are a viable tool for deriving interesting variants of structural specifications of programming languages, including our novel approach to concisely representing divergence using big-step semantics.

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

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

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

[6] 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.

[7] 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.

[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] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, Springer, 2004.

[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] P. Bahr, G. Hutton, Calculating correct compilers, unpublished, URL: http://www.cs.nott.ac.uk/~gmh/ccc.pdf (2014).

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

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

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

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

[16] 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.

[17] 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.

[18] R. Backhouse, Program Construction: Calculating Implementations from Specificationse, John Wiley and Sons, Inc., 2003.

[19] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: POPL'79, ACM, 1979, pp. 269–282. doi:10.1145/567752.567778.

[20] 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.

[21] 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.

[22] P. D. Mosses, Denotational semantics, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science (Vol. B), MIT Press, 1990, pp. 575–631.
[23] V. Capretta, General recursion via coinductive types, LMCS 1 (2) (2005) 1–18. doi:10.2168/LMCS-1(2:1)2005.
[24] N. A. Danielsson, Operational semantics using the partiality monad, in: ICFP'12, ACM, 2012, pp. 127–138. doi:10.1145/2364527.2364546.
[25] 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.
[26] M. Piróg, J. Gibbons, The coinductive resumption monad, ENTCS 308 (2014) 273–288. doi:10.1016/j.entcs.2014.10.015.
[27] 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.
[28] E. Moggi, Notions of computation and monads, Inf. Comput. 93 (1) (1991) 55–92. doi:10.1016/0890-5401(91)90052-4.
[29] P. Cenciarelli, E. Moggi, A syntactic approach to modularity in denotational semantics, in: Category Theory and Computer Science, 1993.
[30] P. D. Mosses, Modular structural operational semantics, J. Log. Algebr. Program. 60-61 (2004) 195–228. doi:10.1016/j.jlap.2004.03.008.
[31] 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.
[32] G. Hutton, J. Wright, What is the meaning of these constant interruptions?, JFP 17 (2007) 777–792. doi:10.1017/S0956796807006363.
[33] 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.
[34] A. K. Wright, M. Felleisen, A syntactic approach to type soundness, Inf. Comput. 115 (1) (1994) 38–94. doi:10.1006/inco.1994.1093.
[35] 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.
[36] P. Cousot, Types as abstract interpretations, in: POPL'97, ACM, 1997, pp. 316–331. doi:10.1145/263699.263744.
[37] 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.
[38] 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.
[39] 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.
[40] 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.
[41] G. Klein, T. Nipkow, A machine-checked model for a java-like language, virtual machine, and compiler, ACM Trans. Program. Lang. Syst. 28 (4) (2006) 619–695. doi:10.1145/1146809.1146811.
[42] C. Bach Poulsen, P. D. Mosses, Divergence as state in coinductive big-step semantics, presented at NWPT'14 (2014).

Appendix A. Coinductive Trace-Based SOS for Definite Assignment

The following SOS rules correspond to the I-MSOS relation for definite assignment defined in Sect. 6.2:

$$\boxed{(e, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IEA}} (v, \langle (R', A) \rangle)}$$

$$\frac{}{(v, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IEA}} (v, \langle (R, A) \rangle)}\ \text{IEA-Val} \qquad \frac{x \in \text{dom}(\sigma)}{(x, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IEA}} (\sigma(x), \langle (R \cup \{x\}, A) \rangle)}\ \text{IEA-Var}$$

$$\frac{(e_1, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IEA}} (n_1, \langle (R', A) \rangle) \qquad (e_2, \sigma, \langle (R', A) \rangle) \Rightarrow_{\text{IEA}} (n_2, \langle (R'', A) \rangle)}{(bop(e_1, e_2), \sigma, \langle (R, A) \rangle \Rightarrow_{\text{IEA}} (n_1\ bop\ n_2, \langle (R'', A) \rangle)}\ \text{IEA-Bop}$$

$$\textsf{co}\frac{}{(\textsf{skip}, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IA}} (\sigma, (R, A) :: \langle (R, A) \rangle)}\ \text{IA-Skip} \qquad \boxed{(c, \sigma, T) \Rightarrow_{\text{IA}} (\sigma', T')}$$

$$\textsf{co}\frac{x \notin \text{dom}(\sigma)}{(\textsf{alloc}\ x, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IA}} (\sigma[x \mapsto \textsf{null}], (R, A) :: \langle (R, A) \rangle)}\ \text{IA-Alloc}$$

$$\textsf{co}\frac{x \in \text{dom}(\sigma) \qquad (e, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IEA}} (v, \langle (R', A) \rangle)}{(x := e, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IA}} (\sigma[x \mapsto v], (R, A) :: (R', A) :: \langle (R', A \cup \{x\}) \rangle)}\ \text{IA-Assign}$$

$$\textsf{co}\frac{(c_1, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IA}} (\sigma', T) \qquad (c_2, \sigma', T) \Rightarrow_{\text{IA}} (\sigma'', T')}{(c_1; c_2, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IA}} (\sigma'', T')}\ \text{IA-Seq}$$

$$\textsf{co}\frac{(e, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IEA}} (v, \langle (R', A) \rangle) \qquad v \neq 0 \qquad (c_1, \sigma, \langle (R', A) \rangle) \Rightarrow_{\text{IA}} (\sigma', T)}{(\textsf{ifnz}\ e\ c_1\ c_2, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IA}} (\sigma', T)}\ \text{IA-If}$$

$$\textsf{co}\frac{(e, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IEA}} (0, \langle (R', A) \rangle) \qquad (c_2, \sigma, \langle (R', A) \rangle) \Rightarrow_{\text{IA}} (\sigma', T)}{(\textsf{ifnz}\ e\ c_1\ c_2, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IA}} (\sigma', T)}\ \text{IA-IfZ}$$

$$\textsf{co}\frac{(e, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IEA}} (v, \langle (R', A) \rangle) \qquad v \neq 0 \qquad (c, \sigma, \langle (R', A) \rangle) \Rightarrow_{\text{IA}} (\sigma', T) \qquad (\textsf{whilenz}\ e\ c, \sigma', T) \Rightarrow_{\text{IA}} (\sigma'', T')}{(\textsf{whilenz}\ e\ c, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IA}} (\sigma'', T')}\ \text{IA-While}$$

$$\textsf{co}\frac{(e, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IA}} (0, \langle (R', A) \rangle)}{(\textsf{whilenz}\ e\ c, \sigma, \langle (R, A) \rangle) \Rightarrow_{\text{IA}} (\sigma, (R, A) :: \langle (R', A) \rangle)}\ \text{IA-WhileZ}$$

$$\frac{(c, \sigma, T) \Rightarrow_{\text{IA}} (\sigma', T')}{(c, \sigma, (R, A) :: T) \Rightarrow_{\text{IA}} (\sigma', (R, A) :: T')}\ \text{IA-Peel}$$

Appendix B. Inductive SOS with Definite Assignment as Abrupt Termination

The following SOS rules correspond to the I-MSOS relation for definite assignment as abrupt termination defined in Sect. 7.1:

$$\boxed{(e, \sigma, A, \varepsilon) \Rightarrow_{\text{IEB}} (v, \varepsilon')}$$

$$\frac{}{(v, \sigma, A, \textsf{ok}) \Rightarrow_{\text{IEB}} (v, \textsf{ok})}\ \text{IEB-Val} \qquad \frac{x \in \text{dom}(\sigma) \qquad x \in A}{(x, \sigma, A, \textsf{ok}) \Rightarrow_{\text{IEB}} (\sigma(x), \textsf{ok})}\ \text{IEB-Var} \qquad \frac{x \in \text{dom}(\sigma) \qquad x \notin A}{(x, \sigma, A, \textsf{ok}) \Rightarrow_{\text{IEB}} (v, \textsf{err})}\ \text{IEB-Var-Err}$$

$$\frac{(e_1, \sigma, A, \textsf{ok}) \Rightarrow_{\text{IEB}} (n_1, \varepsilon) \qquad (e_2, \sigma, A, \varepsilon) \Rightarrow_{\text{IEB}} (n_2, \varepsilon')}{(bop(e_1, e_2), \sigma, A, \textsf{ok}) \Rightarrow_{\text{IEB}} (n_1\ bop\ n_2, \varepsilon')}\ \text{IEB-Bop} \qquad \frac{}{(e, \sigma, A, \textsf{err}) \Rightarrow_{\text{IEB}} (v, \textsf{err})}\ \text{IEB-Err}$$

$$\boxed{(c, \sigma, A, \varepsilon) \Rightarrow_{\text{IB}} (\sigma', A', \varepsilon')}$$

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

$$\frac{x \in \mathrm{dom}(\sigma) \qquad (e, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IEB}} (v, \varepsilon)}{(x := e, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IB}} (\sigma[x \mapsto v], A \cup \{x\}, \varepsilon)}\ \text{IB-Assign} \qquad \frac{(c_1, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IB}} (\sigma', A', \varepsilon) \quad (c_2, \sigma', A', \varepsilon) \Rightarrow_{\text{IB}} (\sigma'', A'', \varepsilon')}{(c_1; c_2, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IB}} (\sigma'', A'', \varepsilon')}\ \text{IB-Seq}$$

$$\frac{(e, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IEB}} (v, \varepsilon) \qquad v \neq 0 \quad (c_1, \sigma, A, \varepsilon) \Rightarrow_{\text{IB}} (\sigma', A', \varepsilon')}{(\mathsf{ifnz}\ e\ c_1\ c_2, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IB}} (\sigma', A', \varepsilon')}\ \text{IB-If} \qquad \frac{(e, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IEB}} (0, \varepsilon) \quad (c_2, \sigma, A, \varepsilon) \Rightarrow_{\text{IB}} (\sigma', A', \varepsilon')}{(\mathsf{ifnz}\ e\ c_1\ c_2, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IB}} (\sigma', A', \varepsilon')}\ \text{IB-IfZ}$$

$$\frac{(e, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IEB}} (v, \varepsilon) \qquad v \neq 0 \quad (c, \sigma, A, \varepsilon) \Rightarrow_{\text{IB}} (\sigma', A', \varepsilon') \quad (\mathsf{whilenz}\ e\ c, \sigma', A', \varepsilon') \Rightarrow_{\text{IB}} (\sigma'', A'', \varepsilon'')}{(\mathsf{whilenz}\ e\ c, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IB}} (\sigma'', A'', \varepsilon'')}\ \text{IB-While} \qquad \frac{(e, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IEB}} (0, \varepsilon)}{(\mathsf{whilenz}\ e\ c, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IB}} (\sigma, A, \varepsilon)}\ \text{IB-WhileZ}$$

$$\mathsf{du}\frac{}{(c, \sigma, A, \mathsf{err}) \Rightarrow_{\text{IB}} (\sigma', A', \mathsf{err})}\ \text{IB-Err}$$

Appendix C. Inductive SOS with Definite Assignment as Abrupt Termination

The following SOS rules correspond to the small-step I-MSOS relation for definite assignment as abrupt termination alluded to in Sect. 7.3. Here, $\Rightarrow_{\text{IEB}}$ is as defined in Sect. 7.1 (with SOS rules in Appendix B):

$$\boxed{(c, \sigma, A, \varepsilon) \rightarrow_{\text{SB}} (c', \sigma', A', \varepsilon)}$$

$$\frac{x \notin \mathrm{dom}(\sigma)}{(\mathsf{alloc}\ x, \sigma, A, \mathsf{ok}) \rightarrow_{\text{SB}} (\mathsf{skip}, \sigma[x \mapsto \mathsf{null}], A, \mathsf{ok})}\ \text{SB-Alloc}$$

$$\frac{x \in \mathrm{dom}(\sigma) \qquad (e, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IEB}} (v, \varepsilon)}{(x := e, \sigma, A, \mathsf{ok}) \rightarrow_{\text{SB}} (\mathsf{skip}, \sigma[x \mapsto v], A \cup \{x\}, \varepsilon)}\ \text{SB-Assign}$$

$$\frac{(c_1, \sigma, A, \mathsf{ok}) \rightarrow_{\text{SB}} (c_1', \sigma', A', \varepsilon)}{(c_1; c_2, \sigma, A, \mathsf{ok}) \rightarrow_{\text{SB}} (c_1'; c_2, \sigma', A', \varepsilon)}\ \text{SB-Seq} \qquad \frac{}{(\mathsf{skip}; c_2, \sigma, A, \mathsf{ok}) \rightarrow_{\text{SB}} (c_2, \sigma, A, \mathsf{ok})}\ \text{SB-SeqSkip}$$

$$\frac{(e, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IEB}} (v, \varepsilon) \qquad v \neq 0}{(\mathsf{ifnz}\ e\ c_1\ c_2, \sigma, A, \mathsf{ok}) \rightarrow_{\text{SB}} (c_1, \sigma, A, \varepsilon)}\ \text{SB-If} \qquad \frac{(e, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IEB}} (0, \varepsilon)}{(\mathsf{ifnz}\ e\ c_1\ c_2, \sigma, A, \mathsf{ok}) \rightarrow_{\text{SB}} (c_2, \sigma, A, \varepsilon)}\ \text{SB-IfZ}$$

$$\frac{(e, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IEB}} (v, \varepsilon) \qquad v \neq 0}{(\mathsf{whilenz}\ e\ c, \sigma, A, \mathsf{ok}) \rightarrow_{\text{SB}} (c; \mathsf{whilenz}\ e\ c, \sigma, A, \varepsilon)}\ \text{SB-While}$$

$$\frac{(e, \sigma, A, \mathsf{ok}) \Rightarrow_{\text{IEB}} (0, \varepsilon)}{(\mathsf{whilenz}\ e\ c, \sigma, A, \mathsf{ok}) \rightarrow_{\text{SB}} (\mathsf{skip}, \sigma, A, \varepsilon)}\ \text{SB-WhileZ}$$