Big-Stop Semantics

Small-Step Semantics in a Big-Step Judgment

DAVID M KAHN, Denison University, USA
JAN HOFFMANN, Carnegie Mellon University, USA
RUNMING LI, Carnegie Mellon University, USA

As is evident in the programming language literature, many practitioners favor specifying dynamic program behavior using big-step over small-step semantics. Unlike small-step semantics, which must dwell on every intermediate program state, big-step semantics conveniently jumps directly to the ever-important result of the computation. Big-step semantics also typically involves fewer inference rules than their small-step counterparts. However, in exchange for ergonomics, big-step semantics gives up power: Small-step semantics describes program behaviors that are outside the grasp of big-step semantics, notably divergence.

This work presents a little-known extension of big-step semantics with inductive definitions that captures diverging computations without introducing error states. This *big-stop* semantics is illustrated for typed, untyped, and effectful variants of PCF, as well as a while-loop-based imperative language. Big-stop semantics extends the standard big-step inference rules with a few additional rules to define an evaluation judgment that is equivalent to the reflexive-transitive closure of small-step transitions. This simple extension contrasts with other solutions in the literature that sacrifice ergonomics by introducing many additional inference rules, global state, and/or less-commonly-understood reasoning principles like coinduction. The ergonomics of big-stop semantics is exemplified via concise Agda proofs for some key results and compilation theorems.

CCS Concepts: • **Theory of computation → Operational semantics**; *Algebraic semantics.*

Additional Key Words and Phrases: semantics, dynamic semantics, operational semantics, big-step semantics, small-step semantics, big-stop semantics, nontermination, divergence, verification

ACM Reference Format:
David M Kahn, Jan Hoffmann, and Runming Li. 2018. Big-Stop Semantics: Small-Step Semantics in a Big-Step Judgment. 1, 1 (November 2018), 35 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Operational semantics is a popular choice for defining the dynamic behavior of programs, both in the programming language literature and for precisely defining industrial-strength languages such as Standard ML [36], C [4, 13, 38], or Web Assembly [1, 45]. Advantages of operational semantics include that they are accessible to non-experts without in-depth mathematical training and that they scale well to describe novel and advanced language features.

Operational semantics can be characterized as belonging to two main styles: The small-step style defines a step relation between the states of a transition system in which the states correspond to programs. It originates from Plotkin's structural operational semantics (SOS) [40] and encompasses abstract machines [17]. The outcome of an evaluation of a program is a sequence of transitions

Authors' Contact Information: David M Kahn, kahnd@denison.edu, Denison University, Granville, Ohio, USA; Jan Hoffmann, jhoffmann@cmu.edu, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; Runming Li, runming@cmu.edu, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

ACM XXXX-XXXX/2018/11-ART
https://doi.org/XXXXXXX.XXXXXXX

that starts at the state that corresponds to the program. Transitions are usually defined inductively using syntax-directed inference rules.

The big-step style, first proposed by Kahn [28], defines a relation between programs and outcomes. Outcomes often consist of a returned value and potentially an effect, for instance, the cost of the evaluation or a sequence of I/O operations. A big-step operational semantics does not define a notion of a step and is thus also—perhaps more appropriately—called evaluation dynamics [19] or natural semantics [28]. Like transitions in a small-step semantics, the big-step evaluation relation is usually defined inductively with syntax-directed inference rules.

For most purposes, big-step and small-step semantics are considered to be equally expressive for describing terminating computations but not for diverging (non-terminating) computations. While both the step relation of small-step and the evaluation relation of big-step semantics are inductively defined, small-step semantics naturally formalizes diverging computations: They are unbounded transition sequences that never reach a final state. Capturing diverging computations in the dynamics is important for describing their effects and for distinguishing them from failing computations (transition sequences that get stuck in a state that is not final), which is crucial for formulating and proving type soundness. Big-step semantics, in its standard definition, only relates terminating programs to their outcomes. This means that it does not describe the effects of diverging computations and cannot distinguish between diverging and failing computations. Big-step semantics is therefore not suitable for some purposes such as formulating type soundness, which can be expressed elegantly with small-step semantics using *progress* and *preservation* [46].

Despite this shortcoming, big-step semantics remains a popular choice in the literature [6, 35]. The use of big-step semantics in comprehensive and influential works, such as the definition of Standard ML [36] and formalizations of C [4, 38], is a testament to its scalability, succinctness, and readability. Many authors prefer big-step semantics because it can lead to simpler and more succinct proofs. Examples include correctness proofs for semantics-preserving program translations arising in compiler verification [29, 34, 44] as well as for type soundness proofs where type judgments imply properties of global outcomes, such as resource bounds [22, 23]. One reason is that the structure of big-step semantics often corresponds to the typing rules, and big-step semantics directly describes outcomes without fussing with the intermediate program states reached along the way.

There is a line of work that proposes to use *coinductive* definitions to remedy the deficiencies of big-step semantics for describing divergence [6–11, 24, 35, 37, 41, 47]. Such work aims to generalize big-step semantics to enjoy its advantages in situations where traditional big-step semantics is not suitable. Since the outcome of a diverging computation can be infinite, it is natural to use coinduction to capture outcomes. However, coinductive big-step semantics has some undesirable properties and has not been widely adopted. For one thing, proofs by induction on the semantic judgment turn into coinductive proofs that hamper the close correspondence with inductively defined type judgments. For another thing, coinductive big-step semantics can yield counter-intuitive notions of evaluation (coevaluation) or result in an explosion in the number of inference rules [35].

In this work, we develop a little-known alternative approach that extends big-step semantics with *inductive* rules to capture diverging computations. Our work is based on an idea proposed by Hoffmann and Hofmann that we dub *big-stop semantics* [21]. Instead of describing complete, possibly infinite outcomes using coinduction, big-stop semantics inductively describes all finite prefixes of outcomes. Moreover, big-stop semantics does so without significantly altering the language, whereas other inductive work adds features like *error states* and *fuel* [2, 14, 43] (see Section 9). Inductively describing finite prefixes of computation corresponds to small-step semantics.

The essence of big-stop semantics is to take a standard formulation of big-step semantics for a language and add just one rule schema for nondeterministically halting the computation. In a well-chosen setting, this schema can even be reduced to just a single rule (see Section 7). Because big-stop semantics is just a small extension to standard big-step semantics, it retains all the typical benefits of big-step semantics. However, because big-stop semantics can precisely match the relation of small-step semantics, big-stop semantics is suitable for reasoning about nonterminating computation.

This work makes the following contributions:

- The big-stop system of operational semantics is described for purposes of simplifying proofs covering diverging computation. This system is created by extending big-step with rules for nondeterministically stopping computation. This extension is exemplified for call-by-value PCF [39] both with and without effects (Sections 4 and 5), as well as a while-loop-based imperative language (Section 8), showing that this technique is broadly applicable.
- Formal properties of the big-stop system for call-by-value PCF are proven in Agda [27] as marked by the symbol . These properties include the key equivalence of Theorem 16 showing that big-stop semantics coincides with the reflexive-transitive closure of small-step semantics and the progress property Theorem 18. See Section 5 for further discussion.
- In Section 6, Agda proofs (again marked by) are given for the correctness of compiling effectful call-by-value PCF to a Krivine machine [30]. These proofs exhibit the ease of big-stop formalization where using small-step semantics is unpleasant and using big-step semantics is insufficient. They also exemplify a general principle—which we call the *big-stop method*—for systematically extending a theorem about terminating computation that relies on big-step semantics to diverging computations.
- Further ergonomic optimizations to big-stop semantics are outlined in Section 7. These optimizations allow big-step semantics to be extended to big-stop semantics with as little as one additional rule, which can significantly reduce the number of proof cases required when performing rule induction. As described in Section 9, existing systems covering the same niche as big-stop semantics typically require more complex extensions.

2 Small-Step and Big-Step Semantics

We start by providing the standard definitions of small-step and big-step semantics.

2.1 Call-By-Value PCF

The language we consider throughout most of this article and in the mechanization is a variant of PCF [39] with call-by-value (or eager) evaluation order. We choose PCF to be the basis of our language because it is simple yet expressive enough to demonstrate the most interesting properties of big-stop semantics. Nonetheless, we do not believe that call-by-value evaluation is essential and expect all concepts to transfer to call-by-name without complications.[1] Note that we also consider untyped PCF expressions to make certain points that are masked in the typed language.

Expressions and values. The expression syntax of our version of pure PCF is given by the grammar of e in Figure 1a where f, x range over variable names. In the order appearing in the grammar, expressions can be variables, unary natural numbers, the case analysis for natural numbers, recursive functions, or function applications. The expression grammar is set up so that every expression in the language is of the form $E[e_1, \dots, e_n]$, where E indicates the kind of expression along with any variable bindings, and where each e_i is the i^{th} subexpression that would need to be a value for the whole expression to be a redex. This notation makes it convenient to pick out the list of such subexpressions uniformly from any given expression. It should be noted that, despite superficial similarities, this notation is *not* the evaluation context notation of Felleisen [15] (though we discuss

[1] Launchbury has shown that call-by-need languages also admit big-step semantics [32], which should be readily adaptable.

$$e \ ::= \ x \mid \mathsf{Z}[\,] \mid \mathsf{S}[e'] \mid \mathsf{case}\{e_1; x.e_2\}[e_3] \mid \mathsf{fun}\{f, x.\, e\}[\,] \mid \mathsf{app}[e_1, e_2]$$

(a) Expressions

$$\text{V-Zero}\ \frac{}{\mathsf{Z}\ \mathit{val}} \qquad \text{V-Succ}\ \frac{v\ \mathit{val}}{\mathsf{S}[v]\ \mathit{val}} \qquad \text{V-Fun}\ \frac{}{\mathsf{fun}\{f, x.\, e\}\ \mathit{val}}$$

(b) Values

Fig. 1. Language of call-by-value PCF

evaluation contexts more in Section 2.2 and Section 7). To make this notation friendlier, we elide empty such lists so that, e.g., we write Z instead of $\mathsf{Z}[\,]$.

As expected of PCF, this language extends lambda calculus with natural numbers and a fixed-point operator. The expression Z represents the number 0 and $\mathsf{S}[n]$ represents $n+1$. The presence of primitive natural numbers provides a simple way of writing (ill-typed) programs that get stuck, such as the application of zero to itself, $\mathsf{app}[\mathsf{Z}, \mathsf{Z}]$. Additionally, the presence of a fixed-point operator allows one to write nonterminating programs, even in a well-typed setting (for types, see Figure 28).

Values v of the language are indicated by the judgment $v\ \mathit{val}$. The rules defining this judgment are given in Figure 1b. The only well-typed values of the language are natural numbers and functions.

PCF typically evaluates its fixed-point operator lazily, but we wish to work in an eager setting. There are at least two ways to work around this obstacle: either explicitly work with thunks, or change to a suitable eager fixed-point operator. While either solution should suffice, our presentation uses the latter solution, effectively exchanging the Y fixed-point operator for the Z fixed-point operator. This exchange results in the expression $\mathsf{fun}\{f, x.\, e\}$, which defines the recursive function f with parameter x. This fixed-point expression also happens to subsume the behaviour of function abstraction, so we simplify by having it perform that role as well. When abstracting a non-recursive function, we may simply write $\mathsf{fun}\{\_, x.\, e\}$.

Static Semantics. The static semantics of PCF is based on the type judgment $\Gamma \vdash e : \tau$. It states that the expression e has type τ in context Γ, which assigns types to the free variables in e. The definition is standard and given in Figure 28 in Section A.

The programs of PCF are defined to be well-typed, closed expressions, that is, expressions e such that $\cdot \vdash e : \tau$ for some τ and the empty context $\cdot$. In the mechanization of the results, we only consider well-typed expressions. However, many of our results also hold for untyped expressions and we present these stronger results if applicable.

2.2 Small-Step Semantics

Small-step semantics grew out of Gordon Plotkin's structural operational semantics (SOS) [40]. The idea behind this approach is to describe individual computation steps in a transition system in a structurally-directed manner. Describing operational semantics in this way allows for program behaviour to be reasoned about inductively in a straightforward, syntax-oriented way.

Pure PCF's call-by-value small-step transitions are given in Figure 2, where the judgment $e_1 \mapsto e_2$ means that the expression e_1 transitions to the expression e_2 in one computational step. We then take the reflexive-transitive closure of this small-step relation, $e_1 \overset{*}{\mapsto} e_2$, to describe sequences of computation as defined in Figure 3.

Small-step semantics typically require multiple rules for language constructs with multiple subexpressions, which are called *congruence rules*. Such rules can bloat the number of rules required for small-step semantics. However, the rules of Figure 2 avoid listing out all the congruence rules through the use of the rule schema S-Seq(κ). For each choice of the parameter k, this schema results

$$\text{S-Seq(k)}\quad \frac{1 \le k \le n \qquad \forall\, 1 \le i < k.\, e_i\ val \qquad e_k \mapsto e_k' \qquad \forall\, i \ne k.\, e_i = e_i'}{E[e_1, \ldots, e_n] \mapsto E[e_1', \ldots, e_n']}$$

$$\text{S-CaseZ}\quad \frac{}{\mathsf{case}\{e_1; x.e_2\}[\mathsf{Z}] \mapsto e_1}$$

$$\text{S-CaseS}\quad \frac{v\ val}{\mathsf{case}\{e_1; x.e_2\}[\mathsf{S}[v]] \mapsto [v/x]e_2}$$

$$\text{S-App}\quad \frac{v\ val}{\mathsf{app}[\mathsf{fun}\{f, x.\, e\}, v] \mapsto [\mathsf{fun}\{f, x.\, e\}/f,\, v/x]e}$$

Fig. 2. Small-step transitions for pure call-by-value PCF

in a rule that says that one computation step can be made, in place, for the k^{th} subexpression of $E[e_1, \ldots, e_n]$ so long as all prior subexpressions are already values. This setup also enforces left-to-right evaluation of the subexpressions. For our particular language, this rule schema replaces four congruence rules: one each to step the function and argument in an application, one to step the contents of a successor, and one to step the argument to a case analysis.

An alternative to using our rule schema is to use evaluation contexts [15]. Evaluation contexts provide a meta-syntax for expressions that helps deal with congruences by cutting out a hole to indicate where an expression is to step next. In a small-step semantics with evaluation contexts, we would replace the rule schema S-Seq(k) with one rule stating that expressions in holes can step independently of their surrounding context. However, this approach inherently orients the language toward small-step semantics, as big-step semantics considers more than just the next step of evaluation. Thus, to prevent obscuring eventual comparison between big-step and big-stop semantics, we initially focus on schema-based semantics and revisit evaluation contexts in Section 7.

$$\text{M-Refl}\quad \frac{}{e \overset{*}{\mapsto} e} \qquad\qquad \text{M-Step}\quad \frac{e_1 \mapsto e_2 \qquad e_2 \overset{*}{\mapsto} e_3}{e_1 \overset{*}{\mapsto} e_3}$$

Fig. 3. Multi-step semantics

Aside from S-Seq(k), the remaining three rules of Figure 2 are simply the rules for redexes in our language. S-CaseZ and S-CaseS are the case analysis rules for zero and non-zero numbers, respectively, and S-App is the rule for function application.

Small-step systems of operational semantics are used to state some foundational results in the study of programming languages. For example, small-step semantics are used to state the concepts of *progress* and *preservation*, which are the key underpinnings of syntactic proofs of type soundness [46]. “Progress” states that well-typed expressions either are values or can take a small step. “Preservation” states that the typing of an expression is maintained through taking small steps. If both progress and preservation hold for a language, then well-typed expressions cannot get stuck at type errors during evaluation. This approach does not require that evaluation terminates.

2.3 Big-Step Semantics

Big-step semantics originates from Gilles Kahn’s natural semantics [28]. In a sense, the big-step approach achieves the main thrust of small-step semantics by directly relating expressions to the values they reduce to. The big-step relation is written here as $e \Downarrow v$, meaning that the expression e evaluates to the value v. The rules of big-step semantics are syntax-directed and given in Figure 4.

Crucially, these big-step semantics match those given by the small-step system of Figure 2. This equivalence can be formalized by Theorem 1, which can be proved by straightforward rule induction. More precisely, this equivalence states that the two systems agree on terminating computations.

$$\frac{v\ \mathit{val}}{v \Downarrow v}\ \text{B-Val} \qquad \frac{e \Downarrow v}{\mathsf{S}[e] \Downarrow \mathsf{S}[v]}\ \text{B-Succ} \qquad \frac{e \Downarrow \mathsf{Z} \quad e_1 \Downarrow v_1}{\mathsf{case}\{e_1; x.e_2\}[e] \Downarrow v_1}\ \text{B-CaseZ} \qquad \frac{e \Downarrow \mathsf{S}[v] \quad [v/x]e_2 \Downarrow v_2}{\mathsf{case}\{e_1; x.e_2\}[e] \Downarrow v_2}\ \text{B-CaseS}$$

$$\frac{e_1 \Downarrow \mathsf{fun}\{f, x.\, e\} \quad e_2 \Downarrow v_2 \quad [\mathsf{fun}\{f, x.\, e\}/f, v_2/x]e \Downarrow v}{\mathsf{app}[e_1, e_2] \Downarrow v}\ \text{B-App}$$

Fig. 4. Big-step semantics for pure call-by-value PCF

However, this is already sufficient to obtain equivalence of the two semantics for a type-safe language without effects such as pure PCF since divergence is the only other possible outcome.

Theorem 1 (Big/Multi Equivalence). *For all expressions e and values v, $e \Downarrow v \iff e \overset{*}{\mapsto} v$.*

Big-step rules are very similar in structure to sequent-style rules for natural deduction.[2] Because typical typing rules are also structurally similar to these rules, there is a good correspondence between a language's types and big-step operational semantics. This high level of structural similarity between semantics and typing rules comes with ergonomic benefits. For example, proving type soundness for these rules is rather straightforward because the structures of the derivations of corresponding judgments match. In contrast, the small-step system's congruence rules typically have no structural match to typing rules, rendering a more cumbersome type soundness proof that might require, e.g., context substitution lemmas.

Another benefit of using big-step semantics is that intermediate computations never need to be inspected. This property manifests in how the application of a given big-step rule depends solely on the identity and subexpressions of the expression e in $e \Downarrow v$; the value v is irrelevant for syntax-guided rule application. In contrast, rule M-Step from Figure 3 has expression e_2 as the righthand element of the premiss $e_1 \mapsto e_2$ *and* the left-hand element of the premiss $e_2 \overset{*}{\mapsto} e_3$. As a result, every intermediate computation e_2 must be inspected when using multi-step semantics.

However, the benefits of big-step semantics come at a cost: if an expression does not evaluate, big-step semantics cannot reason about it. As a result, programs that loop forever, get stuck on undefined behavior, etc. cannot be described by big-step semantics. In contrast, small-step semantics can describe execution arbitrarily deep into infinite loops and right up to getting stuck.

3 Motivation: Semantic Preservation

It has been argued in the literature [6, 35] that big-step semantics can simplify compiler correctness proofs. When the dynamic semantics of the source language is different from the (lower-level) dynamic semantics of the target language, it can be hard for a proof of semantic preservation to proceed by relating each language's intermediate states of computation, as they may not share any convenient structural similarities. Big-step semantics can avoid these problems by completely skipping past intermediate states of computation.

Here we exhibit a particularly elementary setting where this problem still arises: semantic preservation for call-by-value PCF with respect to a Krivine machine (K machine) [12, 30, 33]. If we view this problem through the lens of compiler correctness, then the source and target languages are identical, so the compiler should simply copy the source unchanged into the K machine. However, this case already demonstrates that typical proofs of semantic preservation for compilation with

[2] Hence the name "natural semantics".

respect to PCF's small-step semantics are inconvenient. This section describes (1) where such inconveniences arise, (2) how big-step semantics can be used to alleviate them, and (3) why it is desirable to generalize this approach to diverging computations using big-stop semantics.

The K Machine. A Krivine machine (K machine) is a stack-based, abstract machine similar to the SECD machine [31]. While Krivine [30] designed his machine for call-by-name, the concept is quite flexible and has been adapted here for our call-by-value PCF. Our presentation of the K machine follows the notational conventions of *Practical Foundations of Programming Languages* (PFPL) [19] (Chapter 28), which also considers the correctness of a K machine with respect to SOS.

Whereas small-step systems need various congruence rules for propagating computation into subexpressions (e.g., S-Seq(κ) and SE-Seq(κ)), a K machine deals with such congruences directly in its term language by working with an explicit stack. The K machine operates on an expression by pushing subexpressions to be evaluated onto the stack and popping values from the stack.

Each state in the K machine takes the form of either $k \triangleright e$ or $k \triangleleft e$ where k is a stack of *frames* and e is a PCF expression. States of the form $k \triangleright e$ indicate that the expression e is not yet a value and needs to be evaluated, and states of the form $k \triangleleft v$ indicate that v has been found to be a value and is ready to be plugged into the top frame of the stack k. These features of the K machine are formally described in Section 6 across Figures 9 and 10. In particular, Figure 10 defines the judgment $S \mapsto_K T$ to indicate that state S transitions to state T, in addition to its multi-step analogue $S \overset{*}{\mapsto}_K T$. For example, the following rules define the transitions for function application.

$$\text{K-App1}\ \frac{}{k \triangleright \mathsf{app}[e_1, e_2] \mapsto_K k; \langle \mathsf{app}[-, e_2] \rangle \triangleright e_1} \qquad \text{K-App2}\ \frac{}{k; \langle \mathsf{app}[-, e] \rangle \triangleleft v \mapsto_K k; \langle \mathsf{app}[v, -] \rangle \triangleright e} \qquad \text{K-App3}\ \frac{}{k; \langle \mathsf{app}[\mathsf{fun}\{f, x.\, e\}, -] \rangle \triangleleft v \mapsto_K k \triangleright [\mathsf{fun}\{f, x.\, e\}/f, v/x]e}$$

3.1 Proving the Correctness of the K machine

Now we can discuss compiling PCF to the K machine and semantic preservation. Compiling a PCF expression e to the K machine simply yields the state $\epsilon \triangleright e$. To verify that this (trivial) compilation preserves the semantics of expressions, the following two key lemmas are desired.

Lemma 2 (Soundness). *If* $\epsilon \triangleright e \overset{*}{\mapsto}_K \epsilon \triangleleft v$ *then* $e \overset{*}{\mapsto} v$.

Lemma 3 (Completeness). *If* $e \overset{*}{\mapsto} v$ *and* v *val then* $\epsilon \triangleright e \overset{*}{\mapsto}_K \epsilon \triangleleft v$.

Even though these lemmas are essentially by-the-book, they are already inconvenient to prove, and we largely follow PFPL [19] to demonstrate this fact. The problem is that there does not exist a direct proof by rule induction on the judgments on the left side of the implication. Instead, the classic proof approaches require the formal development of additional theoretical machinery, such as simulation relations or alternative semantics, in particular for the proof of Lemma 3, which we focus on in this section. To summarize the results that we find: proofs using big-step semantics can be carried out by simple rule induction and are easier than other approaches.

Proving Completeness Directly. The direct proof of Lemma 3 (completeness) would induct over the derivation of the antecedent. So consider proving completeness by rule induction over the derivation of the multi-step relation $e \overset{*}{\mapsto} v$. First, it is necessary to generalize the inductive hypothesis to consider any stack k: If $e \overset{*}{\mapsto} v$ and v *val* then $k \triangleright e \overset{*}{\mapsto}_K k \triangleleft v$

Now consider the structure of the multi-step relation's derivation. One possibility is that the relation was derived via M-Step, requiring the premisses $e \mapsto e'$ and $e' \overset{*}{\mapsto} v$ for some expression e'. The latter premiss is ripe for applying the inductive hypothesis, but the former is not because it does not use the multi-step judgment and e' may not be a value. Thus, further reasoning is warranted.

To continue the inductive proof, one needs to show the following statement, where the value v is the evaluation of both expressions e and e': If $e \mapsto e'$ and $k \triangleright e' \overset{*}{\mapsto}_K k \triangleleft v$, then $k \triangleright e \overset{*}{\mapsto}_K k \triangleleft v$.

Given the antecedent, by transitivity, it would suffice to show that $k \triangleright e \overset{*}{\mapsto}_K k \triangleright e'$. Similarly, by determinacy, it would suffice to show that $k \triangleright e' \overset{*}{\mapsto}_K k \triangleright e$. However, neither statement holds, as the next time the stack k should be present after transitioning from a state of the form $k \triangleright e''$ should be in the state $k \triangleleft v''$ where v'' is the evaluation of e''. Thus, neither of the transition sequences of $k \triangleright e$ nor $k \triangleright e'$ should be expected to be an extension of the other, and the first point at which their transition sequences coincide should be expected to be $k \triangleleft v$. This circumstance is structurally inconvenient, and one is forced to continue by inner induction on the structure of $e \mapsto e'$.

Many cases here pose a problem. Consider the case from S-Seq(1) where $e = \mathsf{app}[e_1, e_2]$ and $e' = \mathsf{app}[e_1', e_2]$ for some expressions e_1, e_1', e_2 where $e_1 \mapsto e_1'$. Here one wants to show that, if $k \triangleright \mathsf{app}[e_1', e_2] \overset{*}{\mapsto}_K k \triangleleft v$, then $k \triangleright \mathsf{app}[e_1, e_2] \overset{*}{\mapsto}_K k \triangleleft v$. To continue this case, note that K-App1 yields both that $k \triangleright \mathsf{app}[e_1, e_2] \mapsto_K k; \langle \mathsf{app}[-, e_2] \rangle \triangleright e_1$ and that $k \triangleright \mathsf{app}[e_1', e_2] \mapsto_K k; \langle \mathsf{app}[-, e_2] \rangle \triangleright e_1'$, which unwraps the first steps of both transition sequences of interest. At this point, the proof seems feasible if one can apply an inductive hypothesis. However, both the outer and inner inductive hypotheses would require the judgment $e_1' \overset{*}{\mapsto} v_1$ (at least as a subderivation of $e_1 \overset{*}{\mapsto} v_1$), but this judgment is not readily available and it is not clear how to proceed.

The Simulation Approach. Simulation arguments are an available avenue to proving Lemma 3 if one is willing to develop some additional proof machinery. Specifically, a simulation requires defining a functional relation $\looparrowright$ between K machine states and PCF expressions such that:

- $\epsilon \triangleright e \looparrowright e$
- $\epsilon \triangleleft v \looparrowright v$
- if $e \overset{*}{\mapsto} e'$ and $S \looparrowright e$, then there is some state T such that $T \looparrowright e'$, and $S \overset{*}{\mapsto}_K T$

A sensible such simulation relates each K machine state to the PCF expression obtained by unwinding its stack. For example, $\langle \mathsf{S}[\,-\,] \rangle \triangleright \mathsf{Z} \looparrowright \mathsf{S}[\mathsf{Z}]$. This relation clearly satisfies the first two necessary properties. However, verifying the last of the necessary properties for this relation is a problem. The issue can be seen when considering the case where e' is a value and S is chosen to be $\epsilon \triangleright e$. Then this final property is just a slight generalization of the completeness lemma itself. Proving this slight generalization is no easier.

Another conceptual annoyance for making a simulation argument is that there is no bound on how many K machine states may be related to an expression, nor is there a bound on how many steps of the K machine may be necessary to match a single step in PCF. As a result, one must at the very least perform an inner induction on the structure of the expression e, which runs into similar issues as the basic inductive approach.

Note that the complexity of a simulation argument would increase when considering language features like effects, different source and target languages, and/or more involved compiler passes.

The Big-Step Approach. The textbook approach to proving Lemma 3 is to avoid using small-step semantics entirely. First one derives Theorem 1 (big/multi equivalence), which follows from straightforward inductive arguments. Then Lemma 3 is a consequence of the following lemma, which can be directly proved by induction on the judgment $e \Downarrow v$.

Lemma 4 (Big-Step Completeness). *If $e \Downarrow v$ then $k \triangleright e \overset{*}{\mapsto}_K k \triangleleft v$.*

The key to the proof of Lemma 4 is that the big-step derivation tracks a resulting value v for every relevant subexpression of e. With such a value always handy, the recovery of such evaluations is trivial, and big-step semantics avoids the problem encountered by small-step semantics.

3.2 Accounting for Diverging Computations

If one assumes that programs are well-typed, then it is sufficient for Lemmas 2 and 3 to focus on computations that result values. Thanks to type-safety, computations cannot get stuck, so the only two possible outcomes for computations are to converge to a value or to diverge and loop forever. Because all divergence is observationally identical in a pure language, one only needs to ensure that every diverging expression in PCF is compiled to some diverging state in the K machine. This property clearly holds because Lemmas 2 and 3 clearly show the contrapositive—converging expressions are compiled to converging states.

However, the story changes if one considers effects, which real-world languages virtually always have. In a setting with effects, diverging computations can be distinguished as they may induce different effects. These distinctions arise even with very simple writer-monad-style effects that do not affect computation, like printing. To compile such a language correctly, it is necessary to ensure that the observable effects emitted by the source language and target language match, regardless of program termination.

In Sections 5 and 6, we extend both SOS and the K machine with effects, defining the judgments

$$e_1 \overset{*}{\mapsto} e_2 \rightsquigarrow a \qquad \text{and} \qquad S \overset{*}{\mapsto}_K T \rightsquigarrow a$$

where e_i are expressions, S and T are K machine states, and a is an abstract sequence of effects.

In a setting with effects, additional properties analogous to Lemmas 2 and 3 must be formalized to handle diverging computations. In particular, our goal is to prove Lemmas 5 and 6. Essentially, these properties state that, if one dynamic semantics emits some sequence of effects at some point, the other semantics emits a matching sequence of effects. Without these latter two lemmas, programs that run indefinitely cannot be considered to have preserved semantics in the K machine.

Lemma 5 (Divergent Soundness). *If $\epsilon \triangleright e \overset{*}{\mapsto}_K S \rightsquigarrow a$ then $e \overset{*}{\mapsto} e' \rightsquigarrow a$ for some expression e'.*

Lemma 6 (Divergent Completeness). *If $e \overset{*}{\mapsto} e' \rightsquigarrow a$ then $\epsilon \triangleright e \overset{*}{\mapsto}_K S \rightsquigarrow a$ for some state S.*

Unfortunately, while big-step semantics are the nicest way of showing Lemma 3, they have no hope of showing the diverging analogue, as big-step semantics can only describe converging computations, not diverging ones. It therefore appears that one must fall back to less favorable approaches like simulation.

For this reason, this article develops big-*stop* semantics, which can maintain the niceties of big-step-style reasoning even for diverging computations. Specifically, we show in Section 6 that Lemma 6 can be proven with straightforward rule induction. The key step is formulating and proving the big-stop analog of Lemma 4 to account for diverging computations. Alternative approaches for handling divergence are discussed in Section 9 and compared to big-stop.

4 Pure Big-Stopping

This section develops big-stop semantics to conveniently reason about potentially nonterminating programs while maintaining the ease of using big-step semantics. Moreover, this development proceeds in a minimally invasive manner, making only a small adaptation to the rules of big-step semantics. The essence of the approach is to take a standard big-step system and add just one rule schema to allow for arbitrarily stopping computation. The remainder of this section shows how to perform such a big-stop extension to our effectful, call-by-value PCF. However, nothing about the technique is intrinsically tied to PCF, so the same technique should apply to other languages. We exhibit such an extension for an imperative language in Section 8.

The fundamental idea of big-stop semantics is to add new rules to big-step semantics that enable the semantics to nondeterministically stop a computation midway through. Each of the possible

$$\frac{\forall 1 \le i \le k.\, e_i \Downarrow e_i' \qquad \forall 1 \le i < k.\, e_i'\ val \qquad \forall k+1 \le i \le n.\, e_i' = e_i}{E[e_1, \ldots, e_n] \Downarrow E[e_1', \ldots, e_n']}\ \textsc{St-Stop}(\kappa)$$

$$\frac{e \Downarrow \mathsf{Z} \qquad e_1 \Downarrow e_1'}{\mathsf{case}\{e_1; x.e_2\}[e] \Downarrow e_1'}\ \textsc{St-CaseZ}$$

$$\frac{e \Downarrow \mathsf{S}[v] \qquad v\ val \qquad [v/x]e_2 \Downarrow e_2'}{\mathsf{case}\{e_1; x.e_2\}[e] \Downarrow e_2'}\ \textsc{St-CaseS}$$

$$\frac{e_1 \Downarrow \mathsf{fun}\{f, x.\, e\} \qquad e_2 \Downarrow v_2 \qquad v_2\ val \qquad [\mathsf{fun}\{f, x.\, e\}/f, v_2/x]e_2 \Downarrow e'}{\mathsf{app}[e_1, e_2] \Downarrow e'}\ \textsc{St-App}$$

Fig. 5. Big-stop semantics for pure call-by-value PCF

resulting partially-computed expressions corresponds to computing only a prefix of the complete sequence of computation steps. Even if the complete computation sequence is infinite because the computation is nonterminating, its prefixes will be finite and inductively capturable. Capturing all such prefixes of computation is sufficient to describe both terminating and nonterminating computations uniformly—terminating computations will simply have finitely many prefixes, one of which ends with a value. These prefixes also allow big-stop semantics to rival its coinductive alternatives in reasoning about infinite computations, in the same way that infinite streams can be equivalently reasoned about directly using coinduction or via their prefixes using induction.

When adapting a big-step system to big-stop, it may also be necessary to add explicit premisses to existing big-step rules to ensure certain expressions are values. In the original big-step system, such expressions would necessarily be values, and thus such premisses are optional in the big-step system, while they are required in the big-stop extension. The details of this extension are exemplified for pure PCF in the following paragraphs. Effectful PCF is covered in Section 5, and Section 7 shows techniques that further minimize the changes required for this kind of extension.

Formally, the big-stop relation is given by the judgment $e \Downarrow e'$, which means that the expression e partially evaluates to the expression e'. The rules for this judgment are given in Figure 5 and described in the following paragraphs. Most of the rules correspond to Figure 4's big-step rules. Some examples of the use of the big-stop rules can be found in Section B.

The key new feature in Figure 5 is the rule schema St-Stop(κ), which represents the core idea of nondeterministically halting evaluation. This rule allows any expression $E[e_1, \ldots, e_n]$ to stop evaluating after reducing only some of its subexpressions e_i. In accordance with the evaluation order of PCF, these subexpressions must be reduced from left to right so that e_i is reduced to a value when $i < k$ and is left untouched when $i > k$—only e_k may be not fully evaluated. Moreover, when instantiated at $k = 0$, this rule schema allows all expressions to immediately stop evaluation by big-stopping to themselves, subsuming the role of B-Val, which already halted computation for values. Finally, this rule schema is suggestively similar to the *small*-step rule schema S-Seq(κ) from Figure 2, hinting at the role it plays for big-stop semantics.

To be explicit, the schema St-Stop(κ) stands in for the following five rules:

$$\frac{}{e \Downarrow e}\ \textsc{St-Stop} \qquad \frac{e \Downarrow e'}{\mathsf{S}[e] \Downarrow \mathsf{S}[e]'}\ \textsc{St-Succ} \qquad \frac{e \Downarrow e'}{\mathsf{case}\{e_1; x.e_2\}[e] \Downarrow \mathsf{case}\{e_1; x.e_2\}[e']}\ \textsc{St-Case}$$

$$\frac{e_1 \Downarrow e_1'}{\mathsf{app}[e_1, e_2] \Downarrow \mathsf{app}[e_1', e_2]}\ \textsc{St-App1} \qquad \frac{e_1 \Downarrow v_1 \qquad v_1\ val \qquad e_2 \Downarrow e_2'}{\mathsf{app}[e_1, e_2] \Downarrow \mathsf{app}[v_1, e_2']}\ \textsc{St-App2}$$

Aside from St-Stop(κ), the other rules are nearly unchanged from their big-step counterparts. However, some additional premisses are added to ensure that certain subexpressions are values.

For example, in ST-CASES, the premiss $v\ \ val$ is added to ensure that e fully evaluates to $\mathsf{S}[v]$. This additional premiss is necessary to ensure that partial evaluations of e are not prematurely plugged into the case expression. In big-*step* semantics, such a premiss is unnecessary because the dynamics already ensure that $\mathsf{S}[v]$ is a value. As big-*stop* semantics allows for partial evaluations, this invariant is no longer present. Thus the new value premisses do not comprise new restrictions for the semantic rules, but rather explicitly maintain pre-existing restrictions. An alternative formulation of big-stop semantics could simply use a big-step judgment for these premisses to maintain the same invariant, but it requires fewer rules overall to have only one kind of judgment.

One subtlety of the big-stop extension is that, while a big-step system may be deterministic, big-stop is always nondeterministic—every expression can either reduce as normal or halt. However, this nondeterminism is as benign as the multi-step judgment's nondeterminism—each can only relate expressions to partial evaluations that arise during typical evaluation (regardless of whether that typical evaluation is deterministic or not). Moreover, this nondeterminism poses no problems for proving with big-stop semantics, as exemplified by the Agda proofs referenced in Sections 5 and 6 [27], which each maintain the straightforward nature of their big-step counterparts.

4.1 Properties

The following property, Theorem 7, is the key to showing that big-stop semantics captures precisely the notion we are interested in. That is, it shows big-stop semantics captures precisely the notion of evaluation typically built upon small-step semantics.

THEOREM 7 (STOP/MULTI EQUIVALENCE). *For all expressions* e, e', $\quad e \Downarrow e' \iff e \overset{*}{\mapsto} e'$

PROOF. This equivalence follows by induction on the derivation of each judgment. □

The key benefit of big-stop semantics is not that it infers any new relations beyond the multi-step judgment, but rather that it allows the derivation of the same multi-step relations in a manner more ergonomic for use in proofs. From Theorem 7, it immediately follows that any property one might state using multi-step semantics can be equivalently stated using big-stop semantics. Already, this result covers most of the gap between big- and small-step semantics, and the only cost is a minor extension to standard big-step operational semantics.

In light of Theorem 7, one should also expect that big-stopping indeed captures stopped evaluations. This property can be formally expressed with the following statement, which says that stopped expressions could continue to be evaluated as if they had not stopped. That is, the big-stop relation is transitive just like the multi-step relation.

LEMMA 8 (TRANSITIVITY). *If* $e_1 \Downarrow e_2$ *and* $e_2 \Downarrow e_3$*, then* $e_1 \Downarrow e_3$.

PROOF. This property follows from induction over the derivations. □

An important observation is that the big-stop system does not interfere with standard big-step evaluation to values. Call a derivation *strict* if every use of the rule schema ST-STOP(κ) in that derivation has the conclusion $v \Downarrow v$ for some value v. Then there is a canonical isomorphism between strict derivations of $e \Downarrow v$ and derivations of $e \Downarrow v$ where each rule ST-X corresponds to B-X, except for ST-STOP(0) and ST-STOP(1), which correspond to B-VAL and B-SUCC, respectively.

LEMMA 9 (DERIVATION ISOMORPHISM). *For all expressions* e *and values* v*, strict derivations of* $e \Downarrow v$ *are isomorphic to derivations of* $e \Downarrow v$.

PROOF. This property follows by induction over the structure of the derivations. □

COROLLARY 10 (STOP/BIG EQUIVALENCE). *For all expressions* e *and values* v, $e \Downarrow v \iff e \Downarrow v$.

4.2 Progress and Preservation

While multi-step judgments are commonly used incarnations of small-step semantics, there still exist programming language properties that are traditionally stated using small-step semantics simpliciter, without any multi-stepping. In particular, progress and preservation are typically stated this way. The rest of this section is devoted to recovering these statements in the big-stop system.

The preservation property can be restated using big-stop semantics as follows:

Theorem 11 (Preservation). *Big-stopping preserves typing.*
That is, if $\cdot \vdash e : \tau$ *and* $e \Downarrow e'$*, then* $\cdot \vdash e' : \tau$.

Proof. This property can be shown by induction on the derivation of the big-stop judgment. □

Note that preservation could also have been proven by first observing that the same property holds for multi-stepping and then applying Theorem 7. However, it is not necessary to make such an indirect proof; using big-stop directly is unproblematic.

With a little additional maneuvering, one can also restate the property of progress. To do so, call all of the rules "progressing" except for St-Stop(κ). Further, call a big-stop derivation "progressing" if it uses a progressing rule. We use the notation $e \downarrow e'$ to mean that $e \Downarrow e'$ is the conclusion of a progressing derivation. We can then state a big-stop version of the progress property as follows:

Theorem 12 (Progress). *Well-typed, non-value expressions can progress.*
That is, if $\cdot \vdash e : \tau$*, then either* e *val or* $e \downarrow e'$ *for some* e'.

Proof. This property follows from induction over the structure of the expression e. □

In other words, Theorem 12 ensures that well-typed, non-value expressions e can make at least one small-step of evaluation progress to some e'. To confirm that Theorem 12 really does capture the key notion of progress, we also point out the following lemma:

Lemma 13 (Progressing Means Progress). *If* $e_1 \downarrow e_2$*, then* $\exists e_3.\, e_1 \mapsto e_3$ *and* $e_3 \overset{*}{\mapsto} e_2$.

Proof. This property follows from induction over the derivation of $e_1 \downarrow e_2$. □

As a result, progressing derivations are the correct big-stop notion for making evaluation progress. Thus, e.g., progressing derivations can be made arbitrarily deep iff small-step reduction loops forever. For convenience, we use the notation $e \Downarrow \infty$ to mean that such arbitrarily deep progressing derivations can be made, i.e., that e diverges. This notation will help to compare to other systems for inferring divergence in Section 9, but will not otherwise be used in the big-stop inference system.

5 Effectful Big-Stopping

This section develops big-stop semantics with effects. Here, we specifically consider writer-monad-style effects that are emitted but do not otherwise affect computation, like printing. Such a system can be used to solve the effectful compilation problem presented in Section 3. For effects that may interact with computation, like mutation, see Section 8.

In this section, we also prove all provided properties in Agda in an intrinsically typed setting [27]. This formalization shows that big-stop semantics makes good on its ergonomic promise: these proofs largely follow by straightforward induction on the derivation of the big-stop judgment and do not need the detours required by small-step systems. Moreover, these proofs work in an effectful setting with nontermination, which previously was a blocker for the use of big-step semantics.

To introduce writer-monad-style effects, we extend PCF syntax with the expression $\texttt{eff}\{a; e\}$, which emits the effect indicated by a and then continues on as e. To capture these effects, we

$$\textsc{SE-Seq}(\kappa)\quad \frac{1 \le k \le n \qquad \forall\, 1 \le i < k.\, e_i\ \mathit{val} \qquad e_k \mapsto e_k' \rightsquigarrow a \qquad \forall\, i \neq k.\, e_i = e_i'}{E[e_1, \ldots, e_n] \mapsto E[e_1', \ldots, e_n'] \rightsquigarrow a}$$

$$\textsc{SE-CaseZ}\quad \frac{}{\mathsf{case}\{e_1; x.e_2\}[\mathsf{Z}] \mapsto e_1 \rightsquigarrow 1}$$

$$\textsc{SE-CaseS}\quad \frac{v\ \mathit{val}}{\mathsf{case}\{e_1; x.e_2\}[\mathsf{S}[v]] \mapsto [v/x]e_2 \rightsquigarrow 1}$$

$$\textsc{SE-App}\quad \frac{v\ \mathit{val}}{\mathsf{app}[\mathsf{fun}\{f, x.\, e\}, v] \mapsto [\mathsf{fun}\{f, x.\, e\}/f,\, v/x]e \rightsquigarrow 1}$$

$$\textsc{SE-Effect}\quad \frac{}{\mathsf{eff}\{a; e\} \mapsto e \rightsquigarrow a} \qquad \textsc{ME-Refl}\quad \frac{}{e \overset{*}{\mapsto} e \rightsquigarrow 1} \qquad \textsc{ME-Step}\quad \frac{e_1 \mapsto e_2 \rightsquigarrow a \qquad e_2 \overset{*}{\mapsto} e_3 \rightsquigarrow b}{e_1 \overset{*}{\mapsto} e_3 \rightsquigarrow ab}$$

Fig. 6. Small- and multi-step semantics for effectful call-by-value PCF

$$\textsc{BE-Succ}\quad \frac{e \Downarrow v \rightsquigarrow a}{\mathsf{S}[e] \Downarrow \mathsf{S}[v] \rightsquigarrow a} \qquad \textsc{BE-CaseZ}\quad \frac{e \Downarrow \mathsf{Z} \rightsquigarrow a \qquad e_1 \Downarrow v_1 \rightsquigarrow b}{\mathsf{case}\{e_1; x.e_2\}[e] \Downarrow v_1 \rightsquigarrow ab}$$

$$\textsc{BE-CaseS}\quad \frac{e \Downarrow \mathsf{S}[v] \rightsquigarrow a \qquad [v/x]e_2 \Downarrow v_2 \rightsquigarrow b}{\mathsf{case}\{e_1; x.e_2\}[e] \Downarrow v_2 \rightsquigarrow ab}$$

$$\textsc{BE-App}\quad \frac{e_1 \Downarrow \mathsf{fun}\{f, x.\, e\} \rightsquigarrow a \qquad e_2 \Downarrow v_2 \rightsquigarrow b \qquad [\mathsf{fun}\{f, x.\, e\}/f, v_2/x]e \Downarrow v \rightsquigarrow c}{\mathsf{app}[e_1, e_2] \Downarrow v \rightsquigarrow abc} \qquad \textsc{BE-Val}\quad \frac{v\ \mathit{val}}{v \Downarrow v \rightsquigarrow 1} \qquad \textsc{BE-Eff}\quad \frac{e \Downarrow v \rightsquigarrow b}{\mathsf{eff}\{a; e\} \Downarrow v \rightsquigarrow ab}$$

Fig. 7. Big-step semantics for effectful call-by-value PCF

consider sequences of them in the order they occur. Thus, for example, the effect term abc just means that effect a is followed by effect b and then effect c. We also use 1 to represent the identity effect, which can be thought of as an uninteresting effect or no effect at all.

Such effects can be incorporated into the small-step semantics using the judgment $e_1 \mapsto e_2 \rightsquigarrow a$, meaning that e_1 small-steps to e_2 and has the effects of a. The rules for this system are given in Figure 6. The figure also defines the multi-step variant of this judgment, $e_1 \overset{*}{\mapsto} e_3 \rightsquigarrow a$.

The corresponding effectful big-step semantics are given by the rules of Figure 7. These rules define the judgment $e \Downarrow v \rightsquigarrow a$ meaning that the expression e evaluates to the value v while emitting the effects indicated by a. To ensure that this system agrees with the small-step system of Figure 6, we provide Lemma 14, the effectful analogue of Theorem 1.

Lemma 14 (Effectful Big/Multi Equivalence). *For all expressions e, values v, and effects a,*

$$e \Downarrow v \rightsquigarrow a \iff e \overset{*}{\mapsto} v \rightsquigarrow a$$

Proof. This property follows by induction over the structure of the derivations. □

With a proper big-step system in hand, we can now go over the big-stop extension. Fundamentally, the essence of big-stop semantics remains the same in this new setting: extend big-step semantics with a rule schema for stopping (and possibly insert some value premisses, depending on the exact formulation of the big-step system). The resulting adaptation for the big-step rules of Figure 7 yields the big-stop rules of Figure 8. Examples using these rules can be found in Section B.

$$\textsc{StE-Stop}(\kappa)\ \frac{\forall 1 \le i \le k.\, e_i \Downarrow e_i' \rightsquigarrow a_i \qquad \forall 1 \le i \le k-1.\, e_i'\ \mathit{val} \qquad \forall k+1 \le i \le n.\, e_i' = e_i}{E[e_1, \ldots, e_n] \Downarrow E[e_1', \ldots, e_n'] \rightsquigarrow a_1 \ldots a_k}$$

$$\textsc{StE-CaseZ}\ \frac{e \Downarrow \mathsf{Z} \rightsquigarrow a \qquad e_1 \Downarrow e_1' \rightsquigarrow b}{\mathsf{case}\{e_1; x.e_2\}[e] \Downarrow e_1' \rightsquigarrow ab}$$

$$\textsc{StE-CaseS}\ \frac{e \Downarrow \mathsf{S}[v] \rightsquigarrow a \qquad v\ \mathit{val} \qquad [v/x]e_2 \Downarrow e_2' \rightsquigarrow b}{\mathsf{case}\{e_1; x.e_2\}[e] \Downarrow e_2' \rightsquigarrow ab}$$

$$\textsc{StE-App}\ \frac{e_1 \Downarrow \mathsf{fun}\{f, x.\, e\} \rightsquigarrow a \qquad e_2 \Downarrow v_2 \rightsquigarrow b \qquad v_2\ \mathit{val} \qquad [\mathsf{fun}\{f, x.\, e\}/f, v_2/x]e_2 \Downarrow e' \rightsquigarrow c}{\mathsf{app}[e_1, e_2] \Downarrow e' \rightsquigarrow abc}$$

$$\textsc{StE-Eff}\ \frac{e \Downarrow e' \rightsquigarrow b}{\mathsf{eff}\{a; e\} \Downarrow e' \rightsquigarrow ab}$$

Fig. 8. Big-stop semantics for effectful call-by-value PCF

5.1 Properties

The big-stop system can be verified to properly extend the given big-step system using the following lemma, which is analogous to Lemma 9 and Theorem 10.

THEOREM 15 (EFFECTFUL STOP/STEP EQUIVALENCE 🐓). *For all expressions e, values v, and effects a,*

$$e \Downarrow v \rightsquigarrow a \iff e \Downarrow v \rightsquigarrow a$$

PROOF. This property follows from induction over the derivations. □

More importantly, as described by Theorem 16, the effectful big-stop judgment coincides with the effectful multi-step judgment, just as in the pure setting's Theorem 7.

THEOREM 16 (EFFECTFUL STOP/MULTI EQUIVALENCE 🐓). *For all expressions e, e' and effects a,*

$$e \Downarrow e' \rightsquigarrow a \iff e \overset{*}{\mapsto} e' \rightsquigarrow a$$

PROOF. This property follows from induction over the derivations. □

As might be expected in light of Theorem 16, various other properties also translate cleanly into the effectful big-stop setting. For example, transitivity and progress both hold, as stated below.

LEMMA 17 (EFFECTFUL TRANSITIVITY 🐓). *If $e_1 \Downarrow e_2 \rightsquigarrow a$ and $e_2 \Downarrow e_3 \rightsquigarrow b$, then $e_1 \Downarrow e_3 \rightsquigarrow ab$*

PROOF. This property follows from induction over the derivations. □

We formalize progress with the judgment $e \downarrow e' \rightsquigarrow a$, which means that $e \Downarrow e' \rightsquigarrow a$ is the conclusion of a progressing derivation (a derivation using a rule other than StE-Stop(κ)).

THEOREM 18 (EFFECTFUL PROGRESS 🐓). *Well-typed, non-value expressions can progress. That is, if $\cdot \vdash e : \tau$, then either e val or $e \downarrow e' \rightsquigarrow a$ for some e' and a.*

PROOF. This property follows from induction over the structure of the expression e. □

All of these results are proved in a direct way by induction on the derivation of big-stop, big-step, small-step, or typing judgments. The simplicity of these proofs is reflected in their compact formalizations in Agda [27], as shown in Table 1.

6 Application: K Machine Correctness

Now that big-stop semantics has been established, we can demonstrate its benefits by proving semantics preservation for the K machine compilation of effectful call-by-value PCF, as outlined in Section 3. The results of this section have been proved in Agda [27].

	Lemma 14	Theorem 15	Theorem 16	Lemma 17	Theorem 18
Lines of code in Agda	64	28	88	30	18

Table 1. Size of Agda formalizations for each theorem, measured by non-comment, non-blank lines of code

$$\textsc{Stack-Empty}\quad \frac{}{\epsilon\ \mathit{stack}} \qquad \textsc{Stack-Frame}\quad \frac{k\ \mathit{stack} \qquad f\ \mathit{frame}}{k;f\ \mathit{stack}}$$

$$\textsc{F-Succ}\quad \frac{}{\langle \mathsf{S}[\,-\,]\rangle\ \mathit{frame}} \qquad \textsc{F-Case}\quad \frac{}{\langle \mathsf{case}\{e_1; x.e_2\}[-]\rangle\ \mathit{frame}} \qquad \textsc{F-Fun}\quad \frac{}{\langle \mathsf{app}[-,e]\rangle\ \mathit{frame}} \qquad \textsc{F-Arg}\quad \frac{v\ \mathit{val}}{\langle \mathsf{app}[v,-]\rangle\ \mathit{frame}}$$

Fig. 9. K machine stack and frames

6.1 The Effectful K Machine

A K machine state is either of the form $k \triangleright e$ or $k \triangleleft v$ where k is a *stack* of *frames*, e is a PCF expression, and v is a value. K machine states of the form $k \triangleright e$ indicate that the expression e is not yet a value and needs to be evaluated, and K machine states of the form $k \triangleleft v$ indicate that the value v can be plugged into the top frame of the stack k.

Stacks k and frames f are picked out by the judgments $k\ \ \mathit{stack}$ and $f\ \ \mathit{frame}$, respectively, described in Figure 9. Each frame is just an expression with a hole in it indicating the subexpression undergoing evaluation. The rule F-Arg is notable because it has a premiss requiring the expression v to be a value. Strictly speaking, this premiss is not necessary, as the K machine can only generate such frames where the expression v is a value. Thus, the value premiss only serves to exclude invalid stacks that could not arise anyway. This exclusion makes the value premiss useful in the course of proving properties of the K machine, since impossible stacks need not be considered.

The K machine's transitions are formalized with Figure 10. These rules define the judgment $S \mapsto_K T \rightsquigarrow a$ to mean that K machine state S transitions in one step to state T while emitting the effects indicated by a. This figure also defines the multi-step analogue $S \overset{*}{\mapsto}_K T \rightsquigarrow a$. To obtain transitions for the pure K machine, simply ignore the effects and the rule KE-Eff.

6.2 Proving Correctness

Semantic preservation of the compilation of effectful PCF to the K machine can be boiled down to the following four lemmas. In particular, soundness and completeness must hold for both converging and diverging computations, the latter of which is necessary due to the presence of effects. The rest of this subsection will briefly discuss their proofs, all of which have been formalized in Agda [27].

Lemma 19 (Convergent Soundness). *If* $\epsilon \triangleright e \overset{*}{\mapsto}_K \epsilon \triangleleft v \rightsquigarrow a$ *then* $e \Downarrow v \rightsquigarrow a$.

Lemma 20 (Divergent Soundness). *If* $\epsilon \triangleright e \overset{*}{\mapsto}_K S \rightsquigarrow a$ *then* $e \Downarrow e' \rightsquigarrow a$ *for some* e'

Lemma 21 (Convergent Completeness). *If* $e \Downarrow v \rightsquigarrow a$ *and* v *val then* $\epsilon \triangleright e \overset{*}{\mapsto}_K \epsilon \triangleleft v \rightsquigarrow a$.

Lemma 22 (Divergent Completeness). *If* $e \Downarrow e' \rightsquigarrow a$ *then* $\epsilon \triangleright e \overset{*}{\mapsto}_K S \rightsquigarrow a$ *for some state* S.

$$\frac{}{k \rhd \mathsf{Z} \mapsto_K k \lhd \mathsf{Z} \rightsquigarrow 1}\ \textsc{KE-Zero} \qquad \frac{}{k \rhd \mathsf{S}[e] \mapsto_K k;\langle \mathsf{S}[\,-\,]\rangle \rhd e \rightsquigarrow 1}\ \textsc{KE-Succ1} \qquad \frac{}{k;\langle \mathsf{S}[\,-\,]\rangle \lhd v \mapsto_K k \lhd \mathsf{S}[v] \rightsquigarrow 1}\ \textsc{KE-Succ2}$$

$$\frac{}{k \rhd \mathsf{case}\{e_1; x.e_2\}[e_3] \mapsto_K k;\langle \mathsf{case}\{e_1; x.e_2\}[-]\rangle \rhd e_3 \rightsquigarrow 1}\ \textsc{KE-Case} \qquad \frac{}{k;\langle \mathsf{case}\{e_1; x.e_2\}[-]\rangle \lhd \mathsf{Z} \mapsto_K k \rhd e_1 \rightsquigarrow 1}\ \textsc{KE-CaseZ}$$

$$\frac{}{k;\langle \mathsf{case}\{e_1; x.e_2\}[-]\rangle \lhd \mathsf{S}[v] \mapsto_K k \rhd [v/x]e_2 \rightsquigarrow 1}\ \textsc{KE-CaseS} \qquad \frac{}{k \rhd \mathsf{fun}\{f, x.\,e\} \mapsto_K k \lhd \mathsf{fun}\{f, x.\,e\} \rightsquigarrow 1}\ \textsc{KE-Fun}$$

$$\frac{}{k \rhd \mathsf{app}[e_1, e_2] \mapsto_K k;\langle \mathsf{app}[-, e_2]\rangle \rhd e_1 \rightsquigarrow 1}\ \textsc{KE-App1} \qquad \frac{}{k;\langle \mathsf{app}[-, e]\rangle \lhd v \mapsto_K k;\langle \mathsf{app}[v, -]\rangle \rhd e \rightsquigarrow 1}\ \textsc{KE-App2}$$

$$\frac{}{k;\langle \mathsf{app}[\mathsf{fun}\{f, x.\,e\}, -]\rangle \lhd v \mapsto_K k \rhd [\mathsf{fun}\{f, x.\,e\}/f, v/x]e \rightsquigarrow 1}\ \textsc{KE-App3} \qquad \frac{}{k \rhd \mathsf{eff}\{a; e\} \mapsto_K k \rhd e \rightsquigarrow a}\ \textsc{KE-Eff}$$

$$\frac{}{S \overset{*}{\mapsto}_K S \rightsquigarrow 1}\ \textsc{KEM-Refl} \qquad \frac{S \mapsto_K T \rightsquigarrow a \qquad T \overset{*}{\mapsto}_K U \rightsquigarrow b}{S \overset{*}{\mapsto}_K U \rightsquigarrow ab}\ \textsc{KEM-Step}$$

Fig. 10. Effectful K machine transition rules

Lemmas 19 and 20 (soundness) can both be proven using big-stop semantics in a similar manner to small-step semantics, as they proceed by induction over the K machine dynamics rather than the PCF dynamics. Big-stop semantics is just as fit a target for these proofs as small-step semantics.

Lemma 21 (convergent completeness) is straightforward to prove using big-stop semantics, just as it was for big-step as shown in Section 3. Because big-stop semantics are just a small extension of big-step semantics, the thrust of the proof of Lemma 21 is practically identical to the easy big-step approach of Lemma 3.

Finally, the proof of Lemma 22 (divergent completeness) proceeds by building upon Lemma 21. To extend Lemma 21 to cover nontermination, the new proof just needs a new case for each of the rules following from the schema StE-Stop(κ). Each of these new cases is straightforward and proceeds much the same as a typical big-step case. The whole proof of this lemma is therefore be proved directly with rule induction, unlike when using small- or big-step semantics.

Altogether, these proofs show that big-stop semantics is more useful than the big-step semantics it extends, and it also can provide a superior proving experience as compared to small-step semantics. Moreover, as these proofs concern the correctness of compilation, we expect the benefits of big-stop semantics to be widely applicable. Finally, these proofs demonstrate that the nondeterministic nature of big-stop semantics pose no issue for formalization.

In Table 2, we compare the sizes of the Agda proofs of the big-stop soundness and completeness lemmas (Lemmas 19 to 22) against the equivalent lemmas formulated using big-step and small-step semantics. That is to say, we compare the sizes of proving the soundness and completeness of the K machine with respect to small-step, big-step, and big-stop semantics.

Table 2's results demonstrate that the big-stop proofs of soundness and convergent completeness are each similar in size to traditional big-step proofs of the corresponding convergent properties. Recall that big-step simply cannot handle divergent properties, and so those table entries are left blank. The only place big-stop has a larger proof, divergent completeness, is because the divergent

	Soundness		Completeness	
	Convergent Lemma 19	**Divergent** Lemma 20	**Convergent** Lemma 21	**Divergent** Lemma 22
Big-stop	76	72	32	72
Big-step	81	–	29	–
Small-step	88	84	68	128

Table 2. Comparison of Agda proof sizes, measured by non-comment, non-blank lines of code

proof builds upon the convergent one, so the lines of the big-stop convergent completeness lemma are also counted toward divergent completeness.

Table 2 also shows that traditional small-step proofs of soundness are around the same size as their big-step and -stop counterparts, which is to be expected because these proofs induct over K machine dynamics rather than PCF dynamics. Small-step completeness is where the onerousness of small-step semantics comes into play. Not only are these proofs around twice as large as big-step/stop, but the proofs themselves depend on the big-step/stop proofs. The shortest proofs we could find for small-step completeness involve translating small-step into big-step/stop and then invoking the completeness of big-step/stop. Around half of the lines attributed to these proofs are for the translation and the other half are for big-step/stop completeness. It is not clear that there is any better way to prove these properties using small-step semantics, and it is telling that big-stop semantics seems to be the best way to prove divergent completeness.

6.3 The Big-Stop Method

Lemmas 21 and 22 exemplify a general proof technique for generalizing a theorem about converging computations to cover diverging ones too. We call this technique the *big-stop method*. The starting point is a theorem that has been proved by induction on a standard big-step judgment. Here the appropriate theorem is the effectful version of Lemma 4: If $e \Downarrow v \rightsquigarrow a$ then $k \triangleright e \mapsto_K k \triangleleft v \rightsquigarrow a$.

The methodical generalization to diverging computations proceeds in three steps. First, extend the big-step judgment to big-stop by adding the appropriate stopping rules. Second, reproduce the proof of the original theorem using the new big-stop judgment, yielding a proof concerning values (Lemma 21). Third, add induction cases for the new stopping rules, building upon the previous proof (Lemma 22). All together, these steps result in the generalized theorem for partial evaluations.

The key—which demonstrates the advantage of big-stop over small-step—is that the generalized theorem is void of complicated inductive invariants. In Lemma 22, the example at hand, this materializes in the relationship (or lack thereof) between the partially-evaluated expression e' and state S. In a multi-step setting, one would need to relate e' and S because they would eventually occur on the left-hand side of relations where they would be inductively consumed. An invariant would then be necessary to ensure that the correct properties are maintained through this consumption. In contrast, the big-stop system never consumes any expression it puts out unless that expression is a value, which is much simpler to deal with than an arbitrary partially-evaluated expression—in fact, it can be dispatched with the theorem for values (here Lemma 21), with no complicated invariant needed. Thus, the proof of the generalized theorem can proceed by straightforward induction.

7 Further Ergonomic Optimization

This section presents some additional variants of big-stop systems that further minimize the extent of the change from a typical big-step system. As set up previously in the paper, big-stop semantics

$$v ::= x \mid \texttt{fun}\{f, x.\, e\} \mid \texttt{Z} \mid \texttt{S}[v] \qquad e ::= v \mid \texttt{case}\{e_1; x.e_2\}[v] \mid \texttt{app}[v_1, v_2] \mid \texttt{let}\, x \;=\; e_1 \,\texttt{in}\, e_2$$

(a) Variables and values (b) Expressions

Fig. 11. Language of monadic normal PCF

$$\textsc{SM-Let1}\ \frac{e_1 \mapsto e_1'}{\texttt{let}\, x \;=\; e_1 \,\texttt{in}\, e_2 \mapsto \texttt{let}\, x \;=\; e_1' \,\texttt{in}\, e_2} \qquad \textsc{SM-Let2}\ \frac{v\ \mathit{val}}{\texttt{let}\, x \;=\; v \,\texttt{in}\, e \mapsto [v/x]e} \qquad \textsc{SM-CaseZ}\ \frac{}{\texttt{case}\{e_1; x.e_2\}[\texttt{Z}] \mapsto e_1}$$

$$\textsc{SM-CaseS}\ \frac{}{\texttt{case}\{e_1; x.e_2\}[\texttt{S}[v]] \mapsto [v/x]e_2} \qquad \textsc{SM-App}\ \frac{}{\texttt{fun}\{f, x.\, e\}\ v \mapsto [\texttt{fun}\{f, x.\, e\}/f,\ v/x]e}$$

Fig. 12. Small-step semantics for monadic normal PCF

already essentially requires only one new rule schema, but this schema is most easily dispatched in practice by replacing it with a rule for each of its instantiations—we take this approach with our Agda proofs. This replacement results in a number of additional rules proportional to the number of language constructs, and each of these rules needs its own proof case. This ballooning of proof obligations when using big-stop semantics can be tedious, even if each new case is easy to dispatch.

With fewer changes, the proof obligations of such a system are closer to those of the original big-step system. These obligations can be optimized to a point where one practically gets the nonterminating results of big-stop semantics for free after establishing the standard big-step result. At the limit, big-stop proofs only require extending big-step proofs with one easy proof case.

7.1 Normal Form

One way to minimize the number of rules required in big-stop semantics is to only consider languages that are in monadic normal form [20, 25] (a close relative of A-normal form [16, 42]). Monadic normal form requires that all possible subexpressions are either values or let-bound variables. This normalization does not affect expressivity of the language. An example of monadic normal form PCF expressions is given by the grammar of e in Figure 11.

Monadic normal form simplifies the dynamics of computation by removing most congruences. Instead, monadic normal form uses let-expressions to explicitly sequence computations in a syntactic way. This approach means the only congruence rule needed is that for the let-expression itself. See the small-step semantics of Figure 12 where the only congruence rule is SM-Let1.

Both the small-step semantics of Figure 12 and the corresponding big-step semantics Figure 13 also exhibit another simplification of monadic normal form: fewer premisses are required. Because monadic normal form forces subexpressions to be values as much as possible, no premisses need to be spent on their evaluation. Compare Figures 12 and 13 to their counterparts Figures 2 and 4.

The appropriate big-stop extension of Figure 13 is given by Figure 14, which only contains one more rule in total. This extension allows stopping at any point with the two rules StM-Stop and StM-Let1. The rule StM-Stop subsumes BM-Val much like St-Stop(0) subsumes B-Val, and the rule StM-Let1 is the single congruence rule needed to propagate stopping. The rule StM-Let2 adds a value premiss to the rule BM-Let, just as was done to many rules in previous big-stop extensions. This time, however, *only* the rule StM-Let2 needs a new value premiss, and the rest of the rules can

$$\frac{v\ val}{v \Downarrow v}\ \text{BM-Val} \qquad \frac{e_1 \Downarrow v_1 \qquad [v_1/x]e_2 \Downarrow v_2}{\mathtt{let}\,x\ =\ e_1\ \mathtt{in}\,e_2 \Downarrow v_2}\ \text{BM-Let}$$

$$\frac{e_1 \Downarrow v_1}{\mathtt{case}\{e_1; x.e_2\}[\mathtt{Z}] \Downarrow v_1}\ \text{BM-CaseZ} \qquad \frac{[v/x]e_2 \Downarrow v_2}{\mathtt{case}\{e_1; x.e_2\}[\mathtt{S}[v]] \Downarrow v_2}\ \text{BM-CaseS} \qquad \frac{[\mathtt{fun}\{f, x.\,e\}/f, v/x]e \Downarrow v'}{\mathtt{app}[\mathtt{fun}\{f, x.\,e\}, v] \Downarrow v'}\ \text{BM-App}$$

Fig. 13. Big-step semantics for monadic normal PCF

$$\frac{}{e \Downarrow e}\ \text{StM-Stop} \qquad \frac{e_1 \Downarrow e_1'}{\mathtt{let}\,x\ =\ e_1\ \mathtt{in}\,e_2 \Downarrow \mathtt{let}\,x\ =\ e_1'\ \mathtt{in}\,e_2}\ \text{StM-Let1} \qquad \frac{e_1 \Downarrow v_1 \qquad v_1\ val \qquad [v_1/x]e_2 \Downarrow e_2'}{\mathtt{let}\,x\ =\ e_1\ \mathtt{in}\,e_2 \Downarrow e_2'}\ \text{StM-Let2}$$

$$\frac{e_1 \Downarrow e_1'}{\mathtt{case}\{e_1; x.e_2\}[\mathtt{Z}] \Downarrow e_1'}\ \text{StM-CaseZ} \qquad \frac{[v/x]e_2 \Downarrow e_2'}{\mathtt{case}\{e_1; x.e_2\}[\mathtt{S}[v]] \Downarrow e_2'}\ \text{StM-CaseS} \qquad \frac{[\mathtt{fun}\{f, x.\,e\}/f, v/x]e \Downarrow e'}{\mathtt{app}[\mathtt{fun}\{f, x.\,e\}, v] \Downarrow e'}\ \text{StM-App}$$

Fig. 14. Big-stop semantics for monadic normal PCF

remain totally untouched. The rules StM-Let1 and StM-Let2 together are respectively analogous to the rules SM-Let1 and SM-Let2 of Figure 12, much like how St-Stop(κ) is analogous to S-Seq(κ).

The key takeaway of using monadic normal form is that the unwieldy rule schema for sequencing can be replaced with rules for let-bindings rather than rules for every possible congruence. The resulting system overall has just one more rule in total than its big-step base and lightly touches two existing rules (BM-Val is loosened to StM-Stop and BM-Let gains an explicit value premiss as StM-Let2). Therefore, rather than dispatch a rule schema like St-Stop(κ) by roughly doubling the number of rules to consider in proof cases, monadic normal form enables one to only consider a fixed number of new rules. This accounting applies to any language that can be put into the appropriate monadic normal form, not just PCF.

7.2 Evaluation Contexts

A similar effect to using monadic normal form can be induced via adapting big-step semantics to evaluation contexts. Such an approach reduces the big-stop extension to just one rule replacement.

The appropriate meta-syntax for PCF's evaluation contexts is given by C in the following grammar, where $\langle\ \rangle$ is a hole and where e and v are expressions and values as defined in Figure 1. This meta-syntax allows us to write $C\langle e\rangle$ to represent the expression that results from replacing the hole in C with e, so that, e.g., $\mathtt{S}[\langle\ \rangle]\langle\mathtt{Z}\rangle = \mathtt{S}[\mathtt{Z}]$.

$$C\ ::=\ \langle\ \rangle \mid \mathtt{S}[C] \mid \mathtt{case}\{e; x.e\}[C] \mid \mathtt{app}[C, e] \mid \mathtt{app}[v, C]$$

The rules of Figure 15 save for EC-Stop then yield an evaluation context system equivalent to big-step semantics. Replacing EC-Val with EC-Stop yields the corresponding big-stop semantics.

In this formulation, the rules EC-CaseZ, EC-CaseS, and EC-App are almost the same as their monadic normal counterparts. The only difference is that certain subexpressions are no longer syntactically guaranteed to be values and so require value premisses. These rules therefore function in the same way as their counterparts, describing how to compute redexes.

$$\frac{}{e \Downarrow e}\ \text{EC-Stop} \qquad \frac{v\ val}{v \Downarrow v}\ \text{EC-Val} \qquad \frac{e_1 \Downarrow e_1' \quad C\langle e_1' \rangle \Downarrow e_2}{C\langle e_1 \rangle \Downarrow e_2}\ \text{EC-Seq}$$

$$\frac{e_1 \Downarrow e_1'}{\mathsf{case}\{e_1; x.e_2\}[\mathsf{Z}] \Downarrow e_1'}\ \text{EC-CaseZ} \qquad \frac{v\ val \quad [v/x]e_2 \Downarrow e_2'}{\mathsf{case}\{e_1; x.e_2\}[\mathsf{S}[v]] \Downarrow e_2'}\ \text{EC-CaseS} \qquad \frac{v_2\ val \quad [\mathsf{fun}\{f, x.\, e\}/f,\, v/x]e \Downarrow e'}{\mathsf{app}[\mathsf{fun}\{f, x.\, e\}, v] \Downarrow e'}\ \text{EC-App}$$

Fig. 15. Big-stop semantics for evaluation context PCF

The workhorse of the evaluation context formulation is the rule EC-Seq. This rule handles all congruences by finding the next subexpression ready to take one step, which is picked out with a hole. However, big-step semantics needs to consider the result of many steps of evaluation, not just one, and subsequent steps of evaluation may not be local to the first. The rule EC-Seq handles this by chaining together sequences of evaluation steps, much like the rule M-Step does for multi-step semantics and let-bindings do for monadic normal form. First the hole contents e_1 is evaluated, resulting in e_1'. Then the resulting expression $C\langle e_1' \rangle$ is reassessed for evaluation, allowing the meta-syntax to readjust the hole to the next part of the expression to evaluate.

This approach does have at least one theoretical drawback resulting from how the system contorts around the rule EC-Seq. This rule does not guarantee evaluation progress, as EC-Stop or EC-Val could satisfy its left-hand premiss, leaving the righthand premiss equal to the conclusion. As a result, arbitrarily deep non-progressing derivations exist for all well-typed expressions, which is not a desirable property. Nonetheless, if one is willing to introduce some additional complexity, this issue could be alleviated by mutually inductively defining the big-stop judgment alongside the progressing judgment.

7.3 Annihilator Effect

It is often sufficient to only consider the effects emitted by computation rather than any value resulting from computation. (Even if one wishes to consider resulting values, information about such values can be embedded into the emitted effects.) In such an effectful setting, one can leverage the algebraic structure of effect sequences to simplify away most stopping rules.

To make sense of the algebraic approach provided in this section, it is necessary to consider the structure of the effects represented by the term a. Such a term represents a chronological sequence of effects. Such a sequence can be treated as a *monoid*, which is an algebraic structure with an associative binary operation and an identity element. In this case, the binary operation is sequencing—ab means the effects of b follow those of a—and 1 represents the identity effect.

To support big-stop semantics, the monoid of effects needs to be extended with one new element not used in the original small-step system: 0. This new element does not need to have any actual incarnation as an effect and can instead be thought of as an algebraic bookkeeping device. The new 0 element acts as a left annihilator for the monoid, so that $0a = 0$ for any a. The fact that it is a *left* annihilator rather than an annihilator simpliciter means that $a0$ does not necessarily equal 0. As a result, a string of effects including 0s appears to cut the string after the first 0: $abc0def = abc0$.

To then extend effectful big-step rules into big-stop rules, only one new rule needs to be added, and no other rules need to be touched. This new rule should say that an expression can always halt evaluation at an arbitrary value by emitting the 0 effect. For example, extending the big-step rules of Figure 7 to big-stop yields Figure 16, where the single new stopping rule is StA-Stop. Every other rule is identical to its big-step counterpart.

$$\frac{e \Downarrow e' \rightsquigarrow a}{\mathsf{S}[e] \Downarrow \mathsf{S}[e'] \rightsquigarrow a}\textsc{StA-Succ} \qquad \frac{e \Downarrow \mathsf{Z} \rightsquigarrow a \qquad e_1 \Downarrow v_1 \rightsquigarrow b}{\mathsf{case}\{e_1; x.e_2\}[e] \Downarrow v_1 \rightsquigarrow ab}\textsc{StA-CaseZ} \qquad \frac{e \Downarrow \mathsf{S}[v] \rightsquigarrow a \qquad [v/x]e_2 \Downarrow v_2 \rightsquigarrow b}{\mathsf{case}\{e_1; x.e_2\}[e] \Downarrow v_2 \rightsquigarrow ab}\textsc{StA-CaseS}$$

$$\frac{e_1 \Downarrow \mathsf{fun}\{f, x.\,e\} \rightsquigarrow a \qquad e_2 \Downarrow v_2 \rightsquigarrow b \qquad [\mathsf{fun}\{f, x.\,e\}/f, v_2/x]e \Downarrow v \rightsquigarrow c}{\mathsf{app}[e_1, e_2] \Downarrow v \rightsquigarrow abc}\textsc{StA-App} \qquad \frac{e \Downarrow v \rightsquigarrow b}{\mathsf{eff}\{a; e\} \Downarrow v \rightsquigarrow ab}\textsc{StA-Eff} \qquad \frac{v\ val}{v \Downarrow v \rightsquigarrow 1}\textsc{StA-Val} \qquad \frac{v\ val}{e \Downarrow v \rightsquigarrow 0}\textsc{StA-Stop}$$

Fig. 16. Big-stop semantics with the annihilator

This treatment allows the existing big-step rules to naturally propagate stoppage since every halted expression is just some value. This compatibility is the reason that this big-stop system can get by with only one stopping rule—the resulting arbitrary value is treated just as any other value and the existing rules can compose the effects of its (partial) evaluation with no added difficulty.

One might also be concerned that this approach could be unsafe because any value at all can be chosen for the expression e in StA-Stop, so computation might not continue as the typical semantics would dictate. However, because stopping emits the 0 effect, any computation that follows stopping has its effects annihilated. Moreover, one can use the presence of the 0 effect to determine whether the value resulting from big-stopping is spurious or not. The effects include 0 iff the stopping rule is used in the derivation, since 0 is a new effect not present in the small- or big-step semantics. Thus if there is a 0, the resulting value is meaningless, and if there is no 0, the resulting value matches that found by the big-step semantics. One might achieve the same result by replacing the arbitrary value with a dummy token, but then the dynamics would need to account for premisses that expect certain forms of values, such as the premiss $e \Downarrow \mathsf{Z} \rightsquigarrow a$ of StA-CaseZ.

With the new stopping rule in place, the following correspondence can be drawn between effectful big-stop semantics and small-step semantics:

Theorem 23 (Annihilating Stop/Multi Equivalence). *For all expressions e and effects a,*

$$(\exists e_1.\, e \Downarrow e_1 \rightsquigarrow a0) \iff (\exists e_2.\, e \overset{*}{\mapsto} e_2 \rightsquigarrow a)$$

Proof. This equivalence follows by induction on the derivation of each judgment. □

Unlike the previous effectful equivalence provided, Theorem 16, Theorem 23 does not express an exact equivalence between big-stopping and multi-stepping. In particular, the expressions e_1 and e_2 bear no clear relation, and the effect trace induced in the big-stop judgment ends in 0. (The multi-step judgment's effect cannot include any 0s because 0 is a new effect introduced only via StA-Stop.) Nonetheless, Theorem 23 shows that the key focus of this setting, the effects of a, are clearly maintained through each semantic system. As a result, Theorem 23 can be used similarly to previous equivalences to recover most of the value of small-step semantics in this setting.

8 Imperative Variant

To allay any concerns that the big-stop approach only applies in nicely-behaved functional settings with impoverished effects, this section shows how to apply big-stop techniques to an imperative while-loop language with mutation.

An Imperative Language. The language we consider has the following statements, where x ranges over variable names and a stands for integer-valued arithmetic expressions that contain variables

$$\textsc{SI-Bind}\ \frac{\sigma(a) = z}{\langle x := a \mid \sigma\rangle \mapsto \langle \texttt{skip} \mid [z/x]\sigma\rangle} \qquad \textsc{SI-Seq1}\ \frac{\langle s_1 \mid \sigma\rangle \mapsto \langle s_1' \mid \sigma'\rangle}{\langle s_1; s_2 \mid \sigma\rangle \mapsto \langle s_1'; s_2 \mid \sigma'\rangle} \qquad \textsc{SI-Seq2}\ \frac{}{\langle \texttt{skip}; s \mid \sigma\rangle \mapsto \langle s \mid \sigma\rangle}$$

$$\textsc{SI-Then}\ \frac{\sigma(a) \neq 0}{\langle \texttt{if}\{a\}\texttt{then}\{s\} \mid \sigma\rangle \mapsto \langle s \mid \sigma\rangle} \qquad \textsc{SI-Else}\ \frac{\sigma(a) = 0}{\langle \texttt{if}\{a\}\texttt{then}\{s\} \mid \sigma\rangle \mapsto \langle \texttt{skip} \mid \sigma\rangle}$$

$$\textsc{SI-Do}\ \frac{\sigma(a) \neq 0}{\langle \texttt{while}\{a\}\texttt{do}\{s\} \mid \sigma\rangle \mapsto \langle s; \texttt{while}\{a\}\texttt{do}\{s\} \mid \sigma\rangle} \qquad \textsc{SI-Done}\ \frac{\sigma(a) = 0}{\langle \texttt{while}\{a\}\texttt{do}\{s\} \mid \sigma\rangle \mapsto \langle \texttt{skip} \mid \sigma\rangle}$$

Fig. 17. Small-step semantics for the imperative language

$$\textsc{BI-Skip}\ \frac{}{\langle \texttt{skip} \mid \sigma\rangle \Downarrow \langle \texttt{skip} \mid \sigma\rangle} \qquad \textsc{BI-Then}\ \frac{\sigma(a) \neq 0 \qquad \langle s \mid \sigma\rangle \Downarrow \langle s' \mid \sigma'\rangle}{\langle \texttt{if}\{a\}\texttt{then}\{s\} \mid \sigma\rangle \Downarrow \langle s' \mid \sigma'\rangle} \qquad \textsc{BI-Else}\ \frac{\sigma_1(a) = 0}{\langle \texttt{if}\{a\}\texttt{then}\{s\} \mid \sigma\rangle \Downarrow \langle \texttt{skip} \mid \sigma\rangle}$$

$$\textsc{BI-Seq}\ \frac{\langle s_1 \mid \sigma_1\rangle \Downarrow \langle \texttt{skip} \mid \sigma_2\rangle \qquad \langle s_2 \mid \sigma_2\rangle \Downarrow \langle s' \mid \sigma_3\rangle}{\langle s_1; s_2 \mid \sigma_1\rangle \Downarrow \langle s' \mid \sigma_3\rangle} \qquad \textsc{BI-Bind}\ \frac{\sigma(a) = z}{\langle x := a \mid \sigma\rangle \Downarrow \langle \texttt{skip} \mid [z/x]\sigma\rangle}$$

$$\textsc{BI-Do}\ \frac{\sigma_1(a) \neq 0 \qquad \langle s \mid \sigma_1\rangle \Downarrow \langle \texttt{skip} \mid \sigma_2\rangle \qquad \langle \texttt{while}\{a\}\texttt{do}\{s\} \mid \sigma_2\rangle \Downarrow \langle s' \mid \sigma_3\rangle}{\langle \texttt{while}\{a\}\texttt{do}\{s\} \mid \sigma_1\rangle \Downarrow \langle s' \mid \sigma_3\rangle} \qquad \textsc{BI-Done}\ \frac{\sigma_1(a) = 0}{\langle \texttt{while}\{a\}\texttt{do}\{s\} \mid \sigma\rangle \Downarrow \langle \texttt{skip} \mid \sigma\rangle}$$

Fig. 18. Big-stop versions of the big-step rules of the imperative language

(e.g., $x + 5$). For simplicity, all variables stand for integers.

$$s ::= \texttt{if}\{a\}\texttt{then}\{s\} \mid x := a \mid \texttt{while}\{a\}\texttt{do}\{s\} \mid s; s \mid \texttt{skip}$$

The guards of the control-flow statements, namely if-then conditionals and while-do loops, test for the inequality of a given arithmetic expression with zero. Compound statements are sequenced using $s_1; s_2$. The language is also equipped with the identity statement `skip`, which does nothing.

The small-step semantics of this imperative language is given by Figure 17. In this setting, a step relates pairs of statements s and states σ. A state is a mapping of variables to integers. We use the notation $\sigma(a)$ to evaluate the arithmetic expression a under those mappings, and we use the notation $[z/x]\sigma$ to update σ to include a binding of z to x. Note that, upon termination, one is always left with the statement `skip`; the computational result is recorded via the paired state.

The big-step semantics defines the judgment $\langle s \mid \sigma\rangle \Downarrow \sigma'$, which states that statement s terminates in state σ' when executed with starting state σ. The syntax directed rules for the judgment are identical to the rules in Figure 18 when we omit the first component of the evaluation result.

Big-Stop Semantics. Following the same principle as for PCF, we extend the big-step rules for the imperative language in two steps. To match the format of the small-step semantics, we change the results of computations to be pairs $\langle s \mid \sigma\rangle$ of statements s and states σ instead of simply states like in standard big-step semantics. The big-stop versions of the big-step rules are given in Figure 18.

$$\textsc{StI-Seq}\ \frac{\langle s_1 \mid \sigma\rangle \Downarrow \langle s_1' \mid \sigma'\rangle}{\langle s_1; s_2 \mid \sigma\rangle \Downarrow \langle s_1'; s_2 \mid \sigma'\rangle} \qquad \textsc{StI-Do}\ \frac{\sigma(a) \neq 0 \qquad \langle s \mid \sigma\rangle \Downarrow \langle s' \mid \sigma'\rangle}{\langle \texttt{while}\{a\}\texttt{do}\{s\} \mid \sigma\rangle \Downarrow \langle s'; \texttt{while}\{a\}\texttt{do}\{s\} \mid \sigma'\rangle} \qquad \textsc{StI-Stop}\ \frac{}{\langle s \mid \sigma\rangle \Downarrow \langle s \mid \sigma\rangle}$$

Fig. 19. New rules for big-stop semantics for the imperative language

Because big-step rules only cover terminating computations, these rules derive judgments of the form $\langle s \mid \sigma\rangle \Downarrow \langle \texttt{skip} \mid \sigma'\rangle$ where the resulting statement is always `skip`. To extend the judgment to partial evaluations, one then needs to add the stopping rules of Figure 19. As usual, there is one rule for stopping any program where it stands (StI-Stop) and a few additional rules for propagating that stoppage into the middle of the program. Note that StI-Stop can also just replace BI-Skip.

As expected, these big-stop semantics agree with the small-step semantics (Theorem 24).

Theorem 24 (Imperative Stop/Multi Equivalence). *For all statements s, s' and states σ, σ',*

$$\langle s \mid \sigma\rangle \Downarrow \langle s' \mid \sigma'\rangle \iff \langle s \mid \sigma\rangle \overset{*}{\mapsto} \langle s' \mid \sigma'\rangle$$

Proof. This property follows from induction over the derivations. □

In many formalisms, imperative big-step judgments are of the form $\langle s \mid \sigma\rangle \Downarrow \sigma'$ where s is a statement and σ, σ' are states. This form of judgment can also be extended to big-stop, in particular by using the annihilator optimization from Section 7. We exemplify this extension in Section C.

9 Related Work

Big-stop semantics is not the first system designed to ameliorate the shortcomings of big-step semantics. Various extensions to big-step semantics already exist in the literature. Broadly speaking, these extensions operate along two axes: interpreting big-step rules coinductively, and adding additional rules for handling the problematic cases. We describe a variety of these systems in this section and display what their rules look like using variants of the "big-step" judgment $e \Downarrow \infty$, meaning that the expression e does not terminate.

Coinductive Techniques. Cousot and Cousot were the first to propose using coinductive techniques to handle infinitary computation in the context of big-step semantics [7]. They did so by introducing additional, coinductively-interpreted big-step rules to capture nontermination. Put simply, their rules define nontermination of some expression e via the nontermination of some subevaluation of e. In principle, this approach appears to require a new set of such rules for each possibly-nonterminating subexpression of a given syntactic form, which often more than doubles the number of inference rules. An example of their style of nontermination rules is given in Figure 20 for application.[3] There are three such rules that are added alongside the single terminating big-step application rule. These coinductive rules form the basis of several proposals to characterize nonterminating computation, including Crole [9], Hughes and Moran [24], and others.

Leroy and Grall [35] formalize and compare both Cousot and Cousot's [7] coinductive treatment of nontermination and a notion of "coevaluation" arising from coinductive interpretation of the standard eager big-step rules. This coevaluation is able to handle divergence coinductively in a similar manner to the rules of Figure 20, but simultaneously it can reason about the values that result from terminating evaluation. Some key relations found between evaluation, coevaluation, and divergence include that if e evaluates to v then e also coevaluates to v, and that if e coevaluates

[3] Recall that $e \Downarrow \infty$ means that the reduction of e does not terminate.

$$\frac{e_1 \Downarrow \infty}{\mathsf{app}[e_1, e_2] \Downarrow \infty}\ \textsc{CC-App1} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow \infty}{\mathsf{app}[e_1, e_2] \Downarrow \infty}\ \textsc{CC-App2} \qquad \frac{e_1 \Downarrow \mathsf{fun}\{f, x.\, e\} \qquad e_2 \Downarrow v \qquad [(\mathsf{fun}\{f, x.\, e\})/f, v/x]e \Downarrow \infty}{\mathsf{app}[e_1, e_2] \Downarrow \infty}\ \textsc{CC-App3}$$

Fig. 20. Cousot-Cousot-style [7] coinductive divergence rules for application

$$\frac{e_1 \Downarrow o_1 \qquad \mathsf{app2}\ o_1\ e_2 \Downarrow v}{\mathsf{app}\ e_1\ e_2 \Downarrow v}\ \textsc{Ch-App1} \qquad \frac{v_1\ val \qquad e_2 \Downarrow o_2 \qquad \mathsf{app3}\ v_1\ o_2 \Downarrow v}{\mathsf{app2}\ v_1\ e_2 \Downarrow v}\ \textsc{Ch-App2} \qquad \frac{}{\mathsf{app2}\ \infty\ e_2 \Downarrow \infty}\ \textsc{Ch-App2Div}$$

$$\frac{[\mathsf{fun}\{f, x.\, e\}/f, v/x]e \Downarrow o}{\mathsf{app3}\ \mathsf{fun}\{f, x.\, e\}\ v \Downarrow o}\ \textsc{Ch-App3} \qquad \frac{}{\mathsf{app3}\ \mathsf{fun}\{f, x.\, e\}\ \infty \Downarrow \infty}\ \textsc{Ch-App3Div}$$

Fig. 21. Pretty-big-step-style [6] inductive/coinductive rules for application with divergence

to v then e either evaluates to v or diverges. However, there exist expressions that diverge but do not coevaluate, and big-step coevaluation does not perfectly coincide with small-step coevaluation.

Later work by Cousot and Cousot approaches the problem using fixedpoints in a different way that is neither inductive nor coinductive [8]. The resulting "bifinitary" system is able to handle both terminating and nonterminating big-step-style inference rules using just one kind of judgment, rather than separate terminating and nonterminating ones. As a result, this system requires less rule duplication than their previous system, but still requires multiple rules per syntactic form. Cousot and Cousot provide a three-rule setup for reasoning about function application in section 6.4.4 of their work [8]. The three rules are largely the same as those of Figure 20 except that their version of CC-App3 is generalized to allow for both terminating and nonterminating executions in its third premiss and conclusion, so long as the result of each reduction is the same.

Charguéraud's pretty-big-step semantics [6] takes a similar approach to Cousot and Cousot's bifinitary system [8] but ensures that the interpretation of its rules correspond to standard induction and coinduction. In particular, interpreting pretty-big-step rules inductively corresponds to evaluation (which handles termination), and interpreting the same rules coinductively corresponds to coevaluation (which handles nontermination). The pretty-big-step system also builds upon another technique in Cousot and Cousot [8]: breaking down expression reduction into smaller chunks. In Charguéraud's work, this breakdown minimizes premiss duplication across rules. Charguéraud then goes further to capture features like divergence and exceptions alongside values via "outcomes." The new outcome for, e.g., divergence is an additional pseudo-value ∞ which requires its own propagation rules. An example of a resulting rule set can be found in Figure 21, where application is broken down into app, app2, and app3, and where o ranges over outcomes. While this approach results in more inference rules, Charguéraud reports that having fewer premisses leads to smaller formal definitions and proofs. Charguéraud also proves similar theorems about pretty-big-step coevaluation as Leroy and Grall prove about their coevaluation [35]. However, because pretty-big-step semantics can coevaluate to the nonterminating outcome ∞, all diverging expressions can be shown to coevaluate.

Similar to pretty-big-step semantics is the flag-based semantics of Poulsen and Mosses [41]. They modify pretty-big-step semantics by pairing expressions with status flags that propagate information about termination: $\downarrow$ for successful termination and $\uparrow$ for divergence. This approach

$$\frac{v\ val}{(e,\uparrow) \Downarrow (v,\uparrow)}\ \textsc{PM-Div} \qquad \frac{v\ val}{(v,\downarrow) \Downarrow (v,\downarrow)}\ \textsc{PM-Val} \qquad \frac{(e_1,\downarrow) \Downarrow (v_1,\delta) \quad (\mathsf{app2}\ v_1\ e_2,\delta) \Downarrow (v,\delta')}{(\mathsf{app}\ e_1\ e_2,\downarrow) \Downarrow (v,\delta')}\ \textsc{PM-App1}$$

$$\frac{v_1\ val \quad (e_2,\downarrow) \Downarrow (v_2,\delta) \quad (\mathsf{app3}\ v_1\ o_2,\delta) \Downarrow (v,\delta')}{(\mathsf{app2}\ v_1\ e_2,\downarrow) \Downarrow (v,\delta')}\ \textsc{PM-App2} \qquad \frac{[\mathsf{fun}\{f,x.\,e\}/f, v/x]e, \downarrow \Downarrow (v,\delta)}{(\mathsf{app3}\ \mathsf{fun}\{f,x.\,e\}\ v,\downarrow) \Downarrow (v,\delta)}\ \textsc{PM-App3}$$

Fig. 22. Flag-based [41] inductive/coinductive rules for application with divergence

$$\frac{}{e \downarrow z}\ \textsc{GR-StopS} \qquad \frac{}{e \Downarrow z}\ \textsc{GR-StopR} \qquad \frac{e_1 \downarrow u_1 \quad e_2 \downarrow u_2 \quad \mathsf{app}[u_1,u_2] \Downarrow u}{\mathsf{app}[e_1,e_2] \downarrow u}\ \textsc{GR-AppS} \qquad \frac{[\mathsf{fun}\{f,x.\,e\}/f, v/x]e \downarrow u}{\mathsf{app}[\mathsf{fun}\{f,x.\,e\}, v] \Downarrow u}\ \textsc{GR-AppR}$$

Fig. 23. Partial-proof [18] inductive rules for application

reduces the number of propagation rules needed because, rather than allow a range of outcomes, the result of evaluation is always some value that can be propagated naturally by the existing rules. It might just be the case that the resulting value is ignored because evaluation is flagged as diverging. An example of flag-based rules applied to application can be found in Figure 22, where δ ranges over flags. There are three rules for application, and the rules PM-Div and PM-Val are also included to show how flags are handled. Note that a coinductive interpretation of these rules is necessary to derive nontermination judgments. Such nontermination judgments take the form $(e,\downarrow) \Downarrow (v,\uparrow)$ and ignore the value v so that they have the same meaning as $e \Downarrow \infty$.

Inductive Techniques. Unlike all of other work discussed so far, Gunter and Rémy's partial proof semantics [18] does not use a coinductive relation to handle divergence. Instead, partial proofs augment big-step semantics with the ability to abstractly denote values with "logic variables" and leave the derivation of those variables uncalculated. This approach has the effect of allowing big-step derivations to exist in a partially-completed form, which means nonterminating computation can be represented with a sequence of arbitrary-depth derivations. The rules of this system use two forms of big-step evaluation judgments, the "search" judgment $e \downarrow u$ and the "redex" judgment $e \Downarrow u$, to help ensure that one cannot make arbitrarily-deep derivations of terminating computations. Each syntactic form in this system requires its own search and redex rules. The resulting rules look like those of Figure 23, where v ranges over only values, z ranges over only logic variables, and u ranges over both. This rule set shows the two rules for application as well as the two rules for stopping the growth of the proof tree, GR-StopS and GR-StopR.

Another inductive approach comes from a line of work based on step-indexing or "fuel" [2, 14, 43]. This approach augments the operational semantics with a counter and augments the language with time-out error states. The counter counts down in a manner corresponding to steps or derivation depth. When the counter is above 0, the semantics behave as normal. However, when the counter hits 0, the evaluation is forced into a time-out error state represented here by $\square$. By quantifying over counters, this approach allows for every finite prefix of computation to be captured in a big-step manner (although each strict prefix concludes in an error state). The rules of such a fuel-based system are exemplified in Figure 24, where $e \Downarrow_n v$ means that e evaluates to v with a big-step derivation no deeper than n, and where $\square$ represents the error state. These fuel-based rules structurally match those for the coinductive interpretation of divergence (Figure 20) except

$$\frac{}{e \Downarrow_0 \square}\ \textsc{SC-Stop} \qquad \frac{e_1 \Downarrow_{n-1} \square}{\mathsf{app}[e_1, e_2] \Downarrow_n \square}\ \textsc{SC-App1} \qquad \frac{e_1 \Downarrow_{n-1} v_1 \qquad e_2 \Downarrow_{n-1} \square}{\mathsf{app}[e_1, e_2] \Downarrow_n \square}\ \textsc{SC-App2}$$

$$\frac{e_1 \Downarrow_{n-1} \mathsf{fun}\{f, x.\, e\} \qquad e_2 \Downarrow_{n-1} v \qquad [\mathsf{fun}\{f, x.\, e\}/f, v/x]e \Downarrow_{n-1} \square}{\mathsf{app}[e_1, e_2] \Downarrow_n \square}\ \textsc{SC-App3}$$

Fig. 24. Counter-based [2, 14, 43] inductive rules for application

$$\frac{e_1 \Downarrow_m \infty}{\mathsf{app}[e_1, e_2] \Downarrow_n \infty}\ \textsc{ZB-App1} \qquad \frac{e_1 \Downarrow_\ell v_1 \qquad e_2 \Downarrow_m \infty}{\mathsf{app}[e_1, e_2] \Downarrow_n \infty}\ \textsc{ZB-App2}$$

$$\frac{e_1 \Downarrow_k \mathsf{fun}\{f, x.\, e\} \qquad e_2 \Downarrow_\ell v \qquad [\mathsf{fun}\{f, x.\, e\}/f, v/x]e \Downarrow_m \infty}{\mathsf{app}[e_1, e_2] \Downarrow_n \infty}\ \textsc{ZB-App3}$$

Fig. 25. Counter-based [47] coinductive rules for application

that here they are inductive and they include the rule SC-Stop. A typical type safety theorem using fuel would state that, given any amount of fuel, a well-typed expression evaluates to either a value of the same type or an error state, $\forall n.\, \cdot \vdash e : \tau \wedge e \Downarrow_n v \implies \cdot \vdash v : \tau \vee v = \square$. The idea behind such a theorem is that, when enough fuel is given, the values of terminating computations are captured, and when too little fuel is given (including for infinite computations), it is still verified that evaluation could safely occur up until the fuel is used up and the time-out error is reached.

While fuel counters are inductive, it is also possible to adapt them to work alongside coinductive rules like in the coevaluative work of Zúñiga and Bel-Enguix [47]. This work has two sets of rules: one set for concluding with values and one for concluding with nontermination. The counters count down in the value rules to ensure that, even though they are interpreted coinductively, these rules can only be used for terminating computations—an infinite computation would run down any counter, preventing these rules' use. Thus, nonterminating computations cannot coevaluate to values here, unlike in other coevaluative systems. Simultaneously, the nontermination rules (Figure 25) ignore the counters and thus act like the standard rules of Cousot and Cousot (Figure 20).

Hoffmann and Hofmann's partial big-step semantics tackle nontermination inductively, without coinduction [21]. While their work is specialized to reasoning about program cost, we abstract to its key takeaways here. Partial big-step semantics handles nontermination by introducing a new judgment, $e \Downarrow \square \rightsquigarrow a$, to represent the nondeterministic halting of computation during e's evaluation after accumulating some trace a. (Note that here the symbol $\square$ is *not* an expression and is considered part of the judgment notation.) This approach behaves similarly to the step-counting approach without needing any counter. An example of partial big-step rules for function application can be found in Figure 26.[4] Big-stop semantics is based on this work.

Later work by D. M. Kahn improved upon the partial big-step approach to cost analysis by introducing an algebraic annihilator for cost traces and treating $\square$ as a value to reduce propagation rules [26]. The resulting system only adds one new rule to the standard big-step rules, rather than a few rules for each syntactic form, and it makes use of a variant of monadic normal form to help with this rule minimization. An example of Kahn's rules for application can be found in Figure 27. This work formed the basis of some of the ergonomic optimizations of Section 7.

[4] Technically the work on partial big-step semantics also uses a variant of monadic normal form. However, this form plays no role in their partial big-step semantics, and is instead used for other conveniences related to rule presentation. As monadic normal form would obscure some of the comparisons made in this section, we do not use it for this example.

$$\frac{}{e \Downarrow \square \rightsquigarrow 1}\ \textsc{HH-Stop} \qquad \frac{e_1 \Downarrow \square \rightsquigarrow a}{\mathsf{app}[e_1, e_2] \Downarrow \square \rightsquigarrow a}\ \textsc{HH-App1} \qquad \frac{e_1 \Downarrow v_1 \rightsquigarrow a \qquad e_2 \Downarrow \square \rightsquigarrow b}{\mathsf{app}[e_1, e_2] \Downarrow \square \rightsquigarrow ab}\ \textsc{HH-App2}$$

$$\frac{e_1 \Downarrow \mathsf{fun}\{f, x.\, e\} \rightsquigarrow a \qquad e_2 \Downarrow v \rightsquigarrow b \qquad [\mathsf{fun}\{f, x.\, e\}/f, v/x]e \Downarrow \square \rightsquigarrow c}{\mathsf{app}[e_1, e_2] \Downarrow \square \rightsquigarrow abc}\ \textsc{HH-App3}$$

Fig. 26. Hoffmann-Hofmann-style partial big-step [21] inductive rules for application

$$\frac{}{e \Downarrow \square \rightsquigarrow 0}\ \textsc{K-Stop} \qquad \frac{[\mathsf{fun}\{f, x.\, e\}/f, v/x]e \Downarrow v' \rightsquigarrow a}{\mathsf{app}[\mathsf{fun}\{f, x.\, e\}, v] \Downarrow v' \rightsquigarrow a}\ \textsc{K-App}$$

Fig. 27. Kahn-style partial big-step [26] inductive rules for application with divergence

Comparison with Big-Stop. When comparing big-stop semantics and the other systems discussed here, there are clear similarities. Aside from the work that big-stop is based on, one of the most similar lines of work is that using fuel [2, 14, 43]. By consuming fuel in proportion to evaluation step count, one should be able to induce an error state at exactly the same expressions that big-stop can stop. If the error states are then altered to record the last pre-error expression, then such an approach would successfully match small-step while using inductive, big-step-style rules, just as big-stop does. However, big-stop can get these same results with less work. Big-stop rules are essentially a superset of big-step rules, whereas fuel-based rules are more intrusive to implement because every expression must track additional state: the fuel counter. This state must be quantified over in fuel-based formalizations, and such quantification must be discharged by, e.g., proving that program evaluation is sufficiently independent of fuel count. Because big-stop needs none of this, it shows that the fuel of fuel-based approaches is actually unnecessary. Compare the typical fuel-based type safety statement given earlier in this section to big-stop's Theorem 11.

Otherwise, the effectful version of big-stop semantics has the most obvious similarities to discuss. Structurally, the inductive judgment for effectful stopping is very similar to the coinductive judgment for divergence [7]. Like the use of the flag $\downarrow$ from flag-based semantics [41], annihilator-based big-stop semantics uses the special effect 0 to prevent spurious effects from spurious evaluations that are introduced in the effort to reduce rule count. Like partial proof semantics [18], big-stop derivations can be cut off at any point. And of course, the effectful big-stop semantics is based on partial big-step semantics [21, 26].

Nonetheless, big-stop semantics introduces some clear benefits over most other systems discussed here. One benefit is that it is inductive, which coheres better with other inductively defined judgments like typing, and which enables induction-based proofs rather than less-commonly-understood coinduction-based proofs. Another benefit is that it can be optimized to require extremely few rules—only one or two rules are needed for entire languages, whereas other approaches require multiple rules for each syntactic form. It also does not introduce additional complexity like counters to the judgments. And finally, big-stop semantics has direct correspondence with the behaviour of small-step semantics, whereas coevaluation has a less-clear relationship.

Other Related Work. Much other work exists which is less comparable to big-stop semantics, and almost all of it is coinductive. A few such examples are listed as follows: Nakata and Uustalu introduce a mutually-coinductive trace-based semantics for an imperative language that includes infinite loops [37]. Capretta's delay monad uses coinductive types to capture partial computation in a type-theoretic way [5], and Danielsson has applied this monad to operational semantics, resulting in something similar to a hybrid big-step operational/denotational semantics that can handle nontermination via giving a big-step system a mixed recursive/corecursive definitional interpreter equipped with a partiality monad [11]. Dagnino uses coaxioms to develop a metatheory for divergent reasoning concerning big-step systems. [10]. A similar approach has also been applied to the effectful setting, where infinite sequences of effects are formalized as an ω-monoid [3], rather than just a finite monoid as in this work.

10 Conclusion

Big- and small-step operational semantics are popular forms of semantics that traditionally have occupied subtly different niches. Small-step semantics describe each individual step of computation, and can capture diverging computation just as well as converging. Big-step semantics sacrifice the ability to reason about diverging computation in exchange for easy access to the values resulting from computation, which is more convenient for purposes such as proving semantic preservation.

Big-stop semantics recaptures the expressive power of small-step semantics with all the ergonomic benefits of big-step semantics. Big-stop semantics works by introducing rules for non-deterministically stopping computation. In this article, we have presented big-stop semantics for a call-by-value variant of PCF and a simple imperative language. However, we are not aware of obstacles that prevent or complicate the adaptation of big-stop semantics to more complex languages that enjoy big-step semantics. Nevertheless, it remains an open research question if big-stop semantics can be extended to language features, such as concurrency and continuations, that are challenging to express in a big-step semantics.

Big-stop semantics is the latest in a long line of work that aims to regain the power of small-step semantics in a big-step-style system. However, other work in this vein has focused on coinductive techniques and/or required comparatively large changes to the standard big-step system. For this reason, the simplicity of big-stop semantics is notable. Big-stop semantics is both inductive and can be formulated to require very few changes to the original big-step system.

Acknowledgments

We would like to first acknowledge Martin Hofmann (1965-2018), whose initial idea of replacing coinductive techniques for handling nontermination in a cost analysis setting resulted in the partial big-step semantics upon which big-stop semantics was built. We would also like to acknowledge Bob Harper and the anonymous reviewers of this article for their helpful feedback on this work.

This material is based upon work supported by the National Science Foundation under Grants Nos. 2311983 and 2525102, and the United States Air Force Office of Scientific Research under grant number FA9550-21-0009 and FA9550-23-1-0434 (Tristan Nguyen, program manager). Any opinions, findings, and conclusions or recommendations in this material are those of the authors and do not necessarily reflect the views of the NSF or the AFOSR.

References

[1] 2022. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-2/ https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.

[2] Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 666–679.

[3] Davide Ancona, Francesco Dagnino, Jurriaan Rot, and Elena Zucca. 2020. A big step from finite to infinite computations. *Science of Computer Programming* 197 (2020), 102492.

[4] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *J. Autom. Reason.* 43, 3 (2009), 263–288. doi:10.1007/S10817-009-9148-3

[5] Venanzio Capretta. 2005. General recursion via coinductive types. *Logical Methods in Computer Science* 1 (2005).

[6] Arthur Charguéraud. 2013. Pretty-big-step semantics. In *European Symposium on Programming*. Springer, 41–60.

[7] Patrick Cousot and Radhia Cousot. 1992. Inductive definitions, semantics and abstract interpretations. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 83–94.

[8] Patrick Cousot and Radhia Cousot. 2009. Bi-inductive structural semantics. *Information and Computation* 207, 2 (2009), 258–283.

[9] Roy L Crole. 1998. *Lectures on [Co] induction and [Co] algebras.* Technical Report. Technical Report 1998/12, Department of Mathematics and Computer Science ....

[10] Francesco Dagnino. 2022. A meta-theory for big-step semantics. *ACM Transactions on Computational Logic (TOCL)* 23, 3 (2022), 1–50.

[11] Nils Anders Danielsson. 2012. Operational semantics using the partiality monad. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming.* 127–138.

[12] Rémi Douence and Pascal Fradet. 2007. The next 700 Krivine machines. *Higher-order and symbolic computation* 20 (2007), 237–255.

[13] Chucky Ellison and Grigore Rosu. 2012. An executable formal semantics of C with applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 533–544. doi:10.1145/2103656.2103719

[14] Erik Ernst, Klaus Ostermann, and William R Cook. 2006. A virtual class calculus. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 270–282.

[15] Matthias Felleisen. 1987. *The calculi of lambda-nu-cs conversion: a syntactic theory of control and state in imperative higher-order programming languages.* Ph. D. Dissertation. Indiana University.

[16] Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. *ACM Sigplan Notices* 28, 6 (1993), 237–247.

[17] Maurizio Gabbrielli and Simone Martini. 2010. *Abstract Machines.* Springer London, London, 1–25. doi:10.1007/978-1-84882-914-5_1

[18] Carl A Gunter and Didier Rémy. 1993. A proof-theoretic assessment of runtime type errors. In *Technical Report 11261921230-43TM.* Citeseer.

[19] Robert Harper. 2016. *Practical foundations for programming languages.* Cambridge University Press.

[20] John Hatcliff and Olivier Danvy. 1994. A generic account of continuation-passing styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 458–471.

[21] Jan Hoffmann and Martin Hofmann. 2010. Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In *Asian Symposium on Programming Languages and Systems.* Springer, 172–187.

[22] Jan Hoffmann and Steffen Jost. 2022. Two decades of automatic amortized resource analysis. *Mathematical Structures in Computer Science* 32, 6 (2022), 729–759.

[23] Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. *ACM SIGPLAN Notices* 38, 1 (2003), 185–197.

[24] John Hughes and Andrew Moran. 1995. Making choices lazily. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture.* 108–119.

[25] Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Peter Sestoft.

[26] David Kahn. 2024. *Leveraging Linearity to Improve Automatic Amortized Resource Analysis.* Ph. D. Dissertation. Carnegie Mellon University.

[27] David Kahn, Jan Hoffmann, and Runming Li. 2025. *Big-Stop Semantics.* doi:10.5281/zenodo.17316781

[28] Gilles Kahn. 1987. Natural semantics. In *Annual symposium on theoretical aspects of computer science.* Springer, 22–39.

[29] Gerwin Klein and Tobias Nipkow. 2006. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 4 (2006), 619–695.

[30] Jean-Louis Krivine. 2007. A call-by-name lambda-calculus machine. *Higher-order and symbolic computation* 20 (2007), 199–207.

[31] Peter J Landin. 1964. The mechanical evaluation of expressions. *The computer journal* 6, 4 (1964), 308–320.

[32] John Launchbury. 1993. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 144–154.

[33] Xavier Leroy. 1990. *The ZINC experiment: an economical implementation of the ML language.* Ph. D. Dissertation. INRIA.

[34] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 42–54.

[35] Xavier Leroy and Hervé Grall. 2009. Coinductive big-step operational semantics. *Information and Computation* 207, 2 (2009), 284–304.

[36] Robin Milner, Mads Tofte, and Robert Harper. 1990. *Definition of standard ML.* MIT Press.

[37] Keiko Nakata and Tarmo Uustalu. 2009. Trace-based coinductive operational semantics for While: big-step and small-step, relational and functional styles. In *International Conference on Theorem Proving in Higher Order Logics.* Springer, 375–390.

[38] Michael Norrish. 1999. *C formalised in HOL.* Ph. D. Dissertation. University of Cambridge, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.624131

[39] Gordon D. Plotkin. 1977. LCF considered as a programming language. *Theoretical computer science* 5, 3 (1977), 223–255.

[40] Gordon D Plotkin. 1981. A structural approach to operational semantics. (1981).

[41] Casper Bach Poulsen and Peter D Mosses. 2017. Flag-based big-step semantics. *Journal of Logical and Algebraic Methods in Programming* 88 (2017), 174–190.

[42] Amr Sabry and Matthias Felleisen. 1992. Reasoning about programs in continuation-passing style. *ACM SIGPLAN Lisp Pointers* 1 (1992), 288–298.

[43] Jeremy Siek. 2012. *Big-step, diverging or stuck?* https://siek.blogspot.com/2012/07/big-step-diverging-or-stuck.html

[44] Martin Strecker. 2005. *Compiler verification for C0 (intermediate report).* Technical Report. Tech. rep., Université Paul Sabatier, Toulouse.

[45] Conrad Watt. 2018. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 53–65. doi:10.1145/3167082

[46] Andrew K Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and computation* 115, 1 (1994), 38–94.

[47] Angel Zúñiga and Gemma Bel-Enguix. 2022. On Coevaluation Behavior and Equivalence. *Mathematics* 10, 20 (2022), 3800.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\,\textsc{T-Var} \qquad \frac{\Gamma, f : \tau \to \sigma, x : \tau \vdash e : \sigma}{\Gamma \vdash \mathsf{fun}\{f, x.\, e\} : \tau \to \sigma}\,\textsc{T-Lam} \qquad \frac{}{\Gamma \vdash \mathsf{Z} : \mathsf{nat}}\,\textsc{T-Zero} \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{S}[e] : \mathsf{nat}}\,\textsc{T-Succ}$$

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash \mathsf{app}[e_1, e_2] : \tau}\,\textsc{T-App} \qquad \frac{\Gamma \vdash e_3 : \mathsf{nat} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma, x : \mathsf{nat} \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}\{e_1; x.e_2\}[e_3] : \tau}\,\textsc{T-Case} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{eff}\{a; e\} : \tau}\,\textsc{T-Eff}$$

Fig. 28. Typing rules for PCF

A Types

This section contains types and typing rules for PCF.

The only types of PCF are natural numbers and functions, formalized by the following grammar:

$$\tau ::= \mathsf{nat} \mid \tau_1 \to \tau_2$$

The typign rules are given in Figure 28. The typing rules assign types to expressions using the judgment $\Gamma \vdash e : \tau$, which means that the expression e has type τ given the typing assumptions of the typing context Γ.

The typing rules are standard, with the exception of T-Eff, which is not used in the current pure setting. (It is relevant to the effectful setting of Section 5.) Note that types of expressions may not be unique since functions exist like the identity $\mathsf{fun}\{\_, x.\, x\}$ which can be typed as $\tau \to \tau$ for any τ.

B Big-Stop Examples

Here we collect some example derivations using big-stop.

B.1 Pure

To illustrate some big-stop derivations and show how big-stop operational semantics differ from an alternative approach based on "coevaluation", we consider two expressions extracted from the work of Leroy and Grall [35]. (For more about this system, see section 9.)

B.1.1 Pure Example 1. First we consider the following expression of type $\mathsf{nat} \to \mathsf{nat}$:

$$\mathsf{app}[\mathsf{fun}\{\_, x.\, \mathsf{Z}\}, \mathsf{app}[\mathsf{fun}\{f, y.\, \mathsf{app}[f, y]\}, \mathsf{Z}]]$$

This expression is our language's well-typed version of the expression $(\lambda x.\, 0)\omega$, which Leroy and Grall's system coevaluates to 0 [35]. In typical eager small-step semantics, however, this expression never reduces to a value. Instead, it loops forever while attempting to evaluate the argument $\mathsf{app}[\mathsf{fun}\{f, y.\, \mathsf{app}[f, y]\}, \mathsf{Z}]$.

The big-stop system captures the looping behaviour of the expression in the following ways. Firstly, it allows for a trivial derivation that the expression big-stops to itself via applying St-Stop(0). Secondly, it is able to conclude the same big-stop judgment nontrivially using a progressing derivation, as guaranteed by Theorem 12. To this end, let $e = \mathsf{fun}\{f, y.\, \mathsf{app}[f, y]\}$. (For space, we elide premisses of the form v *val*.)

$$\dfrac{\dfrac{}{\mathsf{fun}\{\_,x.\,\mathsf{Z}\}}\;\textsc{St-Stop}(0) \qquad \dfrac{\dfrac{}{e \mathrel{\underline{\Downarrow}} e}\;\textsc{St-Stop}(0) \quad \dfrac{}{\mathsf{Z} \mathrel{\underline{\Downarrow}} \mathsf{Z}}\;\textsc{St-Stop}(0) \quad \dfrac{}{\mathsf{Z}\ \mathit{val}} \quad \dfrac{}{\mathsf{app}[e,\mathsf{Z}] \mathrel{\underline{\Downarrow}} \mathsf{app}[e,\mathsf{Z}]}\;\textsc{St-Stop}(0)}{\mathsf{app}[e,\mathsf{Z}] \mathrel{\underline{\Downarrow}} \mathsf{app}[e,\mathsf{Z}]}\;\textsc{St-App} \atop \vdots \qquad \dfrac{}{\mathsf{fun}\{\_,x.\,\mathsf{Z}\}\ \mathit{val}}}{\mathsf{app}[\mathsf{fun}\{\_,x.\,\mathsf{Z}\},\mathsf{app}[e,\mathsf{Z}]] \mathrel{\underline{\Downarrow}} \mathsf{app}[\mathsf{fun}\{\_,x.\,\mathsf{Z}\},\mathsf{app}[e,\mathsf{Z}]]}\;\textsc{St-Stop}(2)$$

In fact, any number of applications of the progressing rule St-App may be applied, as the upper-right application of St-Stop(0) has the same conclusion as the whole St-App subtree. This circumstance shows that any finite amount of computation of this expression can be appropriately expressed in the big-stop system. In contrast, coevaluation captures the "infinite" computation that finishes the infinite looping of the argument and then tosses away that value using the constant function $\mathsf{fun}\{\_,x.\,\mathsf{Z}\}$, leaving the result of Z. In effect, this behaviour of coevaluation seems to ignore the distinction between call-by-value and call-by-name evaluation order.

B.1.2 Pure Example 2. The second example is the following expression of type $\mathsf{nat} \to \mathsf{nat}$ found by A. Filinski [35].

$$\mathsf{app}[\mathsf{fun}\{f,x.\,\mathsf{fun}\{\_,y.\,\mathsf{app}[\mathsf{app}[f,x],y]\}\},\mathsf{Z}]$$

This expression does not coevaluate at all in Leroy and Grall's system, but it does have big-stop derivations. Letting e' be $\mathsf{fun}\{f,x.\,\mathsf{fun}\{\_,y.\,\mathsf{app}[\mathsf{app}[f,x],y]\}\}$, the following is such a progressing derivation. This derivation can also be extended to be arbitrarily deep.

$$\dfrac{\dfrac{}{e' \mathrel{\underline{\Downarrow}} e'}\;\textsc{St-Stop}(0) \quad \dfrac{}{\mathsf{Z} \mathrel{\underline{\Downarrow}} \mathsf{Z}}\;\textsc{St-Stop}(0) \quad \dfrac{}{\mathsf{Z}\ \mathit{val}} \quad {\dfrac{}{\mathsf{fun}\{\_,y.\,\mathsf{app}[\mathsf{app}[e',\mathsf{Z}],y]\} \mathrel{\underline{\Downarrow}} \mathsf{fun}\{\_,y.\,\mathsf{app}[\mathsf{app}[e',\mathsf{Z}],y]\}}\;\textsc{St-Stop}(0) \atop \vdots}}{\mathsf{app}[e',\mathsf{Z}] \mathrel{\underline{\Downarrow}} \mathsf{fun}\{\_,y.\,\mathsf{app}[\mathsf{app}[e',\mathsf{Z}],y]\}}\;\textsc{St-App}$$

B.2 Effectful

B.2.1 Application to Resource Analysis 1. To demonstrate the use of the effectful big-stop system, consider reasoning about the memory requirements of some function f. One might care to ensure that f's application does not require more memory than is available, say, 2 cells of memory. One could approach this task with effects by using the effect "*alloc*" to represent the allocation of one new cell of memory. Then the derivation of a judgment like $\mathsf{app}[f,v] \overset{*}{\mapsto} e \rightsquigarrow (\mathit{alloc})^2$ would show that exactly 2 memory cells are required to reduce to e given argument v, and the desired memory safety property would be formalized with the statement $\mathsf{app}[f,v] \overset{*}{\mapsto} e \rightsquigarrow (\mathit{alloc})^n \implies n \leq 2$. In this way, effects can be more useful to consider than the actual results of computation.

Using only effectful big-step (not stop) semantics, a similar memory-safety property can be stated: for all values $v : \mathsf{nat}$, $\mathsf{app}[f,v] \Downarrow v' \rightsquigarrow (\mathit{alloc})^n \implies n \leq 2$. However, this new statement only captures the memory behaviour of terminating computations. It misses nonterminating behaviour of functions like the following:

$$\mathsf{fun}\{f,x.\,\mathsf{case}\{\mathsf{Z};y.\mathsf{app}[\mathsf{fun}\{g,z.\,\mathsf{eff}\{\mathit{alloc};\mathsf{app}[g,z]\}\},\mathsf{Z}]\}[x]\}$$

Depending upon whether the argument to the function f is zero, f either allocates no memory before immediately terminating or allocates infinite memory while looping forever. The latter behaviour will clearly surpass 5 cells and thus is unsafe. However, the big-step memory-safety statement $\texttt{app}[f, v] \Downarrow v' \rightsquigarrow (alloc)^n \implies n \leq 2$ is still true because the infinite looping behaviour never results in a value v'. Big-step semantics is unsuited to formalizing the property of interest.

Using big-stop semantics, however, the property $\texttt{app}[f, v] \Downarrow e \rightsquigarrow (alloc)^n \implies n \leq 2$ actually expresses what we want. Moreover, this statement can be shown false with the following derivation scheme capable of inferring $\texttt{app}[f, v] \Downarrow e \rightsquigarrow (alloc)^n$ for any n, where $v = \texttt{S}[\texttt{Z}]$, $e = \texttt{app}[e', \texttt{Z}]$, and $e' = \texttt{fun}\{g, z.\,\texttt{eff}\{alloc; \texttt{app}[g, z]\}\}$.

$$\dfrac{\dfrac{}{f \Downarrow f \rightsquigarrow 1}\,\textsc{StE-Stop}(0) \quad \begin{array}{c}\dfrac{}{\texttt{S}[\texttt{Z}] \Downarrow \texttt{S}[\texttt{Z}] \rightsquigarrow 1}\,\textsc{StE-Stop}(0)\\ \vdots\end{array} \quad \dfrac{}{\texttt{S}[\texttt{Z}]\ val} \quad \begin{array}{c}\dfrac{\dfrac{}{\texttt{S}[\texttt{Z}] \Downarrow \texttt{S}[\texttt{Z}] \rightsquigarrow 1}\,\textsc{StE-Stop}(0) \quad \dfrac{}{\texttt{S}[\texttt{Z}]\ val} \quad \mathcal{D}(n)}{\texttt{case}\{\texttt{Z}; y.\texttt{app}[e', \texttt{Z}]\}[\texttt{S}[\texttt{Z}]] \Downarrow \texttt{app}[e', \texttt{Z}] \rightsquigarrow (alloc)^n}\,\textsc{StE-CaseS}\\ \vdots\end{array}}{\texttt{app}[\texttt{fun}\{f, x.\,\texttt{case}\{\texttt{Z}; y.\texttt{app}[e', \texttt{Z}]\}[x]\}, \texttt{S}[\texttt{Z}]] \Downarrow \texttt{app}[e', \texttt{Z}] \rightsquigarrow (alloc)^n}\,\textsc{StE-App}$$

where for $n > 0$, $\mathcal{D}(n)$ is given by the following derivation

$$\dfrac{\dfrac{}{e' \Downarrow e' \rightsquigarrow 1}\,\textsc{StE-Stop}(0) \quad \begin{array}{c}\dfrac{}{\texttt{Z} \Downarrow \texttt{Z} \rightsquigarrow 1}\,\textsc{StE-Stop}(0)\\ \vdots\end{array} \quad \dfrac{}{\texttt{Z}\ val} \quad \begin{array}{c}\dfrac{\mathcal{D}(n-1)}{\texttt{eff}\{alloc; \texttt{app}[e', \texttt{Z}]\} \Downarrow \texttt{app}[e', \texttt{Z}] \rightsquigarrow (alloc)^n}\,\textsc{StE-Eff}\\ \vdots\end{array}}{\texttt{app}[e', \texttt{Z}] \Downarrow \texttt{app}[e', \texttt{Z}] \rightsquigarrow (alloc)^n}\,\textsc{StE-App}$$

and $\mathcal{D}(0)$ is the following derivation.

$$\dfrac{\textsc{StE-Stop}(0)}{\texttt{app}[e', \texttt{Z}] \Downarrow \texttt{app}[e', \texttt{Z}] \rightsquigarrow 1}$$

In this way, big-stop semantics allows one to reason correctly about nonterminating behaviour.

B.2.2 Application to Resource Analysis 2. Now consider verifying that memory requirement of the following function f does not exceed 2 cells of memory on any input value $v : \texttt{nat}$.

$$\texttt{fun}\{f, x.\,\texttt{eff}\{alloc; \texttt{case}\{\texttt{Z}; y.\texttt{app}[\texttt{fun}\{g, z.\,\texttt{app}[g, z]\}, \texttt{Z}]\}[x]\}\}$$

Unlike the previous example, this function actually does stay under 2 cells of memory no matter the input; it is true that $\texttt{app}[f, v] \Downarrow v' \rightsquigarrow (alloc)^n \implies n \leq 2$ for all values $v : \texttt{nat}$. In fact, the function only ever requires one allocation.

However, big-step semantics is unable to properly verify this bound, again due to being unable to reason about the nonterminating execution that occurs when the input v is nonzero. Big-*stop* semantics, on the other hand, can verify this bound without issue. Such a proof would look something like the following, where $\omega = \texttt{app}[\texttt{fun}\{g, z.\,\texttt{app}[g, z]\}, \texttt{Z}]$.

Suppose $v = \texttt{Z}$. Then the following derivation shows that only 1 allocation is required to reach the value Z.

$$
\dfrac{
\dfrac{}{f \Downarrow f \leadsto 1}\,\textsc{StE-Stop}(0)
\qquad
\begin{array}{c}\dfrac{}{\mathsf{Z} \Downarrow \mathsf{Z} \leadsto 1}\,\textsc{StE-Stop}(0)\\ \vdots\end{array}
\qquad
\dfrac{}{\mathsf{Z}\ \mathit{val}}
\qquad
\begin{array}{c}
\dfrac{\dfrac{\dfrac{}{\mathsf{Z} \Downarrow \mathsf{Z} \leadsto 1}\,\textsc{StE-Stop}(0) \qquad \dfrac{}{\mathsf{Z} \Downarrow \mathsf{Z} \leadsto 1}\,\textsc{StE-Stop}(0)}{\mathsf{case}\{\mathsf{Z}; y.\omega\}[\mathsf{Z}] \Downarrow \mathsf{Z} \leadsto 1}\,\textsc{StE-CaseZ}}{\mathsf{eff}\{\mathit{alloc}; \mathsf{case}\{\mathsf{Z}; y.\omega\}[\,]\} \Downarrow \mathsf{Z} \leadsto \mathit{alloc}}\,\textsc{StE-Eff}\\
\vdots
\end{array}
}{\mathsf{app}[f, \mathsf{Z}] \Downarrow \mathsf{Z} \leadsto \mathit{alloc}}\,\textsc{StE-App}
$$

Now suppose instead that $v = \mathsf{S}[v'']$ for some value $v'' : \mathtt{nat}$. Observe that arbitrarily deep progressing derivations can be made for which no more than 1 allocation is required. Deep enough derivations take the following form:

$$
\dfrac{
\dfrac{}{f \Downarrow f \leadsto 1}\,\textsc{StE-Stop}(0)
\qquad
\begin{array}{c}\dfrac{}{\mathsf{S}[v''] \Downarrow \mathsf{S}[v''] \leadsto 1}\,\textsc{StE-Stop}(0)\\ \vdots\end{array}
\qquad
\dfrac{}{\mathsf{S}[v'']\ \mathit{val}}
\qquad
\begin{array}{c}
\dfrac{\dfrac{\dfrac{}{\mathsf{S}[v''] \Downarrow \mathsf{S}[v''] \leadsto 1}\,\textsc{StE-Stop}(0) \qquad \mathcal{D}(n)}{\mathsf{case}\{\mathsf{Z}; y.\omega\}[\mathsf{S}[v'']] \Downarrow \omega \leadsto 1}\,\textsc{StE-CaseS}}{\mathsf{eff}\{\mathit{alloc}; \mathsf{case}\{\mathsf{Z}; y.\omega\}[\mathsf{S}[v'']]\} \Downarrow \omega \leadsto \mathit{alloc}}\,\textsc{StE-Eff}\\
\vdots
\end{array}
}{\mathsf{app}[f, \mathsf{S}[v'']] \Downarrow \omega \leadsto \mathit{alloc}}\,\textsc{StE-App}
$$

where for $n > 0$, $\mathcal{D}(n)$ is given by the following derivation

$$
\dfrac{
\dfrac{}{\mathsf{fun}\{g, z.\,\mathsf{app}[g, z]\} \Downarrow \mathsf{fun}\{g, z.\,\mathsf{app}[g, z]\} \leadsto 1}\,\textsc{StE-Stop}(0)
\qquad
\begin{array}{c}\dfrac{}{\mathsf{Z} \Downarrow \mathsf{Z} \leadsto 1}\,\textsc{StE-Stop}(0)\\ \vdots\end{array}
\qquad
\mathcal{D}(n-1)
}{\omega \Downarrow \omega \leadsto 1}\,\textsc{StE-App}
$$

and $\mathcal{D}(0)$ is the following derivation.

$$
\dfrac{}{\omega \Downarrow \omega \leadsto 1}\,\textsc{StE-Stop}(0)
$$

The only other remaining big-stop derivations arise from using StE-Stop to cut the previously given derivations short. In any case, however, such shorter derivations can require no more allocations than the deeper derivations.

This proof sketch also demonstrates how big-stop semantics can make use of both big-step and small-step reasoning techniques when convenient. The first case used typical big-step style reasoning to jump straight to the resulting value with no fuss. The second case used typical small-step style reasoning to show an evaluation loop. The big-stop system is ergnomically convenient because it has access to both kinds of reasoning techniques.

C Additional Imperative Formalism

This section contains some extra semantic formalizations related to the imperative language from Section 8.

$$\textsc{BI2-Bind}\ \frac{\sigma_1(a) = z}{\langle x := a \mid \sigma\rangle \Downarrow [z/x]\sigma} \qquad \textsc{BI2-Seq}\ \frac{\sigma_1(a) = 0 \quad \langle s_1 \mid \sigma_1\rangle \Downarrow \sigma_2 \quad \langle s_2 \mid \sigma_2\rangle \Downarrow \sigma_3}{\langle s_1; s_2 \mid \sigma_1\rangle \Downarrow \sigma_3} \qquad \textsc{BI2-Skip}\ \frac{}{\langle \mathtt{skip} \mid \sigma\rangle \Downarrow \sigma}$$

$$\textsc{BI2-Then}\ \frac{\sigma(a) \neq 0 \quad \langle s \mid \sigma\rangle \Downarrow \sigma'}{\langle \mathtt{if}\{a\}\mathtt{then}\{s\} \mid \sigma\rangle \Downarrow \sigma'} \qquad \textsc{BI2-Else}\ \frac{\sigma_1(a) = 0}{\langle \mathtt{if}\{a\}\mathtt{then}\{s\} \mid \sigma\rangle \Downarrow \sigma} \qquad \textsc{BI2-Done}\ \frac{\sigma_1(a) = 0}{\langle \mathtt{while}\{a\}\mathtt{do}\{s\} \mid \sigma\rangle \Downarrow \sigma}$$

$$\textsc{BI2-Do}\ \frac{\sigma_1(a) \neq 0 \quad \langle s \mid \sigma_1\rangle \Downarrow \sigma_2 \quad \langle \mathtt{while}\{a\}\mathtt{do}\{s\} \mid \sigma_2\rangle \Downarrow \sigma_3}{\langle \mathtt{while}\{a\}\mathtt{do}\{s\} \mid \sigma_1\rangle \Downarrow \sigma_3}$$

Fig. 29. Big-step semantics for the imperative language

$$\textsc{StI2-Bind}\ \frac{\sigma_1(a) = z}{\langle x := a \mid \sigma\rangle \Downarrow [z/x]\sigma} \qquad \textsc{StI2-Seq}\ \frac{\sigma_1(a) = 0 \quad \langle s_1 \mid \sigma_1\rangle \Downarrow \sigma_2 \quad \langle s_2 \mid \sigma_2\rangle \Downarrow \sigma_3}{\langle s_1; s_2 \mid \sigma_1\rangle \Downarrow \sigma_3} \qquad \textsc{StI2-Skip}\ \frac{}{\langle \mathtt{skip} \mid \sigma\rangle \Downarrow \sigma}$$

$$\textsc{StI2-Then}\ \frac{\sigma(a) \neq 0 \quad \langle s \mid \sigma\rangle \Downarrow \sigma'}{\langle \mathtt{if}\{a\}\mathtt{then}\{s\} \mid \sigma\rangle \Downarrow \sigma'} \qquad \textsc{StI2-Else}\ \frac{\sigma_1(a) = 0}{\langle \mathtt{if}\{a\}\mathtt{then}\{s\} \mid \sigma\rangle \Downarrow \sigma} \qquad \textsc{StI2-Done}\ \frac{\sigma_1(a) = 0}{\langle \mathtt{while}\{a\}\mathtt{do}\{s\} \mid \sigma\rangle \Downarrow \sigma}$$

$$\textsc{StI2-Do}\ \frac{\sigma_1(a) \neq 0 \quad \langle s \mid \sigma_1\rangle \Downarrow \sigma_2 \quad \langle \mathtt{while}\{a\}\mathtt{do}\{s\} \mid \sigma_2\rangle \Downarrow \sigma_3}{\langle \mathtt{while}\{a\}\mathtt{do}\{s\} \mid \sigma_1\rangle \Downarrow \sigma_3} \qquad \textsc{StI2-Stop}\ \frac{}{\langle s \mid \sigma\rangle \Downarrow \mathtt{freeze}(\sigma)}$$

Fig. 30. Alternative big-stop semantics for the imperative language

Consider the typical big-step judgment of the form $\langle s \mid \sigma\rangle \Downarrow \sigma'$. This form of judgment is slightly trickier to adapt to big-stop because big-stop leverages judgements of the form $\langle s \mid \sigma\rangle \Downarrow \langle \mathtt{skip} \mid \sigma'\rangle$ to distinguish total from partial evaluations. For example, the use of skip distinguishes the applicability of BI-Do from that of StI-Do.

Nonetheless, this problem can be overcome with the annihilator trick of Section 7. An appropriate new annihilator effect is given by the function freeze. This function renders a state immutable so that $[z/x]\mathtt{freeze}(\sigma) = \mathtt{freeze}(\sigma)$. The result of this approach is the judgment

$$\langle s \mid \sigma\rangle \Downarrow \sigma'$$

defined by the rules of Figure 30. These rules are comprised of the standard big-step rules and the additional rule StI2-Stop.

$$\textsc{StI2-Stop}\ \frac{}{\langle s \mid \sigma\rangle \Downarrow \mathtt{freeze}(\sigma)}$$

These alternative big-stop semantics agree with the small-step semantics in the following way:

Theorem 25 (Alternative Imperative Equivalence). *For all statements s and states σ, σ',*

$$\langle s \mid \sigma\rangle \Downarrow \sigma' \iff \exists s'. \langle s \mid \sigma\rangle \overset{*}{\mapsto} \langle s' \mid \sigma'\rangle$$

Proof. This property follows from induction over the derivations. □