CPS Semantics: Smoother Nondeterminism in Operational Semantics

ARTHUR CHARGUÉRAUD, Inria & Université de Strasbourg, CNRS, ICube, France
ADAM CHLIPALA, MIT CSAIL, USA
ANDRES ERBSEN, MIT CSAIL, USA
SAMUEL GRUETTER, MIT CSAIL, USA

This paper introduces the CPS-big-step and CPS-small-step judgments. These two judgments describe operational semantics by relating starting states to sets of outcomes rather than to individual outcomes. A single derivation of these semantics for a particular starting state and program describes all possible nondeterministic executions, whereas in traditional small-step and big-step semantics, each derivation only talks about one single execution. We demonstrate how this restructuring allows for straightforward modeling of languages featuring both nondeterminism and undefined behavior. Specifically, our semantics inherently assert *safety*, i.e. they guarantee that none of the execution branches gets stuck, while traditional semantics need either a separate judgment or additional error markers to specify safety in the presence of nondeterminism. Applications presented include proofs of type soundness for lambda calculi, mechanical derivation of reasoning rules for program verification, and a forward proof of compiler correctness for terminating but potentially nondeterministic programs. All results in this paper have been formalized in Coq.

1 INTRODUCTION

Today, most projects in rigorous reasoning about programming languages begin with an operational semantics (or maybe several), with proofs of key lemmas proceeding by induction on derivations of the semantics judgement. An extensive toolbox has been built up for formulating these relations, with common wisdom on the style to choose for each situation. With decades having passed since operational semantics became the standard technique in the 1980s, one might expect that the base of wisdom is sufficient. However, across projects using a variety of proof styles, we noticed seemingly unrelated weaknesses, and we were surprised to find a common solution.

Our new approach is based on operational semantics that relate starting states to their sets of possible outcomes, rather than to individual outcomes. Our big-step judgment takes the form $t/s \Downarrow Q$ and asserts that every possible evaluation starting from the configuration t/s reaches a final configuration that belongs to the set Q. This set Q is isomorphic to a postcondition from a Hoare triple, and it may also be viewed as a continuation that applies to the reachable final configurations—hence the name *continuation-passing style (CPS)* big-step semantics.

Our CPS-small-step judgment takes the form $t/s \longrightarrow P$. It asserts both that the configuration t/s can take one reduction step and that, for any step it might take, the resulting configuration belongs to the set P. On top of this judgment, we define the *eventually* judgment $t/s \longrightarrow^{\Diamond} P$, which asserts that every possible evaluation of t/s is safe and eventually reaches a configuration in the set P.

In either case, induction on derivations of the CPS-semantics judgement follows the flow of execution of an arbitrary program, with nondeterministic choices resulting in universally quantified induction hypotheses at the step where the nondeterministic choice is made. As we argue throughout the paper, this feature is particularly helpful for reasoning about languages that combine nondeterminsm and either undefined behavior or the possibility of crashing. But first, let us motivate CPS semantics by presenting four cracks we found in the convenient application of big-step and small-step operational semantics.

Authors' addresses: Arthur Charguéraud, Inria & Université de Strasbourg, CNRS, ICube, France, arthur.chargueraud@inria.fr; Adam Chlipala, MIT CSAIL, Cambridge, MA 02139, USA, adamc@csail.mit.edu; Andres Erbsen, MIT CSAIL, Cambridge, MA 02139, USA, andreser@mit.edu; Samuel Gruetter, MIT CSAIL, Cambridge, MA 02139, USA, gruetter@mit.edu.

Inconvenience #1: crashes and nondeterminism. In an impure language, an execution may crash, for instance due to a division by zero or an out-of-bounds array access. In a nondeterministic language, some executions may produce crashes while others do not. Thus, for an impure, nondeterministic language, the existence of a traditional big-step derivation for a starting configuration is not a proof that crashing is impossible.

How do we fix the problem? A popular but cumbersome approach is to add crashes as explicit outcomes (written err in the rules below), so that we can state theorems ruling out crashes. For example, if the semantics of an impure functional language includes the rule BIG-LET, it needs to be augmented with two additional rules for propagating errors: BIG-LET-ERR-1 and BIG-LET-ERR-2.

$$\frac{t_1/s \Downarrow v_1/s' \qquad ([v_1/x]\,t_2)/s' \Downarrow v/s''}{(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)/s \Downarrow v/s''}\ \textsc{big-let}$$

$$\frac{t_1/s \Downarrow \mathsf{err}}{(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)/s \Downarrow \mathsf{err}}\ \textsc{big-let-err-1} \qquad \frac{t_1/s \Downarrow v_1/s' \qquad ([v_1/x]\,t_2)/s' \Downarrow \mathsf{err}}{(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)/s \Downarrow \mathsf{err}}\ \textsc{big-let-err-2}$$

The set of inference rules grows significantly, and the very type signature of the relation is complicated. Could we find a way to reason, in big-step style, about the absence of crashes in nondeterministic languages without introducing error-propagation rules?

Inconvenience #2: termination and nondeterminism. In a nondeterministic language, a total correctness Hoare triple, written $^{\mathsf{total}}\{H\}\ t\ \{Q\}$, asserts that in any state satisfying the precondition H, *any* execution of the term t terminates and reaches a final state satisfying the postcondition Q. In fundational approaches, the definition of Hoare triples need to be either defined in terms of, or at least formally related to, the operational semantics of the language.

When the (nondeterministic) semantics is expressed using the standard small-step relation, there are two classical approaches to defining total correctness Hoare triples. The first one involves bounding the length of the execution. This approach not only involves tedious manipulation of integer bounds, it is also restricted to finitely-branching forms of nondeterminism. The second approach is to define total correctness as the conjunction of a partial correctness property (if t terminates, then it satisfies the postcondition), and of a separate, inductively-defined termination judgment. With both of these approaches, deriving reasoning rules for total correctness Hoare triples reveals much more tedious than in the case of partial correctness.

One may hope for simpler proofs using a big-step judgment. Indeed, Hoare triples inherently have a big-step flavor. Moreover, for deterministic, sequential languages, the most direct way to derive reasoning rules for Hoare triples is from the big-step evaluation rules. Yet, when the semantics of a nondeterministic language is expressed using a standard big-step judgment, we do not know of any direct way to capture the fact that *all* executions terminate. Could we find a way to define total correctness Hoare triples with respect to a big-step style, nondeterministic semantics, in a way that leads to simple proofs of the Hoare logic rules?

Inconvenience #3: large inversions in type-soundness proofs. Type soundness asserts that if a closed term is well-typed, then none of its possible evaluations gets stuck. A type-soundness proof in the syntactic style [Wright and Felleisen 1994] reduces to a pair of lemmas: preservation and progress.

$$\begin{array}{llll} \textsc{preservation:} & E \vdash t : T \ \wedge\ t \longrightarrow t' & \Rightarrow & E \vdash t' : T \\ \textsc{progress:} & \emptyset \vdash t : T & \Rightarrow & (\mathsf{isvalue}\,t)\ \vee\ (\exists t'.\ t \longrightarrow t') \end{array}$$

The Wright and Felleisen approach, although widely used, suffers from two limitations that can be problematic at the scale of real-world language with hundreds of syntactic constructs.

The first limitation is that this approach requires performing two inductions over the typing judgment. Nontrivial language constructs are associated with nontrivial statements of their induction hypotheses, for which the same manual work needs to be performed twice, once in the preservation proof and once in the progress proof. Factoring out the cases makes a huge difference in terms of proof effort and maintainability.

The second limitation is associated with the case inspection involved in the preservation proof. Concretely, for each possible rule that derives the typing judgment ($E \vdash t : T$), one needs to select the applicable rules that can derive the reduction rule ($t \longrightarrow t'$) for that same term t. Typically, only a few reduction rules are applicable. Yet, the trouble is that a fully rigorous checking of the proof must still inspect all of those cases to confirm their irrelevance. A direct Coq proof, of the form "`induction H1; inversion H2`", results in a proof term of size quadratic in the size of the language. A proof of linear size would be much more satisfying, because we expect to handle each possible transition at most once.

Interestingly, in the particular case of a deterministic language, there exists a strategy [Rompf and Amin 2016][1] for deriving type soundness through a *single* inductive proof, which moreover avoids the quadratic case inspection. The key idea is to carry out an induction over the following statement: a well-typed term is either a value or it can step to a term that admits the same type.

$$\emptyset \vdash t : T \quad \Rightarrow \quad \big(\text{isvalue}\, t\big) \;\vee\; \big(\exists t'.\, (t \longrightarrow t') \wedge (\emptyset \vdash t' : T)\big)$$

Could we generalize this approach to the case of nondeterministic languages?

Inconvenience #4: simulation arguments with nondeterminism and undefined behavior. Many compiler transformations map source programs to target programs that require more steps to accomplish the same work, because they must make do with lower-level primitives. Intuitively, we like to think of a compiler transformation being correct in terms of *forward simulation*: the transformation maps each step from the source program to a number of steps in the target program. Yet, in the context of a nondeterministic language, such a result is famously insufficient even in the special case of safely terminating programs. Concretely, compiler correctness requires to show *all* possible behaviors of the target program correspond to possible behaviors of the source program. A tempting approach is to establish a *backward simulation*, by showing that any step in the target program can be matched by some number of steps in the source program. The trouble is that all intermediate target-level states during a single source-level step need to be related to a source-level state, severely complicating the simulation relation.

To avoid that hassle, most compilation phases from CompCert [Leroy 2009] are carried out on *deterministic* intermediate languages, for which forward simulation implies backward simulation. Yet, many realistic languages (C included) are not naturally seen as deterministic. CompCert involves special effort to maintain determinism, through its celebrated memory model. Rather than revealing pointers as integers, CompCert semantics allocate pointers deterministically, taking care to trigger undefined behavior for any coding pattern that would be sensitive to the literal values of pointers. As a result, any compiler transformations that modify allocation order require the complex machinery of memory injections, to connect executions that use different deterministic pointer values. Could we instead retain the simplicity of forward simulation while keeping nondeterminism explicit? In particular, could we support intricate pointer arithmetic with no special semantic effort?

[1]We are aware that the factorization technique for preservation and progress was used in the formalization of DOT [Rompf and Amin 2016]. It might have also appeared in other work that we are not aware of.

One answer: CPS semantics. We answer all of these wishes affirmatively with a new kind of semantics, which we name CPS-big-step and CPS-small-step semantics. These semantics are easy to embed in proof assistants, with good ergonomics. In fact, they have already been battle-tested in a compiler verification that includes connections to correctness of both applications and hardware, for end-to-end functional correctness of a simple embedded system [Erbsen et al. 2021], where undefined behavior arises for the usual C-style reasons, and nondeterminism arises from communication with devices.[2] However, applicability of CPS semantics is much broader than in just that kind of setting, as we hope to demonstrate with several examples in this paper.

Contributions. Our contributions are as follows.

- In Section 2, we introduce the CPS-big-step judgment, which can be defined either inductively, to capture termination of all executions, or coinductively, in *partial correctness* fashion. We also state and prove properties about the judgment, including the notion of smallest and largest admissible sets of outcomes.
- In Section 3, we introduce the CPS-small-step judgment, as well as the *eventually* judgment defined on top of it, and three practical reasoning rules associated with these judgments.
- In Section 4, we present type-soundness proofs carried out with respect to either CPS-small-step or CPS-big-step semantics. We explain how this improves over the state of the art, by solving the inconviences #1 and #3 described earlier on.
- In Section 5, we explain how the CPS-big-step judgment or the CPS-small-step eventually judgment can be used to define Hoare triples and weakest-precondition predicates. We consider both partial and total correctness, and we show how the associated reasoning rules can be established via one-line proofs.
- In Section 6, we demonstrate how CPS semantics can be used to prove that a compiler correctly compiles terminating programs, via forward-simulation proofs. We illustrate this possibility through two case studies carried out on a while-language. The first one, "heapification" of pairs, increases the amount of nondeterminism; it involves a CPS-big-step semantics for both the source and the target language. The second one, introduction of stack allocation, decreases the amount of nondeterminism; it involves a CPS-big-step semantics for the source language and a CPS-small-step smantics for the target language.
- In Section 7, we give seven possible interpretations of the CPS-big-step judgment, thereby exploring its relationship to existing concepts such as standard big-step semantics, CPS conversion, Hoare triples, weakest preconditions, typing judgments, big-step safety judgments, and reasoning rules for deterministic semantics.

Note that we leave it to future work to investigate how CPS semantics may be exploited to establish *full compiler correctness*, that is, not just the correctness of the compilation for terminating programs but also that of programs that may crash, diverge, or perform infinitely many I/O interactions.

2 CPS-BIG-STEP SEMANTICS

2.1 Definition of the CPS-Big-Step Judgment

Syntax of the language. We consider an imperative lambda-calculus, including a random-number generator rand. Both this operator and allocation are nondeterministic operations.

[2]In that PLDI paper [Erbsen et al. 2021], CPS semantics is sketched in less than 1.5 pages (Section 4). The present paper elaborates on interpretation and properties of the judgments, presents new correctness proofs for compiler transformations that resolve or introduce nondeterminism, and presents additional aspects of the code-generation proof. We also add important applications to type-soundness proofs and Hoare-logic constructions.

$$\textsc{big-val}\ \frac{}{v/s \Downarrow v/s} \qquad \textsc{big-app}\ \frac{v_1 = (\mu f.\lambda x.t) \qquad ([v_2/x]\,[v_1/f]\,t)/s \Downarrow v'/s'}{(v_1\,v_2)/s \Downarrow v'/s'} \qquad \textsc{big-if-true}\ \frac{t_1/s \Downarrow v'/s'}{(\text{if true then } t_1 \text{ else } t_2)/s \Downarrow v'/s'}$$

$$\textsc{big-if-false}\ \frac{t_2/s \Downarrow v'/s'}{(\text{if false then } t_1 \text{ else } t_2)/s \Downarrow v'/s'} \qquad \textsc{big-let}\ \frac{t_1/s \Downarrow v_1/s' \qquad ([v_1/x]\,t_2)/s' \Downarrow v/s''}{(\text{let } x = t_1 \text{ in } t_2)/s \Downarrow v/s''}$$

$$\textsc{big-add}\ \frac{}{(\text{add}\,n_1\,n_2)/s \Downarrow (n_1 + n_2)/s} \qquad \textsc{big-rand}\ \frac{0 \le m < n}{(\text{rand}\,n)/s \Downarrow m/s} \qquad \textsc{big-ref}\ \frac{p \notin \text{dom}\,s}{(\text{ref}\,v)/s \Downarrow p/(s[p := v])}$$

$$\textsc{big-free}\ \frac{p \in \text{dom}\,s}{(\text{free}\,p)/s \Downarrow tt/(s \setminus p)} \qquad \textsc{big-get}\ \frac{p \in \text{dom}\,s}{(\text{get}\,p)/s \Downarrow (s[p])/s} \qquad \textsc{big-set}\ \frac{p \in \text{dom}\,s}{(\text{set}\,p\,v)/s \Downarrow tt/(s[p := v])}$$

Fig. 1. Standard big-step semantics (for terms in A-normal form)

The grammar of the language appears next. The metavariable π ranges over primitive operations, v ranges over *closed* values, t ranges over terms, and x and f range over program variables. A value can be the unit value tt, a Boolean b, a natural number n, a pointer p, or a primitive or a closure.

We consider a grammar of closed values because it makes substitution easier to implement and reason about in a proof assistant.

$$\begin{array}{lcl} \pi & := & \text{add} \mid \text{rand} \mid \text{ref} \mid \text{free} \mid \text{get} \mid \text{set} \\ v & := & tt \mid b \mid n \mid p \mid \pi \mid (\mu f.\lambda x.t) \\ t & := & v \mid x \mid (t\,t) \mid \text{let}\,x = t\,\text{in}\,t \mid \text{if}\ t\ \text{then}\ t\ \text{else}\ t \mid \mu f.\lambda x.t \end{array}$$

For simplicity, we present evaluation rules by focusing first on programs in A-normal form. In an application $(t_1\,t_2)$, the two terms must be either variables or values. Similarly, the condition of an if-statement must be either a variable or a value, and likewise for arguments of primitive operations. We discuss in Appendix A the treatment of terms that are not in A-normal form.

Evaluation judgments. The standard big-step-semantics judgment for this language appears in Figure 1. States s are partial maps from pointers p to values v. The evaluation judgment $t/s \Downarrow v/s'$ asserts that the configuration t/s, made of a term t and an initial state s, may evaluate to the final configuration v/s', made of a value v and a final state s'.

The corresponding CPS-big-step semantics appears in Figure 2. Its evaluation judgment, written $t/s \Downarrow Q$, asserts that all possible evaluations starting from the configuration t/s reach final configurations that belong to the set Q. Observe how the standard big-step judgment $t/s \Downarrow v/s'$ describes the behavior of one possible execution of t/s, whereas the CPS-big-step judgment describes the behavior of all possible executions of t/s. The set Q that appears in $t/s \Downarrow Q$ corresponds to an overapproximation of the set of final configurations: it may contain configurations that are not actually reachable by executing t/s. We return to that aspect later on, at the end of §2.2.

The set Q contains pairs made of values and results. Such a set can be described equivalently by a predicate of type "val → state → Prop", or by a predicate of type "(val × state) → Prop". In this paper, in order to present definitions in the most idiomatic style, we use set-theoretic notation such as $(v, s) \in Q$ for stating semantics and typing rules, and we use the logic-oriented notation $Q\,v\,s$ when discussing program logics.

$$\textsc{cps-big-val}\ \frac{(v,s)\in Q}{v/s \Downarrow Q} \qquad \textsc{cps-big-app}\ \frac{v_1 = \mu f.\lambda x.t_1 \qquad ([v_1/f]\,[v_2/x]\,t_1)/s \Downarrow Q}{(v_1\,v_2)/s \Downarrow Q} \qquad \textsc{cps-big-if-true}\ \frac{t_1/s \Downarrow Q}{(\mathsf{if\ true\ then}\ t_1\ \mathsf{else}\ t_2)/s \Downarrow Q}$$

$$\textsc{cps-big-if-false}\ \frac{t_2/s \Downarrow Q}{(\mathsf{if\ false\ then}\ t_1\ \mathsf{else}\ t_2)/s \Downarrow Q} \qquad \textsc{cps-big-let}\ \frac{t_1/s \Downarrow Q_1 \qquad \big(\forall (v',s') \in Q_1.\ \ ([v'/x]\,t_2)/s' \Downarrow Q\big)}{(\mathsf{let}\,x = t_1\ \mathsf{in}\,t_2)/s \Downarrow Q}$$

$$\textsc{cps-big-add}\ \frac{(n_1+n_2,\ s)\in Q}{(\mathsf{add}\,n_1\,n_2)/s \Downarrow Q} \qquad \textsc{cps-big-rand}\ \frac{n>0 \qquad \big(\forall m.\ \ 0\le m<n\ \Rightarrow\ (m,s)\in Q\big)}{(\mathsf{rand}\,n)/s \Downarrow Q} \qquad \textsc{cps-big-ref}\ \frac{\forall p\notin \mathsf{dom}\,s.\ \ (p,\ s[p:=v])\in Q}{(\mathsf{ref}\,v)/s \Downarrow Q}$$

$$\textsc{cps-big-free}\ \frac{p\in\mathsf{dom}\,s \qquad (tt,\ s\setminus p)\in Q}{(\mathsf{free}\,p)/s \Downarrow Q} \qquad \textsc{cps-big-get}\ \frac{p\in\mathsf{dom}\,s \qquad (s[p],\ s)\in Q}{(\mathsf{get}\,p)/s \Downarrow Q} \qquad \textsc{cps-big-set}\ \frac{p\in\mathsf{dom}\,s \qquad (tt,\ s[p:=v])\in Q}{(\mathsf{set}\,p\,v)/s \Downarrow Q}$$

Fig. 2. CPS-big-step semantics (for terms in A-normal form)

Description of the evaluation rules. The base case is the rule CPS-BIG-VAL: a final configuration v/s satisfies the postcondition Q if this configuration belongs to the set Q.

The let-binding rule CPS-BIG-LET ensures that all possible evaluations of an expression $\mathsf{let}\,x = t_1\ \mathsf{in}\,t_2$ in state s terminate and satisfy the postcondition Q. First of all, we need all possible evaluations of t_1 to terminate. Let Q_1 denote (an overapproximation of) the set of results that t_1 may reach, as captured by the first premise $t_1/s \Downarrow Q_1$. For any configuration v'/s' in that set Q_1, we need all possible evaluations of the term $[v'/x]\,t_2$ in state s' to satisfy the postcondition Q. This property is captured by the second premise.

The evaluation rule CPS-BIG-ADD for an addition operation is almost like that of a value: it asserts that the evaluation of $\mathsf{add}\,n_1\,n_2$ in state s satisfies the postcondition Q if the pair $((n_1+n_2), s)$ belongs to the set Q. The nondeterministic rule CPS-BIG-RAND is more interesting. The term $\mathsf{rand}\,n$ evaluates safely only if $n > 0$. Under this assumption, its result, named m in the rule, may be any integer in the range $[0,n)$. Thus, to guarantee that every possible evaluation of $\mathsf{rand}\,n$ in a state s produces a result satisfying the postcondition Q, it must be the case that every pair of the form (m,s) with $m\in[0,n)$ belongs to the set Q.

The evaluation rule CPS-BIG-REF, which describes allocation at a nondeterministically chosen, fresh memory address, follows a similar pattern. For every possible new address p, the pair made of p and the extended state $s[p := v]$ needs to belong to the set Q. The remaining rules, CPS-BIG-FREE, CPS-BIG-GET and CPS-BIG-SET, are deterministic and follow the same pattern as CPS-BIG-ADD, only with a side condition $p \in \mathsf{dom}\,s$ to ensure that the address being manipulated does belong to the domain of the current state.

2.2 Properties of the CPS-Big-Step Judgment

In this section, we discuss some key properties of the CPS-big-step judgment $t/s \Downarrow Q$ that should help develop some intuition on it. Recall that Q denotes an overapproximation of the set of possible final configurations.

Total correctness. The predicate $t/s \Downarrow Q$ captures total correctness in the sense that it captures the conjunction of termination (all executions terminate) and partial correctness (if an execution terminates, then its final state satisfies the postcondition Q). Formally, let $t/s \Downarrow v/s'$ denote the standard big-step evaluation judgment, and let $\mathsf{terminates}(t, s)$ be a predicate that captures the fact that all executions of t/s terminate (a formal definition is given in §7.6). We prove:

$$\textsc{cps-big-step-iff-terminates-and-correct}:\quad t/s \Downarrow Q \iff \mathsf{terminates}(t,s) \;\wedge\; \Big(\forall v s'.\ (t/s \Downarrow v/s') \Rightarrow (v, s') \in Q\Big).$$

In particular, if we instantiate the postcondition Q with the *always-true* predicate, we obtain the predicate $t/s \Downarrow \{(v, s') \mid \mathsf{True}\}$, which captures only the termination property.

Consequence rule. The judgment $t/s \Downarrow Q$ still holds when the postcondition Q is replaced with a larger set. In other words, the postcondition can always be weakened, like in Hoare logic.

$$\textsc{cps-big-consequence}:\quad t/s \Downarrow Q \;\wedge\; Q \subseteq Q' \;\Rightarrow\; t/s \Downarrow Q'$$

Strongest postcondition. If the CPS-big-step judgment holds for at least one set, then there exists a smallest possible set Q for which $t/s \Downarrow Q$ holds. This set corresponds to the strongest possible postcondition Q, in the terminology of Hoare logic. Formally, if $t/s \Downarrow Q$ holds for at least one Q, then $t/s \Downarrow (\mathsf{strongest\text{-}post}\ t\ s)$ holds, where the strongest postcondition is equal to the intersection of all valid postconditions.

$$\mathsf{strongest\text{-}post}\ t\ s \;=\; \bigcap_{Q \,\mid\, (t/s \Downarrow Q)} Q \;=\; \Big\{(v, s') \;\Big|\; \forall Q,\ (t/s \Downarrow Q) \Longrightarrow (v, s') \in Q\Big\}$$

Outcome set is nonempty. Observe that the judgment $t/s \Downarrow Q$, as defined in Fig. 2, can only hold for a nonempty set Q. This property is a general requirement for a CPS-big-step semantics to make sense, and when designing CPS-big-step rules for a new language, one has to be careful to make sure it holds. For example, omitting the premise $n > 0$ in the rule cps-big-rand would lead to an ill-formed semantics, allowing $\mathsf{rand}\ 0$ to be related to any set Q, including the empty set.

No derivations for terms that may get stuck. The fact that $\mathsf{rand}\ 0$ is a stuck term is captured by the fact that $(\mathsf{rand}\ 0)/s \Downarrow Q$ does not hold for any Q. More generally, if one or more nondeterministic executions of t may get stuck, then $\forall Q.\ \neg\ (t/s \Downarrow Q)$.

2.3 Coinductive Interpretation of the CPS-Big-Step Judgment

Let $t/s \Downarrow^{\mathrm{co}} Q$ denote the judgment defined by the coinductive interpretation of the same set of rules as for the inductively defined judgment $t/s \Downarrow Q$, i.e., rules from Fig. 2. The coinductive interpretation allows for infinite derivation trees, thus the coinductive CPS-big-step judgment can be used to capture properties of nonterminating executions.

More precisely, the judgment $t/s \Downarrow^{\mathrm{co}} Q$ asserts that every possible execution of configuration t/s either diverges or terminates on a final configuration satisfying Q. In particular, this judgment rules out the possibility for an execution of t/s to get stuck, and it can be used to express type soundness, as detailed in §4. The judgment $t/s \Downarrow^{\mathrm{co}} Q$ can also be used to define partial-correctness Hoare triples, as detailed in §5.

Formally, we can relate the meaning of $t/s \Downarrow^{\mathrm{co}} Q$ to the small-step characterization of partial correctness as follows: for every execution prefix, the configuration reached is either a value satisfying the postcondition, or it is a term that can be further reduced. Below, $t/s \longrightarrow t'/s'$ denotes the standard small-step evaluation judgment (defined in Appendix C), and val denotes the

CPS-SMALL-APP
$$\frac{v_1 = (\mu f.\lambda x.t) \qquad ([v_2/x]\,[v_1/f]\,t,\ s) \in P}{(v_1\, v_2)/s \longrightarrow P}$$

CPS-SMALL-IF
$$\frac{b = \mathsf{true} \;\Rightarrow\; P\,(t_1, s) \qquad b = \mathsf{false} \;\Rightarrow\; P\,(t_2, s)}{(\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)/s \longrightarrow P}$$

CPS-SMALL-LET-CTX
$$\frac{t_1/s \longrightarrow P_1 \qquad \Big(\forall (t_1', s') \in P_1.\ \ ((\mathsf{let}\,x = t_1'\ \mathsf{in}\,t_2), s') \in P\Big)}{(\mathsf{let}\,x = t_1\ \mathsf{in}\,t_2)/s \longrightarrow P}$$

CPS-SMALL-LET
$$\frac{([v_1/x]\,t_2,\ s) \in P}{(\mathsf{let}\,x = v_1\ \mathsf{in}\,t_2)/s \longrightarrow P}$$

CPS-SMALL-ADD
$$\frac{(n_1 + n_2,\ s) \in P}{(\mathsf{add}\,n_1\,n_2)/s \longrightarrow P}$$

CPS-SMALL-RAND
$$\frac{n > 0 \qquad \Big(\forall m \in [0, n).\ \ (m, s) \in P\Big)}{(\mathsf{rand}\,n)/s \longrightarrow P}$$

CPS-SMALL-REF
$$\frac{\Big(\forall p \notin \mathsf{dom}\,s.\ \ (p, s[p := v]) \in P\Big)}{(\mathsf{ref}\,v)/s \longrightarrow P}$$

CPS-SMALL-FREE
$$\frac{p \in \mathsf{dom}\,s \qquad (tt,\ s \setminus p) \in P}{(\mathsf{free}\,p)/s \longrightarrow P}$$

CPS-SMALL-GET
$$\frac{p \in \mathsf{dom}\,s \qquad (s[p],\ s) \in P}{(\mathsf{get}\,p)/s \longrightarrow P}$$

CPS-SMALL-SET
$$\frac{p \in \mathsf{dom}\,s \qquad (tt,\ s[p := v]) \in P}{(\mathsf{set}\,p\,v)/s \longrightarrow P}$$

Fig. 3. CPS-small-step semantics (for terms in A-normal form)

constructor that injects values into the grammar of terms.

CO-CPS-BIG-IFF-SAFE-AND-CORRECT
$$t/s \Downarrow^{\mathrm{co}} Q \quad\Longleftrightarrow\quad \forall s't'.\ (t/s \longrightarrow^{*} t'/s') \Rightarrow \begin{array}{l}\Big(\exists v.\ t' = \mathsf{val}\,v \wedge (v, s) \in Q\Big) \\ \vee\ \Big(\exists t's'.\ t/s \longrightarrow t'/s'\Big)\end{array}$$

The judgment $t/s \Downarrow^{\mathrm{co}} Q$ can also be used to characterize divergence, by instantiating Q as the empty set: the predicate $t/s \Downarrow^{\mathrm{co}} \emptyset$ asserts that every possible execution of t/s diverges. Because the judgment $t/s \Downarrow^{\mathrm{co}} Q$ is covariant in Q, the predicate $t/s \Downarrow^{\mathrm{co}} \emptyset$ holds if and only if the predicate $t/s \Downarrow^{\mathrm{co}} Q$ holds for any Q. In summary, we formally characterize divergence as follows.

$$\mathsf{diverges}\,t\,s \;\equiv\; (t/s \Downarrow^{\mathrm{co}} \emptyset) \qquad\qquad \mathsf{diverges}\,t\,s \;\Longleftrightarrow\; \forall Q.\ (t/s \Downarrow^{\mathrm{co}} Q)$$

3 CPS-SMALL-STEP SEMANTICS

In this section, we introduce the CPS-small-step judgment, written $t/s \longrightarrow P$. We then introduce the *eventually* judgment, written $t/s \longrightarrow^{\Diamond} P$. We use these judgments for establishing type soundness (§4.1) and compiler verification results (§6.6).

3.1 The CPS-Small-Step Judgment

The CPS-small-step judgment, written $t/s \longrightarrow P$, asserts that the configuration t/s can take one reduction step, and that for any step it might take, the resulting configuration belongs to the set P. It is defined by the rules from Fig. 3. There is one per small-step transition. The interesting rules are those involving nondeterminism, namely CPS-SMALL-RAND and CPS-SMALL-REF, which follow a pattern similar to the corresponding CPS-big-step rules. Observe also how the rule CPS-SMALL-LET-CTX handles the case of a reduction that takes place in the evaluation context of a let-binding, by quantifying over an intermediate set of results named P_1.

We prove that the judgment $t/s \longrightarrow P$ captures the expected property w.r.t. the standard small-step judgment: the configuration t/s can make a step, and for every step it might take, it reaches a configuration in P.

$$\textsc{cps-small-step-iff-progress-and-correct}$$
$$t/s \longrightarrow P \quad\Longleftrightarrow\quad \Big(\exists t's'.\ t/s \longrightarrow t'/s'\Big) \wedge \Big(\forall t's'.\ t/s \longrightarrow t'/s' \ \Rightarrow\ (t',s') \in P\Big)$$

3.2 The “Eventually” Judgment

The judgment $t/s \longrightarrow^{\Diamond} P$ captures the property that every possible evaluation of t/s is safe and eventually reaches a configuration in the set P. Here, P denotes a set of configurations—it is not limited to being a set of *final* configurations like in the previous section. The judgment $t/s \longrightarrow^{\Diamond} P$ is defined inductively by the following two rules. The first one asserts that the judgment is satisfied if t/s belongs to P. The second one asserts that the judgment is satisfied if t/s is not stuck and that for any configuration t'/s' that it may reduce to, the predicate $t'/s' \longrightarrow^{\Diamond} P$ holds. The latter property is expressed using the CPS-small-step judgment $t/s \longrightarrow P'$, where P' denotes an overapproximation of the set of configurations t'/s' to which t/s may reduce.

$$\textsc{eventually-here}\ \frac{(t,s) \in P}{t/s \longrightarrow^{\Diamond} P} \qquad \textsc{eventually-step}\ \frac{t/s \longrightarrow P' \qquad \Big(\forall (t',s') \in P'.\ t'/s' \longrightarrow^{\Diamond} P\Big)}{t/s \longrightarrow^{\Diamond} P}$$

If Q denotes a set of *final* configurations, then the judgment $t/s \longrightarrow^{\Diamond} Q$ can be viewed as a particular case of the judgment $t/s \longrightarrow^{\Diamond} P$, where P denotes a set of configurations. We prove that $t/s \longrightarrow^{\Diamond} Q$ matches our CPS-big-step judgment $t/s \Downarrow Q$.

$$\textsc{eventually-iff-cps-big-step:} \qquad t/s \longrightarrow^{\Diamond} Q \quad\Longleftrightarrow\quad t/s \Downarrow Q$$

3.3 Chained Rule and Cut Rule for the “Eventually” Judgment

To apply the rule eventually-step, one needs to provide upfront an intermediate postcondition P'. Doing so is not always convenient. It turns out that we can leverage the CPS-small-step judgment $t/s \longrightarrow P'$ to provide an introduction rule for $t/s \longrightarrow^{\Diamond} P$ that does not require providing P' upfront. This rule, which we call the *chained* version of eventually-step, admits the statement shown below. It reads as follows: if every possible step of t/s reduces in one step to a configuration that eventually reaches a configuration from the set P, then every possible evaluation of t/s eventually reaches a configuration from the set P.

$$\textsc{eventually-step-chained}: \qquad t/s \longrightarrow \Big\{(t',s')\ \Big|\ t'/s' \longrightarrow^{\Diamond} P\Big\} \quad\Rightarrow\quad t/s \longrightarrow^{\Diamond} P$$

Another interesting property of the judgment $t/s \longrightarrow^{\Diamond} P$ is its cut rule, which is derivable. It asserts the following: if every possible evaluation of t/s eventually reaches a configuration in the set P', and if every configuration from the set P' eventually reaches a configuration from the set P, then every possible evaluation of t/s eventually reaches a configuration from the set P.

$$\textsc{eventually-cut}: \qquad t/s \longrightarrow^{\Diamond} P' \quad\wedge\quad \Big(\forall (t',s') \in P'.\ t'/s' \longrightarrow^{\Diamond} P\Big) \quad\Rightarrow\quad t/s \longrightarrow^{\Diamond} P$$

This cut rule also admits a *chained* version. The corresponding rule, shown below, reads as follows: if every possible evaluation of t/s *eventually* reaches a configuration that itself eventually reaches a configuration from the set P, then every possible evaluation of t/s eventually reaches a configuration from the set P.

$$\textsc{eventually-cut-chained}: \qquad t/s \longrightarrow^{\Diamond} \Big\{(t',s')\ \Big|\ t'/s' \longrightarrow^{\Diamond} P\Big\} \quad\Rightarrow\quad t/s \longrightarrow^{\Diamond} P$$

The cut rule and the chained rules are particularly handy to work with, as we illustrate in §6.6.

3.4 Coinductive Interpretation of the CPS-Small-Step Judgment

Let $t/s \longrightarrow^{\Diamond}_{\text{co}} P$ denote the coinductive interpretation of the two rules that define $t/s \longrightarrow^{\Diamond} P$. We can relate the coinductive judgment $t/s \longrightarrow^{\Diamond}_{\text{co}} P$ with the coinductive CPS-big-step judgment from §2.3. Here again, we view a set Q of final configurations as a set of configurations.

$$t/s \longrightarrow^{\Diamond}_{\text{co}} Q \quad\iff\quad t/s \Downarrow^{\text{co}} Q$$

Divergence can be captured by instantiating P as the empty set. We prove that the judgment $t/s \longrightarrow^{\Diamond} \emptyset$ is equivalent to the standard small-step characterization of divergence, which asserts that any execution prefix may be extended with at least one additional step.

$$t/s \longrightarrow^{\Diamond}_{\text{co}} \emptyset \qquad\iff\qquad \forall s't'.\ (t/s \longrightarrow^{*} t'/s') \Rightarrow \left(\exists t's'.\ t/s \longrightarrow t'/s'\right)$$

4 TYPE-SOUNDNESS PROOFS USING CPS SEMANTICS

In this section, we show how the CPS-small-step and the CPS-big-step judgments may be used to carry out type-soundness proofs. We illustrate the proof structures using simple types (STLC). As a warm-up, we begin with a presentation of type soundness on a restriction of our language to the state-free fragment of the language.

For this section, we need to consider a different semantics for the random-number generator. Indeed, the current rule CPS-BIG-RAND asserts that the program is stuck if rand n is invoked with an argument $n \le 0$. Since here we are interested in proving that well-typed programs do not get stuck, let us consider a modified semantics, where rand n is turned into a total function that returns 0 when $n \le 0$.

$$\textsc{cps-big-rand-complete}\quad \frac{\left(n > 0 \ \wedge\ \forall m.\ 0 \le m < n \ \Rightarrow\ (m, s) \in Q\right) \quad\vee\quad \left(n \le 0 \ \wedge\ (0, m) \in Q\right)}{(\mathsf{rand}\, n)/s \Downarrow Q}$$

Besides, for this section, we also exclude the primitive operation free, which is not type-safe.

The grammar of types, written T, appears below.

$$T \quad := \quad \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid T \to T \mid \mathsf{ref}\, T$$

A typing environement, written E, maps variable names to types. The judgment $\vdash v : T$ asserts that the closed value v admits the type T. The judgment $E \vdash t : T$ asserts that the term t admits type T in the environment E. We let $\mathbb{V}$ denote the set of terms that are either values or variables—recall that we consider A-normal forms to simplify the presentation. The typing rules (given in appendix D) are essentially standard, appart from the fact that they involve side conditions of the form $t \in \mathbb{V}$ to constrain terms to be in A-normal form.

4.1 CPS-Small-Step Type-Soundness Proof for a State-Free Language

A *stuck term* is a term that is not a value and that cannot take a step. Type soundness asserts that if a closed term t is well-typed, then none of its possible evaluations gets stuck. In other words, if t reduces in a number of steps to t', then t' is either a value or it can further reduce.

TYPE-SOUNDNESS (STATE-FREE LANGUAGE):

$$(\emptyset \vdash t : T) \ \wedge\ (t \longrightarrow^{*} t') \quad\Rightarrow\quad (\mathsf{isvalue}\, t') \ \vee\ (\exists t''.\ t' \longrightarrow t'')$$

The traditional approach to establishing type soundness is by proving the *preservation* and *progress* properties [Pierce 2002; Wright and Felleisen 1994].

PRESERVATION (STATE-FREE LANGUAGE): $\quad E \vdash t : T \quad\wedge\quad t \longrightarrow t' \quad\Rightarrow\quad E \vdash t' : T$

PROGRESS (STATE-FREE LANGUAGE): $\quad \emptyset \vdash t : T \quad\Rightarrow\quad (\mathsf{isvalue}\, t) \ \vee\ (\exists t'.\ t \longrightarrow t')$

Each of these proofs is most typically carried out by induction on the typing judgment. One difficulty that might arise in the type-preservation proof for a large language with dozens (if not hundreds) of typing rules is the fact that one needs, for each case of the typing judgment $E \vdash t : T$, to inspect all the potential cases of the reduction judgment $t \longrightarrow t'$. This inspection is not really quadratic in practice, because one can filter out applicable rules based on the shape of the term t. Nevertheless, a typical Coq proof performing "`intros HT HR; induction HT; inversion HR`" does produce a proof term whose size is quadratic in the number of term constructs. Coq users have experienced performance challenges with quadratic-complexity proof terms when formalizing PL metatheory [Monin and Shi 2013].

Interestingly, in the particular case of a deterministic language, there exists a known strategy (e.g., [Rompf and Amin 2016]) to reformulate the preservation and progress statements in a way that not only factors out the two into a single statement, but also can be proved with a linear proof term. This combined statement, shown below, asserts that a well-typed term t is either a value or it can make a step towards a term t' that admits the same type.

induction-for-type-soundness (state-free, standard small-step, deterministic case)

$$\emptyset \vdash t : T \quad \Rightarrow \quad \big(\mathsf{isvalue}\, t\big) \;\vee\; \big(\exists t'.\ (t \longrightarrow t') \wedge (\emptyset \vdash t' : T)\big)$$

As we explain next, this approach can be generalized to the case of nondeterministic languages using the CPS-small-step judgment. Let us write $t \longrightarrow P$ for the judgment that corresponds to $t/s \longrightarrow P$ without the state argument. We can state type soundness by considering for the postcondition P the set of terms t' that admit the same type as t.

induction-for-type-soundness (state-free, cps-small-step, general case)

$$\emptyset \vdash t : T \quad \Rightarrow \quad \big(\mathsf{isvalue}\, t\big) \;\vee\; \Big(t \longrightarrow \big\{t' \,\big|\, (\emptyset \vdash t' : T)\big\}\Big)$$

The proof is carried out by induction on the typing judgment. For the case where t is a value, the left part of the disjunction applies. For all other cases, the right part needs to be established. For each term construct, we invoke the relevant constructor from the definition of the judgment $t \longrightarrow P$. The application of such a rule typically results in proof obligations of the form $\emptyset \vdash t' : T$, where t' can be any of the terms to which t reduces. In case of a nondeterministic evaluation rule, these possible terms t' are quantified universally in the premise of the CPS-small-step rule, thus they become universally quantified in the proof obligation that arises for the user. In other words, we are required to establish $\emptyset \vdash t' : T$ for each possible t', as expected.

The statement induction-for-type-soundness above entails the preservation property (for empty environments) and the progress property. We prove once and for all that the statement of induction-for-type-soundness entails the type-soundness property.[3]

4.2 CPS-Small-Step Type-Soundness Proof for an Imperative Language

Let us now generalize the results from the previous section to account for memory operations.

A store-typing environment, written S, is a map from locations to types. The typing judgment for values is extended with a store-typing environment, taking the form $S \vdash v : T$. Likewise, the typing judgment for terms is extended to the form $S; E \vdash t : T$. The store-typing entity S only plays a role in the typing rule for memory locations. The rule for typing memory locations appears

[3]The generic entailment from induction-for-type-soundness to type-soundness holds for any type system and for any judgment $t \longrightarrow P$ related to the small-step judgment $t \longrightarrow t'$ in the expected way, that is, satisfying the property cps-small-step-iff-progress-and-correct from §3.2.

next, together with the rules for typing memory operations.

$$\textsc{vtyp-loc}\quad \frac{p \in \mathrm{dom}\, S \qquad S[p] = T}{S \vdash p : (\mathsf{ref}\, T)} \qquad\qquad \textsc{typ-ref}\quad \frac{S;E \vdash t_1 : T \qquad t_1 \in \mathbb{V}}{S;E \vdash (\mathsf{ref}\, t_1) : (\mathsf{ref}\, T)}$$

$$\textsc{typ-get}\quad \frac{S;E \vdash t_1 : (\mathsf{ref}\, T) \qquad t_1 \in \mathbb{V}}{S;E \vdash (\mathsf{get}\, t_1) : T} \qquad \textsc{typ-set}\quad \frac{S;E \vdash t_1 : (\mathsf{ref}\, T) \qquad S;E \vdash t_2 : T \qquad t_1, t_2 \in \mathbb{V}}{S;E \vdash (\mathsf{set}\, t_1\, t_2) : \mathsf{unit}}$$

The type-soundness property asserts that the execution of any well-typed term, starting from the empty state, does not get stuck. In the statement below, $\varnothing$ denotes an empty state or an empty store typing, whereas $\emptyset$ denotes, as before, the empty typing context.

TYPE-SOUNDNESS:

$$(\varnothing;\emptyset \vdash t : T) \ \wedge\ (t/s \longrightarrow^* t'/s') \quad\Rightarrow\quad (\mathsf{isvalue}\, t') \ \vee\ (\exists t''s''.\ t'/s' \longrightarrow t''/s'')$$

To establish a type-soundness result by induction on a reduction sequence, one needs to introduce a typing judgment for stores. A store s admits type S, written $\vdash s : S$, if and only if s and S have the same domain and, for any location p in the domain, $s[p]$ admits the type $S[p]$. Formally:

$$\vdash s : S \qquad\equiv\qquad \big(\mathsf{dom}\, s = \mathsf{dom}\, S\big) \ \wedge\ \big(\forall p \in \mathsf{dom}\, s.\ \vdash s[p] : S[p]\big)$$

The preservation and progress lemmas associated with the traditional approach to proving type soundness are updated as shown below. In particular, the preservation lemma requires the output state to admit a type that extends the store typing associated with the input state ($S' \supseteq S$).

$$\begin{array}{lll}\textsc{preservation:} & & S;\emptyset \vdash t : T \ \ \wedge\ \ \vdash s : S \ \ \wedge\ \ t/s \longrightarrow t'/s' \\ & \Rightarrow & \exists S' \supseteq S.\ \vdash s' : S' \ \ \wedge\ \ S';\emptyset \vdash t' : T \\ \textsc{progress:} & & S;\emptyset \vdash t : T \ \Rightarrow\ (\mathsf{isvalue}\, t) \ \vee\ (\exists t's'.\ t/s \longrightarrow t'/s')\end{array}$$

In constrast, using the CPS-small-step judgment, we can establish type soundness through a single induction on the typing judgment. To that end, we formulate a lemma in terms of the predicate $t/s \longrightarrow P$, instantiating the set P as the set of configurations t'/s' such that t' admits the same type as t, and such that s' admits a type that extends the type of s.

INDUCTION-FOR-TYPE-SOUNDNESS (CPS-SMALL-STEP)

$$\begin{array}{ll} & \big(S;\emptyset \vdash t : T\big) \ \wedge\ \big(\vdash s : S\big) \\ \Rightarrow & \big(\mathsf{isvalue}\, t\big) \ \vee\ \Big(t/s \longrightarrow \big\{(t', s') \,\big|\, \exists S' \supseteq S.\ (\vdash s' : S') \wedge (S';\emptyset \vdash t' : T)\big\}\Big)\end{array}$$

4.3 CPS-Big-Step Type-Soundness Proof for an Imperative Language

Traditionally, a big-step soundness proof can only be carried out if the semantics is completed using error-propagation rules. We here demonstrate how to establish type soundness with respect to a CPS-big-step judgment, without any need for error-propagation rules.

Consider a type T and a store-typing S. We define $[\![T/S]\!]$ as the set of final configurations of the form v/s such that the state s admits a type S' that extends S, and the value v admits type T, under the store typing S'. Formally:

$$[\![T/S]\!] \quad\equiv\quad \big\{(v, s) \mid \exists S' \supseteq S.\ (\vdash s : S') \wedge (S' \vdash v : T)\big\}$$

The coinductive CPS-big-step judgment $t/s \Downarrow^{\mathrm{co}} [\![T/S]\!]$ asserts that any evaluation of t/s executes safely, without ever getting stuck; and that if an evaluation reaches a final configuration v/s', then this configuration satisfies the postcondition $[\![T/S]\!]$. Given our definition of $[\![T/S]\!]$, the judgment

$t/\varnothing \Downarrow^{co} [\![T/\varnothing]\!]$ thus captures exactly the type-soundness property associated with the typing judgment $\varnothing;\emptyset \vdash t : T$.

Type soundness may be established by proving the following statement by coinduction.

$$\textsc{coinduction-for-type-soundness (cps-big-step)}:\\ S;\emptyset \vdash t : T \quad \wedge \quad \vdash s : S \quad \Rightarrow \quad t/s \Downarrow^{co} [\![T/S]\!]$$

The coinduction hypothesis asserts that we can invoke the result that we are trying to prove, provided that we have already made some progress by applying one of the introduction rules that define the coinductive judgment $t/s \Downarrow^{co} Q$. The first step of the proof is to perform a case analysis on the typing hypothesis $S;\emptyset \vdash t : T$. We then consider each of the possible typing rules one by one. For each typing rule, we exploit the constraints that apply to the term t to show that there exists a CPS-big-step evaluation rule that applies to this term. We apply this rule to the current proof obligation, of the form $t/s \Downarrow^{co} [\![T/S]\!]$. For each evaluation judgment that appears as premise of the evaluation rule, we can invoke the coinduction hypothesis.

Like for the small-step settings, we proved once and for all that, for any type system, the statement COINDUCTION-FOR-TYPE-SOUNDNESS entails TYPE-SOUNDNESS.

Our coinductive CPS-big-step approach offers, to those who have good reasons to work with a big-step-style semantics, a means to establish type soundness without introducing error rules.

Regarding the comparison with the standard preservation and progress approach, at this stage we cannot draw general conclusions on whether CPS-big-step and CPS-big-small-step type-soundness proofs are more effective, because we considered a relatively simple language. Nevertheless, it appears that each of the two approaches that we propose results in proof scripts that (1) require only one induction or one coinduction instead of two separate inductions, (2) are no longer than with preservation and progress separated, and (3) avoid the issue of nested inversions requiring a number of cases quadratic in the size of the language.

5 DEFINITION OF TRIPLES

5.1 Semantics of Hoare Triples for Nondeterministic Languages

A Hoare triple, written $\{H\}\ t\ \{Q\}$, describes the behavior of the evaluation of the configurations t/s for any s satisfying the precondition H, in terms of the postcondition Q. The exact interpretation of a triple depends on whether it accounts for *total correctness* or *partial correctness*, which differ on how they account for termination. For nondeterministic languages, the key notions of interest for definining a triple $\{H\}\ t\ \{Q\}$ are as follows.

- **Safety**: for any s satisfying H, none of the possible evaluations of t/s can get stuck.
- **Correctness**: for any s satisfying H, if t/s can evaluate to v/s', then $Q\,v\,s'$ holds.
- **Termination**: for any s satisfying H, all possible evaluations of t/s are finite.
- **Partial correctness**: safety and correctness hold.
- **Total correctness**: safety and correctness and termination hold.

5.2 Definition of Hoare Triples w.r.t. CPS-Big-Step Semantics

For a nondeterministic semantics, a total-correctness Hoare triple asserts that every possible execution of t/s terminates and satisfies the postcondition. This property can be captured directly by the CPS-big-step judgment:

$$^{\text{total, nondeterministic}}\{H\}\,t\,\{Q\} \quad \equiv \quad \forall s.\ H\,s \ \Rightarrow \ (t/s \Downarrow Q)$$

For a nondeterministic semantics, a partial-correctness Hoare triple asserts that every possible execution of t/s either diverges or terminates and satisfies the postcondition. This property can be captured directly by the coinductive CPS-big-step judgment:

$$^{\text{partial,nondeterministic}}\{H\}\,t\,\{Q\} \quad\equiv\quad \forall s.\ H\,s \ \Rightarrow\ (t/s \Downarrow^{\text{co}} Q)$$

Note that instantiating Q with the always-false predicate in the partial-correctness triple yields a characterization of programs whose execution always diverges—and never gets stuck.

5.3 Deriving Reasoning Rules for Triples

After defining a notion of Hoare triples, the usual next step is to state and prove the reasoning rules associated with these triples. These rules are proved correct with respect to the semantics. Consider for example the case of a let-binding. Let us compare the semantics rule CPS-BIG-LET with the Hoare-logic rule HOARE-LET.

Throughout this section, we formulate rules by viewing postconditions as predicates of type val → state → Prop, as this presentation style is more idiomatic in program logics. Also, we here present reasoning rules using the horizontal bar, but keep in mind that the statements are not inductive definitions: they correspond to lemmas proved correct with respect to the semantics.

$$\text{CPS-BIG-LET}\quad \frac{t_1/s \Downarrow Q_1 \qquad \big(\forall v's'.\ Q_1\,v'\,s' \ \Rightarrow\ ([v'/x]\,t_2)/s' \Downarrow Q\big)}{(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)/s \Downarrow Q} \qquad \text{HOARE-LET}\quad \frac{\{H\}\ t_1\ \{Q_1\} \qquad \big(\forall v'.\ \{Q_1\,v'\}\ ([v'/x]\,t_2)\ \{Q\}\big)}{\{H\}\ (\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\ \{Q\}}$$

The only difference between the two rules is that the first one considers one specific state s, whereas the second one considers a set of possible states satisfying the precondition H. By exploiting the fact that $\{H\}\ t\ \{Q\}$ is defined as $\forall s.\ H\,s \Rightarrow (t/s \Downarrow Q)$, it is straightforward to establish that HOARE-LET is a consequence of CPS-BIG-LET. The corresponding Coq proof script witnesses the simplicity of the proof: "`intros. applys mbig_let; eauto.`"

To give one more example, consider the consequence rule. The Hoare-logic rule is, again, an immediate consequence from the CPS-big-step rule.

$$\text{CPS-BIG-CONSEQUENCE}\quad \frac{t/s \Downarrow Q \qquad Q \subseteq Q'}{t/s \Downarrow Q'} \qquad \text{HOARE-CONSEQUENCE}\quad \frac{H' \subseteq H \qquad \{H\}\ t\ \{Q\} \qquad Q \subseteq Q'}{\{H'\}\ t\ \{Q'\}}$$

5.4 Deriving Weakest-Precondition-Style Reasoning Rules

The judgment $t/s \Downarrow Q$ provides an inductive definition of a weakest-precondition operator, up to the order of arguments. Thus, we can define the wp operator in terms of the semantics as $\lambda tQs.\,(t/s \Downarrow Q)$.

Reasoning rules in weakest-precondition style are even easier to derive than the rules for triples. Recall that $H \vdash H'$ denotes entailment, defined as pointwise predicate implication $(\forall s.\,H\,s \Rightarrow H\,s')$. Consider for example the semantics rule and the wp-reasoning rule associated with a let-binding.

$$\frac{t_1/s \Downarrow Q_1 \qquad \big(\forall v's'.\ Q_1\,v'\,s' \ \Rightarrow\ ([v'/x]\,t_2)/s' \Downarrow Q\big)}{(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)/s \Downarrow Q}\ \text{CPS-BIG-LET}$$

$$\frac{}{\mathsf{wp}\,t_1\,(\lambda v_1.\,\mathsf{wp}\,([v_1/x]\,t_2)\,Q) \ \vdash\ \mathsf{wp}\,(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\,Q}\ \text{WP-LET}$$

To derive the rule WP-LET from CPS-BIG-LET, it suffices to instantiate Q_1 as $\lambda v_1.\,\mathsf{wp}\,([v_1/x]\,t_2)\,Q$.

In summary, by considering a semantics expressed in CPS-big-step style, one can derive practical reasoning rules, both in Hoare triple style and in weakest-precondition style, via one-line proofs. The construction of a program logic on top of a CPS-big-step is thus a significant improvement, both over the use of a standard big-step semantics, which falls short in the presence of nondeterminism, and over the use of a small-step semantics, which requires much more work for deriving the reasoning rules, especially if one aims for total correctness.

6 COMPILER-CORRECTESS PROOFS FOR TERMINATING PROGRAMS

6.1 Motivation: Avoiding Both Backward Simulations and Artificial Determinism

Following CompCert's terminology [Leroy 2009], one particular evaluation of a program can admit one out of four possible behaviors: *terminate* (with a value, exception, etc.), trigger *undefined behavior*, *diverge silently* after performing a finite number of I/O operations, or be *reactive* by performing an infinite sequence of I/O operations. A general-purpose compiler ought to preserve behaviors, except that undefined behaviors can be replaced with anything.

In this paper, we focus on proofs of compiler correctness for programs that always terminate safely. Such a result is sufficient for many practical applications in software verification where source programs are proven to be safe, and often, the only use case for nontermination is a top-level infinite event-handling loop, which can be implemented outside of the compiler [Erbsen et al. 2021]. We leave to future work the application of CPS semantics to the correct compilation of programs that diverge, react, or trigger undefined behavior on some inputs but not others.

In the particular case of a deterministic programming language, compiler correctness for terminating programs can be established via a *forward-simulation* proof. Such a proof consists of showing that each step from the source program *corresponds to* a number of steps in the compiled program. The correspondance involved is captured by a relation between source states and target states. Such forward-simulation proofs work well in practice. The main problem is that they do not generalize to nondeterministic languages.

Indeed, in the presence of nondeterminism, a source program may have several possible executions. As we restrict ourselves to the case of terminating programs, let us assume that all executions of the source program terminate, only possibly with different results. In that setting, a compiler is correct if (1) the compiled program always terminates, and (2) for any result that the compiled program may produce, the souce program could have produced that result. It may not be intuitive at first, but the inclusion is indeed *backwards*: the set of behaviors of the target program must be included in the set of behaviors of the source program.

To establish the backward behavior inclusion, one may set up a *backward simulation* proof. Such a proof consists of showing that each step from the target program corresponds to one or more steps in the source program.[4] Yet, backward simulations are much more unwieldy to set up than forward simulations. Indeed, in most cases one source program step is implemented by multiple steps in the compiled program, thus a backward simulation relation typically needs to relate many more pairs than a forward simulation relation.

This observation motivated the CompCert project [Leroy 2009] to exploit forward simulations as much as possible, at the cost of making the semantics of the intermediate language deterministic even when it is not natural to do so, and even when implementing determinacy requires introducing artificial functions for, e.g., computing fresh memory locations in a deterministic manner.

In this section:

- We explain how CPS semantics allow to avoid backward simulations, by carrying out forward-simulation proofs of compiler correctness, for nondeterministic terminating programs.

[4] The number of corresponding steps in the source program should not be zero, otherwise the target program could diverge whereas the source program terminates. In practice, however, it is not always easy to find one source-program step that corresponds to a target-program step. In such situations, one may consider a generalized version of backward simulations that allow for zero source program steps, provided that some well-founded measure decreases [Leroy 2009].

- We show how our the idea generalizes to languages including I/O operations and to the case where the source language and the target language are different.
- We present two case studies: one transformation that increases the amount of nondeterminism and one that decreases the amount of nondeterminism.
- We comment on the fact that our second case study features a CPS-big-step semantics for the source language, whereas it features a CPS-small-step semantics for the target language.

6.2 Establishing Correctness via Forward Simulations using CPS Semantics

Consider a compilation function written $C(t)$. For simplicity, we assume that the source and the target language are identical, we assume that the compilation does not alter the result values, and we assume the language to be state-free—we will generalize the results in the next subsection. In this subsection, $t \Downarrow v$ denotes the standard big-step judgment, $t \Downarrow Q$ denotes the CPS-big-step judgment, and terminates(t) asserts that all executions of t terminate safely, without undefined behavior. The compiler-correctness result for terminating programs captures preservation of termination and backward inclusion for results—points (1) and (2) stated earlier.

CORRECTNESS-FOR-TERMINATING-PROGRAMS:

$$\mathsf{terminates}(t) \quad\Rightarrow\quad \mathsf{terminates}(C(t)) \;\wedge\; \Big(\forall v.\;\; C(t) \Downarrow v \quad\Rightarrow\quad t \Downarrow v\Big)$$

We claim that this correctness result can be derived from the following statement, which describes forward preservation of specifications.

$$\textsc{cps-forward-preservation:}\qquad \forall Q. \qquad t \Downarrow Q \quad\Rightarrow\quad C(t) \Downarrow Q$$

Let us demonstrate the claim. Let us assume that terminates(t) hold. First of all, recall from §7.6 the equivalence named CPS-BIG-STEP-IFF-TERMINATES-AND-CORRECT that relates the CPS-big-step judgment and the termination judgment.

$$t \Downarrow Q \quad\Longleftrightarrow\quad \mathsf{terminates}(t) \;\wedge\; \Big(\forall v.\, (t \Downarrow v) \Rightarrow v \in Q\Big)$$

Exploiting this equivalence, the CPS-FORWARD-PRESERVATION assumption reformulates as follows.

$$\begin{aligned}\forall Q. \qquad & \Big(\mathsf{terminates}(t) \;\wedge\; \big(\forall v.\, (t \Downarrow v) \Rightarrow v \in Q\big)\Big) \\ \Rightarrow\; & \Big(\mathsf{terminates}(C(t)) \;\wedge\; \big(\forall v.\, (C(t) \Downarrow v) \Rightarrow v \in Q\big)\Big)\end{aligned}$$

The hypothesis terminates(t) holds by assumption. Let us instantiate Q as the strongest post-condition for t, that is, as the set $\{v \mid (t \Downarrow v)\}$. We obtain:

$$\Big(\forall v.\, (t \Downarrow v) \Rightarrow (t \Downarrow v)\Big) \quad\Rightarrow\quad \mathsf{terminates}(C(t)) \;\wedge\; \Big(\forall v.\, (C(t) \Downarrow v) \Rightarrow (t \Downarrow v)\Big).$$

The premise is a tautology, and the conclusion proves CORRECTNESS-FOR-TERMINATING-PROGRAMS.

6.3 Generalization to Handle I/O and Cross-Language Compilation

More generally, the behavior of a terminating program consists of the final result and its interactions with the outside world (input and output). These interactions include, e.g., interaction with the standard input and output streams, system calls, etc. Each interaction is called an *event*, and the semantics judgment is extended to collect such events into a *trace* τ. Figure 4 shows three illustrative cases of how the rules from Figure 2 are augmented with traces, making the choice to treat rand calls as observable events while reference-allocation nondeterminism remains internal.

Requiring a compiler to preserve only the nondeterministic choices recorded in the trace enables us to pick and choose which (external) interactions must be preserved by compilations and which (internal) nondeterministic choices the compiler may resolve as it sees fit. As a particularly fine-grained example, the trace might record that `malloc` was called and succeeded, but omit

$$\textsc{cps-big-let-trace}\quad \frac{t_1/s/\tau \Downarrow Q_1 \qquad \big(\forall (v', s', \tau') \in Q_1.\;\; ([v'/x]\, t_2)/s'/\tau' \Downarrow Q\big)}{(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)/s/\tau \Downarrow Q}$$

$$\textsc{cps-big-rand-trace}\quad \frac{n > 0 \qquad \big(\forall m.\;\; 0 \le m < n \;\Rightarrow\; (m, s, (n, m) :: \tau) \in Q\big)}{(\mathsf{rand}\, n)/s/\tau \Downarrow Q} \qquad \textsc{cps-big-ref}\quad \frac{\forall p \notin \mathrm{dom}\, s.\;\; (p,\; s[p := v], \tau) \in Q}{(\mathsf{ref}\, v)/s/\tau \Downarrow Q}$$

Fig. 4. CPS-big-step semantics with traces, selected rules

the pointer it returned to allow for optimizations that reduce the amount of allocation. To our knowledge, this level of flexibility is unique to CPS semantics. For a forward-simulation-based compiler correctness proof, constructing a deterministic model of all internal nondeterminism can be arbitrarily complicated (the CompCert memory model is an example).

We restrict our attention to semantics that only accept terminating commands c that do not go wrong and do not return a value for the rest of this section. For languages of terms (that return values) rather than commands (that don't return values), we would need a representation relation between source-level and target-level values—we omit this here for brevity, but §6.4 tackles a similar challenge. In the current setting, *behavior inclusion* holds between a source language program and a target language program if all traces that the target language program can produce (according to traditional small-step or big-step semantics) can also be produced by the source language program. More formally, we define the traces that can be produced from a starting configuration $c/s/\tau$ as

$$\mathrm{traces}(c, s, \tau) := \{\tau' \mid \exists s'.\; c/s/\tau \Downarrow s'/\tau'\}$$

and say that a compiler $C()$ satisfies behavior inclusion for a command starting from the initial source-level state s_{src} related to the initial target-level state s_{tgt} and initial trace τ if

$$\mathrm{traces}(C(c), s_{tgt}, \tau) \subseteq \mathrm{traces}(c, s_{src}, \tau)$$

Assuming CPS-big-step semantics $\Downarrow_{src}$ and $\Downarrow_{tgt}$ for the source and target language, plus a relation R between source- and target-languages states, we define *CPS simulation* as follows:

$$\forall s_{src}\; s_{tgt}\; \tau\; Q.\quad R(s_{src}, s_{tgt}) \wedge c/s_{src}/\tau \Downarrow_{src} Q \quad\Longrightarrow\quad C(c)/s_{tgt}/\tau \Downarrow_{tgt} Q_R$$
$$\text{where } Q_R(s'_{tgt}, \tau') := \exists s'_{src}.\; R(s'_{src}, s'_{tgt}) \wedge Q(s'_{src}, \tau')$$

Our goal is to prove that a CPS simulation implies trace inclusion if the source program is safe. We rely on two properties: First, soundness of CPS-big-step semantics with respect to traditional big-step semantics:

$$\forall c\; s\; s'\; \tau\; \tau'\; Q.\quad c/s/\tau \Downarrow s'/\tau' \;\wedge\; c/s \Downarrow Q \quad\Longrightarrow\quad Q(s', \tau') \tag{1}$$

And conversely, that a program which terminates safely and whose traditional big-step executions all satisfy a postcondition also has a CPS-semantics derivation:

$$\forall c\; s\; \tau\; Q.\; \mathrm{terminates}(c, s, \tau) \wedge (\forall s'\; \tau'.\; c/s/\tau \Downarrow s'/\tau' \Longrightarrow Q(s', \tau')) \quad\Longrightarrow\quad c/s/\tau \Downarrow Q \tag{2}$$

To show trace inclusion, i.e. $\mathrm{traces}(C(c), s_{tgt}, \tau) \subseteq \mathrm{traces}(c, s_{src}, \tau)$, we can assume a target language execution $C(c)/s_{tgt}/\tau \Downarrow s'_{tgt}/\tau'$ producing trace τ', and need to show $\tau' \in \mathrm{traces}(c, s_{src}, \tau)$. By applying (2) to the source program (whose termination we assume) and setting $Q(s'_{src}, \tau') := c/s_{src}/\tau \Downarrow s'_{src}/\tau'$ so that the second premise becomes a tautology, we obtain the source-level CPS semantics derivation $c/s_{src}/\tau \Downarrow (\lambda s'_{src}\; \tau'.\; c/s_{src}/\tau \Downarrow s'_{src}/\tau')$. Passing this into the CPS simulation yields $C(c)/s_{tgt}/\tau \Downarrow (\lambda s'_{tgt}\; \tau'. \exists s'_{src}. R(s'_{src}, s'_{tgt}) \wedge c/s_{src}/\tau \Downarrow s'_{src}/\tau')$. Now we can apply (1) to this

EVAL-UNOP
$$\frac{(y, v_y) \in \ell \qquad \text{evalunop}(op, v_y, v) \qquad Q(m, \ell[x := v], \tau)}{(x = op(y))/m/\ell/\tau \Downarrow Q}$$

EVAL-INPUT
$$\frac{\forall n, Q(m, \ell[x := n], \tau :: \mathsf{IN}\ n)}{(x = \text{input}())/m/\ell/\tau \Downarrow Q}$$

EVAL-SEQ
$$\frac{\begin{array}{c} c_1/m/\ell/\tau \Downarrow Q_1 \\ (\forall m'\ \ell'\ \tau',\ Q_1(m', \ell', \tau') \implies c_2/m'/\ell'/\tau' \Downarrow Q) \end{array}}{(c_1; c_2)/m/\ell/\tau \Downarrow Q}$$

EVAL-STORE
$$\frac{\begin{array}{c} (x, a) \in \ell \qquad (a + n) \in \text{dom}\ m \\ (y, v) \in \ell \qquad Q(m[(a + n) := v], \ell, \tau) \end{array}}{(x[n] = y)/m/\ell/\tau \Downarrow Q}$$

EVAL-ALLOC
$$\frac{\begin{array}{c} (\forall a\ \bar{v},\ \text{len}(\bar{v}) = n\ \wedge\ a, \ldots, (a + n - 1) \notin \text{dom}\ m \\ \implies Q(m[(a, \ldots (a + n - 1)) := \bar{v}], \ell, \tau)) \end{array}}{(x = \text{alloc}(n))/m/\ell/\tau \Downarrow Q}$$

EVAL-WHILE-AGAIN
$$\frac{\begin{array}{c} (x, 1) \in \ell \qquad c/m/\ell/\tau \Downarrow Q_1 \\ (\forall s',\ Q_1\ s' \implies (\text{while}\ x\ \text{do}\ c)/s' \Downarrow Q) \end{array}}{(\text{while}\ x\ \text{do}\ c)/m/\ell/\tau \Downarrow Q}$$

Fig. 5. Nondeterministic total CPS-big-step semantics (selected rules)

fact and the originally assumed target-level execution and obtain an s'_{src} such that $R(s'_{src}, s'_{tgt})$ and $c/s_{src}/\tau \Downarrow s'_{src}/\tau'$. This is, by definition, exactly what needs to hold to have $\tau' \in \text{traces}(c, s_{src}, \tau)$.

6.4 Case Study: Compiling Immutable Pairs to Heap-Allocated Records

This section describes a simple compiler pass that *increases* the amount of nondeterminism. The source language assumes a primitive notion of tuples, whereas the target language encodes such tuples by means of heap allocation. This case study is formalized with respect to a language based on commands whose arguments all must be variables. Such a language could be an intermediate language in a compiler pipeline, reached after an expression-flattening phase.

Language syntax. We let c denote a command, x, y, and z denote identifiers, and n denote unbounded natural-number constants. The grammar of the language is as follows.

$$\begin{array}{rcl} c & := & x = unop(y) \mid x = binop(y, z) \mid x = \text{input}() \mid \text{output}(x) \mid x = y[n] \mid x[n] = y \mid \\ & & x = \text{alloc}(n) \mid x = n \mid x = y \mid c_1; c_2 \mid \text{if}\ x\ \text{then}\ c_1\ \text{else}\ c_2 \mid \text{while}\ x\ \text{do}\ c \mid \text{skip} \end{array}$$

We actually consider two variants of this language, differing only in the type of values and in the available unary operators *unop* and binary operators *binop*.

The source language features an inductively defined type of values that can be natural numbers n or immutable pairs of values (i.e., the grammar of values is $v := n \mid (v, v)$). It includes as unary operators the projection function fst and snd (defined only on pairs) and the Boolean negation not (defined only on $\{0, 1\}$). Its binary operators are addition (+) and pair creation mkpair.

The target language admits only natural numbers as values. It includes only the negation and the addition operators.

CPS-big-step semantics. For both languages, the CPS-big-step evaluation judgment takes the form $c/m/\ell/\tau \Downarrow Q$, where c is a command, m is a memory state (a partial map from natural numbers to natural numbers), ℓ is an environment of local variables (a partial map from identifiers to values, whose type differs between the source and target languages as described above), τ denotes the I/O trace made of the events already performed *before* executing c, and the postcondition Q is a predicate over triples of the form (m', ℓ', τ'). A trace consists of a list of I/O events e whose grammar is $e := \mathsf{IN}\ n \mid \mathsf{OUT}\ n$.

The rules defining the judgment appear in Figure 5. They are common to both languages—only the set of supported unary and binary operators differs. In practice, we found it convenient also to derive a chained version of the CPS-big-step rule EVAL-SEQ, just like we did for CPS-small-step rules in §3.2.

$$\text{EVAL-SEQ-CHAINED}: \qquad c_1/m/\ell/\tau \Downarrow \left(\lambda m'\ell'\tau'.\ (c_2/m'/\ell'/\tau' \Downarrow P)\right) \quad \Rightarrow \quad (c_1; c_2)/m/\ell/\tau \Downarrow P$$

The semantics of operators are defined by two straightforward auxiliary relations (spelled out in Appendix E), $\mathsf{evalunop}(\mathit{unop}, v_1, v_2)$ asserting that applying *unop* to value v_1 results in v_2, and $\mathsf{evalbinop}(\mathit{binop}, v_1, v_2, v_3)$ asserting that applying *binop* to v_1 and v_2 results in v_3.

The command $x = \mathsf{input}()$ reads a natural number n, stores it into local variable x, and adds the event $(\mathsf{IN}\ n)$ to the event trace. The number n is chosen nondeterministically but recorded in the trace, so this results in *external* nonderterminism. The language has a built-in memory allocator but, for simplicity, we do not deal with deallocation. The command $x = \mathsf{alloc}(n)$ nondeterministically picks an address (natural number) a such that a, as well as the $n - 1$ addresses following a, are not yet part of the memory, initializes these addresses with nondeterministically chosen values, and returns a, resulting in *internal* nondeterminism, because this action is not recorded in the event trace. The *load* command $x = y[n]$ requires that the local variable y contains a natural number a and stores the value of the memory at address $a + n$ into variable x (and is undefined if $a + n$ is not mapped by the memory). The *store* command $x[n] = y$ stores the natural number contained in the local variable y at memory location $a + n$, where a is the address contained in local variable x, but only if memory at address $a + n$ has already been allocated.

Compilation function. The compilation function C lays out the pairs of the source language on the heap memory of the target language. This function is defined recursively on the source program. It maps the source-language operators that are not supported by the target language as follows.

$$\begin{array}{lcl} C(x = \mathsf{fst}(y)) & := & x = y[0] \\ C(x = \mathsf{snd}(y)) & := & x = y[1] \\ C(x = \mathsf{mkpair}(y, z)) & := & \mathsf{tmp} = \mathsf{alloc}(2);\ \mathsf{tmp}[0] = y;\ \mathsf{tmp}[1] = z;\ x = \mathsf{tmp} \end{array}$$

Note that to compile mkpair, we cannot simply store the address returned by alloc directly into x, because if x is the same variable name as y or z, then we would be overwriting the argument. For this reason, we use a temporary variable tmp that we declare to be reserved for compiler usage.

Simulation relation. To carry out the proof of correctness of the function $C(c)$, we introduce a simulation relation R for relating a source-language state (m_1, ℓ_1) with a target-language state (m_2, ℓ_2). To that end, we first define the relation $\mathsf{valuerepr}(v, w, m)$, to relate a source-language value v with the corresponding target-language value w, in a target-language memory m. This relation is implemented as the recursive function shown below—it could equally well consist of an inductive definition. A pair (v_1, v_2) is represented by address w if recursively v_1 is represented by the value at address w and v_2 is represented by the value at address $w + 1$. A natural number n has the same representation on the target-language level, i.e. we just assert $w = n$.

$$\begin{array}{lcl} \mathsf{valuerepr}((v_1, v_2), w, m) & := & (\exists w_1, (w, w_1) \in m\ \wedge\ \mathsf{valuerepr}(v_1, w_1, m)) \wedge \\ & & (\exists w_2, (w + 1, w_2) \in m \wedge \mathsf{valuerepr}(v_2, w_2, m)) \\ \mathsf{valuerepr}(n, w, m) & := & w = n \end{array}$$

The relationship R between source and target states can then be defined using valuerepr. In the definition shown below, we write $m_2 \supseteq m_1$ to mean that memory m_2 extends m_1, and write $m_2 \setminus m_1$ to denote the map-subtraction operator that restricts m_2 to contain only addresses not bound in

m_1. The locations bound by m_2 but not by m_1 correspond to the memory addresses of the pairs allocated on the heap in the target language.

$$\begin{aligned} R((m_1,\ell_1),(m_2,\ell_2)) \quad := \quad & \mathsf{tmp} \notin \mathsf{dom}\ \ell_1 \wedge m_2 \supseteq m_1 \wedge \\ & \forall (x,v) \in \ell_1.\ \exists w.\ (x,w) \in \ell_2 \wedge \mathsf{valuerepr}(v,w,m_2 \setminus m_1) \end{aligned}$$

Correctness proof. We are now ready to tackle the CPS-forward-simulation proof.

Theorem 6.1 (cps simulation for the pair-heapification compiler). $\forall c\ m_{src}\ \ell_{src}\ m_{tgt}\ \ell_{tgt}\ \tau\ Q.$

$$\begin{aligned} &\mathsf{tmp} \notin \mathsf{vars}(c)\ \wedge\ R((m_{src},\ell_{src}),(m_{tgt},\ell_{tgt}))\ \wedge\ c/m_{src}/\ell_{src}/\tau \Downarrow_{src} Q \quad \Longrightarrow \quad C(c)/m_{tgt}/\ell_{tgt}/\tau \Downarrow_{tgt} Q_R \\ &\quad \text{where} \quad Q_R(m'_{tgt},\ \ell'_{tgt},\ \tau') := \exists m'_{src}\ \ell'_{src}\ .\ R((m'_{src},\ \ell'_{src}),(m'_{tgt},\ \ell'_{tgt})) \wedge Q(m'_{src},\ \ell'_{src},\ \tau') \end{aligned}$$

Proof. By induction on the derivation of $c/m_{src}/\ell_{src}/\tau \Downarrow Q$. In each case, the goal to prove is initially of the form $C(c)/m_{tgt}/\ell_{tgt}/\tau \Downarrow Q_R$, where c has some structure that allows us to simplify $C(c)$ into a more concrete program snippet. We consider the resulting simplified goal as an invocation of a weakest-precondition generator on that program snippet, and we view the rules of Figure 5 as weakest-precondition rules that we can apply in order to step through the program snippet, using the hypotheses obtained from inverting the source-level derivation $c/m_{src}/\ell_{tgt}/\tau \Downarrow Q$ to discharge the side conditions that arise. Whenever we encounter a sequence of commands, we use eval-seq-chained instead of eval-seq, so that we do not have to provide an intermediate postcondition. In the cases where commands have subcommands, we use the inductive hypotheses about their execution as if they were previously proven lemmas about these “functions”.

We only present the case where $c = (x = \mathsf{mkpair}(y,z))$ in more detail: We have to prove a goal of the form $C(x = \mathsf{mkpair}(y,z))/m_{tgt}/\ell_{tgt}/\tau \Downarrow Q_R$, which simplifies to

$$(\mathsf{tmp} = \mathsf{alloc}(2); \mathsf{tmp}[0] = y; \mathsf{tmp}[1] = z; x = \mathsf{tmp})/m_{tgt}/\ell_{tgt}/\tau \Downarrow Q_R$$

Applying eval-seq-chained turns it into

$$(\mathsf{tmp} = \mathsf{alloc}(2))/m_{tgt}/\ell_{tgt}/\tau \Downarrow \left(\lambda m'_{tgt}\ \ell'_{tgt}\ \tau'.\ (\mathsf{tmp}[0] = y; \mathsf{tmp}[1] = z; x = \mathsf{tmp})/m'_{tgt}/\ell'_{tgt}/\tau' \Downarrow Q_R\right)$$

Applying eval-alloc turns it into

$$\begin{gathered} \forall\ a\ \overline{v}.\mathsf{len}(\overline{v}) = 2 \Longrightarrow a, a+1 \notin \mathsf{dom}\ m_{tgt} \Longrightarrow \\ (\mathsf{tmp}[0] = y; \mathsf{tmp}[1] = z; x = \mathsf{tmp})/m_{tgt}[a..(a+1) := \overline{v}]/\ell_{tgt}[x := a]/\tau \Downarrow Q_R \end{gathered}$$

Note how the fact that the address a and the list of initial values $\overline{v}$ are chosen nondeterministically naturally shows up as a universal quantification, and note how the memory and locals appearing in the state to the left of the $\Downarrow$ have been updated by the alloc function. After introducing these universally quantified variables and the hypotheses, we again have a goal of the form “$\ldots \Downarrow \ldots$” and continue by applying eval-seq-chained, eval-store, eval-seq-chained, eval-store, eval-set. Finally, we prove Q_R for the locals and memory updated according to the various eval-… rules that we applied by using map laws and the initial hypothesis $R((m_{src},\ell_{src}),(m_{tgt},\ell_{tgt}))$. □

6.5 Case Study: Introduction of Stack Allocation

This second case study illustrates the case of a transformation that reduces the amount of nondeterminism. The transformation consists of adding a *stack-allocation* feature to the compiler developed by Erbsen et al. [2021]. Proving this transformation correct using a CPS-big-step forward simulation was straightforward and took us only a few days of work—most of the work was *not* concerned with dealing with nondeterminism. This is in stark contrast to the outlook of using traditional evaluation judgments: verifying the same transformation would have required either more complex invariants to set up a backward simulation, or completely rewriting the memory model so that pointers are represented by deterministically-generated unobservable identifiers to allow for a compiler correctness proof by forward simulation. In fact, addressable stack allocation was initially omitted from the language exactly to avoid these intricacies (based on the experience from CompCert), but switching to CPS semantics made its addition a local and uncomplicated.

The input language is an imperative command language similar to the one described in §6.4. The memory is modeled as a partial map from machine words (32-bit or 64-bit integers) to bytes. The stack-allocation feature here consists of a command let x = stackalloc(n) in c made available to the source language. This command assigns an address to variable x at which n bytes of memory will be available during the execution of command c. Our compiler extension implements this command by allocating the requested n bytes on the stack, computing the address at runtime based on the stack pointer.

The key challenge is that the source-language semantics does not feature a stack. The stack gets introduced further down the compilation chain. Thus, the simplest way to assign semantics to the stackalloc function in the source language is to pretend that it allocates memory at a *nondeterministically chosen* memory location. This nondeterministic choice is described using a universal quantification in the CPS-big-step rule shown below, like in rule CPS-BIG-REF from §2.

$$\frac{\begin{array}{c}\forall m_{new}\ a.\ \text{dom}\ m_{new} \cap \text{dom}\ m = \emptyset \ \wedge\ \text{dom}\ m_{new} = \{a' \mid a \leq a' < a + n\} \quad \Longrightarrow \\ c/m \cup m_{new}/\ell[x := a]/\tau \Downarrow \lambda m'\ \ell'\ \tau'.\ P(m' \setminus m_{new},\ \ell',\ \tau')\end{array}}{(\text{let } x = \text{stackalloc}(n) \text{ in } c)/m/\ell/\tau \Downarrow P}$$

In the source language, the address returned by stackalloc is picked nondeterministically, whereas in the target language the address used for the allocation is deterministically computed, as the current stack pointer augmented with some offset. Thus, the compiler phase that compiles away the stackalloc command *reduces* the amount of nondeterminism.

The compiler correctness proof proceeds by induction on the CPS-semantics derivation for the source language, producing a target-language execution with a related postcondition. The simulation relation R describes the target-language memory as a disjoint union of unallocated stack memory and the source-language memory state. Critically, the case for stackalloc has access to a universally quantified induction hypothesis (derived from the rule shown above) about *target-level executions of* $C(c)$ *for any address a.*

As the address of the stack-allocated memory is not recorded in the trace, we are free to instantiate it with the specific stack-space address, expressed in terms of compile-time stack layout parameters and the runtime stack pointer. Reestablishing the simulation relation to satisfy the precondition of that induction hypothesis then involves carving out the freshly allocated memory from unused stack space and considering it a part of the source-level memory instead, matching the compiler-chosen memory layout and the preconditions of the stackalloc CPS-semantics rule. It is this last part that made up the vast majority of the verification work in this case study; handling the nondeterminism itself is as straightforward as it gets.

Note that it would not be possible to complete the proof by instantiating a with a compiler-chosen offset from the stack pointer if the semantics recorded the value of a in the trace. The (unremarkable) proof for the input command in the previous section also has access to a universally quantified execution hypothesis, but it *must* directly instantiate its universally quantified induction hypothesis with the variable introduced when applying the target-level CPS-semantics input rule to the goal to match the target-language trace to the source-language trace. Either way, reasoning about the reduction of nondeterministim in a CPS-forward-preservation proof boils down to instantiating a universal quantifier.

6.6 Compilation from a Language in CPS-Big-Step to One in CPS-Small-Step

If the semantics of the source language of a compiler phase are most naturally expressed in CPS-big-step, but the target language's semantics are best expressed in CPS-small-step semantics, it is convenient to prove a CPS forward simulation directly from a big-step source execution to a small-step target execution. For instance, the compiler in the project by Erbsen et al. [2021] includes such a translation, relating a big-step intermediate language to a small-step assembly language. In fact, this translation happens in the same case study described in the previous subsection. In what follows, we attempt to give a flavor of the proof obligations that arise from switching from CPS-big-step to CPS-small-step during the correctness proof.

Consider one sample CPS-small-step rule, for the load-word instruction `lw` that loads the value at the address stored in register r_2 and assigns it to register r_1:

$$\textsc{asm-lw}\quad \frac{(pc, \texttt{lw}\, r_1\, r_2) \in m \qquad (r_2, a) \in rf \qquad (a, v) \in m \qquad P(m, rf[r_1 := v], pc + 1, \tau)}{m/rf/pc/\tau \longrightarrow P}$$

Here, we model a machine state s_{tgt} as a quadruple of a memory m (that contains both instructions and data), a register file rf mapping register names to machine words a program counter pc, and a trace τ.

One can prove a CPS forward simulation from big-step source semantics directly to small-step target semantics:

$$\forall s_{src}\, s_{tgt}\, P.\ R(s_{src}, s_{tgt}) \ \wedge\ s_{src} \Downarrow P \quad \Longrightarrow \quad s_{tgt} \longrightarrow^{\Diamond} (\lambda s'_{tgt}.\exists s'_{src}.\ R(s'_{src}, s'_{tgt}) \wedge P(s'_{src}))$$

where R asserts, among other conditions, that the memory of the target state s_{tgt} contains the compiled program.

Like the proof described in §6.4, this proof also works by stepping through the target-language program by applying target-language rules and discharging their side conditions using the hypotheses obtained by inverting the source-language execution, with the only difference that instead of using the derived big-step rule EVAL-SEQ-CHAINED for chaining, one now uses the following two rules: EVENTUALLY-STEP-CHAINED and EVENTUALLY-CUT.

Applying EVENTUALLY-STEP-CHAINED turns the goal into a CPS-single-small-step goal with a given postcondition, which is suitable to apply rules with universally quantified postconditions like ASM-LW. Applying EVENTUALLY-CUT, on the other hand, creates two subgoals containing an uninstantiated unification variable for the intermediate postcondition. The unification variable appears as the postcondition in the first subgoal, so an induction hypothesis with the concrete postcondition from the theorem statement can be applied. In the second subgoal, this postcondition becomes the precondition for the remainder of the execution.

7 DISCUSSION: SEVEN INTERPRETATIONS OF CPS SEMANTICS

Trying to solve different, seemingly unrelated problems, the authors of this paper independently discovered CPS semantics several times. This section describes seven ways of discovering CPS semantics, resulting in different ways to think about the same concept, and should provide the reader with good intuition about what this judgment means and why it makes sense to consider it.

7.1 Generalizing from one result to a set of results

The standard big-step judgment $t/s \Downarrow v/s'$ relates one input configuration t/s to one single result configuration v/s'. The CPS-big-step judgment $t/s \Downarrow Q$ relates one input configuration t/s to a set of results, described by Q. The CPS-big-step judgment thus appears as the immediate generalization of the standard big-step judgment to go from one result to a set of results. The following results formally relate the CPS-big-step judgment to the standard big-step judgment.

If $t/s \Downarrow Q$ holds, then there exists at least one evaluation according to the standard big-step judgment whose final configuration satisfies Q:

$$\textsc{cps-big-to-one-big:} \qquad t/s \Downarrow Q \quad \Rightarrow \quad \exists vs'.\ t/s \Downarrow v/s' \ \wedge\ (v, s') \in Q$$

Moreover, if the two judgments hold simultaneously, then v/s' must belong to Q:

$$\textsc{cps-big-and-big-inv:} \qquad t/s \Downarrow Q \quad \wedge \quad t/s \Downarrow v/s' \quad \Rightarrow \quad (v, s') \in Q$$

As a corollory of the two previous properties, we get that if the CPS-big-step judgment $t/s \Downarrow Q$ holds for a singleton set Q made of only one configuration v/s', then the big-step judgment $t/s \Downarrow v/s'$ holds:

$$\textsc{cps-big-singleton:} \qquad t/s \Downarrow \{(v, s')\} \quad \Rightarrow \quad t/s \Downarrow v/s'$$

Note that, for deterministic semantics, the reciprocal to cps-big-singleton also holds.

Besides, one may wonder: why consider an overapproximation rather than the precise set of results? An attempt at inductively defining a judgment with a precise result set fails because it leads to a recursive invocation of the judgment in a "non-strictly-positive position"—allowing such definitions is unsound in general and therefore disallowed by Coq. It does not matter, though, because (1) working with an overapproximation is perfectly fine for many applications—types and postconditions inherently correspond to overapproximations—and (2) we can always recover the exact set of results by considering the strongest postcondition.

7.2 A CPS version of the standard big-step judgment

Consider the view of the standard big-step judgment $t/s \Downarrow v/s'$ as a relation modeling a nondeterministic function that, given an input configuration t/s, returns a possible termination outcome v/s'. Now, consider the continuation-passing style (CPS) version of that function. It consists of a function, of the form $t/s \Downarrow Q$ that applies to an input configuration t/s and to a function Q meant to be applied to a possible result of the form v/s'. In CPS, we are free to choose the return type of the continuation. Here, we choose Q to have return type Prop, meaning that Q is a predicate over final configurations. Such a predicate is isomorphic to a set of final configurations.

7.3 Separating preconditions from Hoare triples

Consider a Hoare logic with total correctness triples written $\{H\}\ t\ \{Q\}$, with a precondition H of type state $\rightarrow$ Prop and a postcondition Q of type val $\rightarrow$ state $\rightarrow$ Prop. (Again, this type is isomorphic to sets of final configurations.) Such a triple asserts that, in every state s satisfying the precondition H, every possible evaluation of t/s reaches a final configuration satisfying the postcondition Q.

Many proofs of a triple $\{H\}\ t\ \{Q\}$ start by assuming that H holds for some state s, so one might wonder why H has to be part of the judgment at all, or whether it could just appear on the left of an implication, and indeed it can, thanks to the following equivalence:

$$\{H\}\ t\ \{Q\} \quad \equiv \quad \forall s.\ H\, s \ \Rightarrow\ (t/s \Downarrow Q)$$

On the other hand, the CPS-big-step judgment is equivalent to a Hoare triple with a precondition that characterizes a single state. Let us write $(= s)$ as a shorthand for $\lambda s''.\ s'' = s$. The judgment $t/s \Downarrow Q$ is equivalent to $\{(= s)\}\ t\ \{Q\}$.

7.4 An inductively defined weakest precondition

A weakest precondition generator, written $\mathsf{wp}\, t\, Q$, takes a term t and a desired postcondition Q, and returns the minimal precondition needed for t to execute safely and satisfy Q at the end.

If we have a program verification tool based on $\mathsf{wp}\, t\, Q$ and want to combine it with a verified compiler, what kind of semantics should the compiler use in order to be as close as possible to $\mathsf{wp}\, t\, Q$? As soon as a language has recursion, compiler correctness proofs cannot induct over program syntax any more, but need to induct over executions, so we need an inductively defined predicate for program execution, and it turns out that if we try to define an inductive predicate equivalent to $\mathsf{wp}\, t\, Q$, we obtain exactly CPS big-step semantics:

$$t/s \Downarrow Q \qquad \Longleftrightarrow \qquad \mathsf{wp}\, t\, Q\, s$$

So, if CPS-big-step semantics is *just* a weakest precondition, what is new about it? The inductive definition of $t/s \Downarrow Q$ equips a (nondeterministic) language with an operational semantics. Moreover, it provides a working definition of wp. In contrast, the standard notion of weakest precondition, as axiomatized by reasoning rules such as "$\mathsf{wp}\, t_1\, (\lambda v'.\, \mathsf{wp}\, ([v'/x]\, t_2)\, Q) \vdash \mathsf{wp}\, (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\, Q$" provides a working definition of wp, but does not provide an operational semantics. With a CPS-big-step semantics, one gets in a single shot both the operational semantics and (almost all of) the weakest-precondition reasoning rules. This approach is thus particularly appealing in the context of *foundational* constructions, where weakest-precondition rules need to be formally proved correct with respect to an operational semantics.

Another difference between wp and CPS-big-step concerns rules for while loops—while loops appear in the language used for the case studies in §6. Typically, standard weakest precondition rules for while loops are stated using an invariant; without providing such an invariant, one cannot reason further about the execution of the code. In constrast, our rules provide a rule that essentially unfolds the first iteration of the loop, just like in a standard big-step semantics. From that unfolding rule, one can derive the loop-invariant based rule in just a few lines of proof.

7.5 A typing judgment using propositions instead of syntactic types

Let us argue informally that the CPS-big-step judgment is a direct generalization of a typing judgment. To ease the discussion, let us assume a state-free language. In that setting, the CPS-big-step judgment simplifies to $t \Downarrow Q$. As a first step towards a typing judgment, let us assume an environment-based semantics as opposed to a substitution-based semantics—the two are equivalent—and let us write the evaluation judgment in the form $E \vdash t : Q$. In the rule shown below, the value v_1 produced by t_1 is no longer substituted for x in t_2 but instead bound to x in the environment E.

$$\frac{E \vdash t_1 : Q_1 \qquad \Big(\forall v_1 \in Q_1.\ \ (E, x \mapsto v_1) \vdash t_2 : Q\Big)}{E \vdash (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2) : Q}\ \textsc{cps-big-let revisited}$$

As second step, we can abstract *semantic environments* as *typing environments* in the following sense: rather than binding in the environment a variable x to an arbitrary value v_1 that belongs in the set Q_1, we can view Q_1 as a type and directly bind the variable x to Q_1. The environment becomes $(E, x : Q_1)$, and the resulting rule corresponds exactly to the standard typing rule for let-bindings. This discussion of the similarities between the CPS-big-step judgment and a typing judgment explains well, we believe, the role of the intermediate postcondition Q_1 that appears in the rule cps-big-let: it plays exactly the same role as the type of t_1 in the typing rule for $\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$.

7.6 Adding a postcondition to a safety judgment

In a language with both nondeterminism and crashes (stuck terms), expressing the basic property that no execution starting in configuration t/s gets stuck is cumbersome: It either requires error-propagation rules like the ones shown in the introduction, or a separate safety judgment $\mathsf{safe}(t, s)$ asserting that all possible executions of configuration t/s execute safely, in the sense that they do not get stuck. For instance, Wang et al. [2014] define such a judgment coinductively in big-step style. (A coinductive interpretation means that infinite derivation trees are allowed.) In particular, the rule for let-bindings, shown below, reads as follows: to prove that $\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$ executes safely, we need to show that t_1 executes safely and that, for any possible result v_1 produced by t_1, the result of the substitution $[v_1/x]\, t_2$ executes safely.

$$\textsc{safe-let}\quad \frac{\mathsf{safe}(t_1, s) \qquad \big(\forall v_1 s'.\ (t_1/s \Downarrow v_1/s') \Rightarrow \mathsf{safe}(([v_1/x]\, t_2), s')\big)}{\mathsf{safe}((\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2), s)}\,(\text{co-inductive})$$

Our judgment $t/s \Downarrow Q$ generalizes the notion of safety, by baking the postcondition directly into the judgment. The coinductive interpretation of the rules defining $t/s \Downarrow Q$ yields a judgment, written $t/s \Downarrow^{\text{co}} Q$. This judgment, which is described in more detail in §2.3, asserts that t/s can never get stuck and that any final configuration that it might produce falls in Q. The judgment $t/s \Downarrow^{\text{co}} Q$ is formally related to the predicate safe by the following equivalence.

$$\textsc{cps-co-big-step-iff-safe-and-correct}:\quad t/s \Downarrow^{\text{co}} Q \iff \mathsf{safe}(t, s) \wedge \big(\forall v s'.\ (t/s \Downarrow v/s') \Rightarrow (v, s') \in Q\big)$$

Our rule cps-big-let extends safe-let not just by adding the postcondition Q to the judgment but also by replacing the quantification over all v_1/s'. In safe-let the quantification is constrained by $t_1/s \Downarrow v_1/s'$. In cps-big-let, in contrast, the quantification is constrained by $(v_1, s') \in Q_1$, where Q_1 denotes the postcondition of t_1/s. The gain is that it no longer refers to the standard big-step judgment, making the judgment $t/s \Downarrow Q$ a stand-alone definition of the semantics of the language.

A similar equivalence relates the inductive CPS-big-step judgment with the inductive counterpart of the coinductive predicate safe. In other words, let $\mathsf{terminates}(t, s)$ be the inductive predicate defined by the same rules as the predicate safe. This predicate asserts that every possible execution of t/s terminates safely. We prove the equivalence cps-big-step-iff-terminates-and-correct presented in §2.2, and whose statement is reproduced below.

$$t/s \Downarrow Q \iff \mathsf{terminates}(t, s) \wedge \big(\forall v s'.\ (t/s \Downarrow v/s') \Rightarrow (v, s') \in Q\big)$$

7.7 Hiding existential quantification over final configurations

In the case of deterministic programming languages, correctness proofs for concrete programs or for compilers typically state that the execution reaches a final configuration satisfying some property Q, that is, they prove statements of the form $\exists v s'.\ (t/s \Downarrow v/s') \wedge Q(v, s')$. However, to prove such a statement in the case where t is of the form $\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$, one needs to instantiate upfront the existential variables v and s'. Yet, their values typically depend on variables introduced only during the execution of t_1. The proof would be much easier if carried out by following the execution flow of the program, that is, by first reasoning about the execution of t_1.

It is possible to state a reasoning rule that follows the execution flow. This reasoning style can be achieved by introducing the auxiliary judgment:

$$t/s \Downarrow^{\mathsf{det}} Q \quad\equiv\quad \exists v s'.\ (t/s \Downarrow v/s') \wedge Q(v, s')$$

and using this judgment to state the following let-binding rule:

$$\textsc{det-let}: \qquad t_1/s \Downarrow v_1/s' \quad \wedge \quad ([v_1/x]\,t_2)/s' \Downarrow^{\text{det}} Q \quad \Rightarrow \quad (\text{let}\, x = t_1 \,\text{in}\, t_2)/s \Downarrow^{\text{det}} Q.$$

The rule above only applies to deterministic languages, but appart from that it is not so far from the rule CPS-BIG-LET. Going from DET-LET to CPS-BIG-LET takes three steps. The first step is to view the introduction rule DET-LET not as derived rules but instead as part of an inductive definition. The second step is to change the expression of the first assumption of rule DET-LET from $t_1/s \Downarrow v_1/s'$ to the form $t/s \Downarrow^{\text{det}} Q_1$, introducing an intermediate postcondition Q_1. Doing so not only removes the reference to the standard big-step judgment, it also allows to generalize from one possible result to a set of possible results and thereby to introduce support for nondeterminism. The third step is to introduce, in front of the premise $([v_1/x]\,t_2)/s' \Downarrow^{\text{det}} Q$, a universal quantification over all configurations v_1/s' in Q_1.

8 RELATED AND FUTURE WORK

Semantics for nondeterminism. Nondeterminism appears in the early work on semantics, including the language of guarded commands of Dijkstra [1976] that admits nondeterministic choice where guards overlap, and the par construct of Milner [1975]. Plotkin [1976] develops a *powerdomain* construction to give a fully abstract model in which equivalences such as $(p \,\text{par}\, q) = (q \,\text{par}\, p)$ hold. Francez et al. [1979] also present semantics that map each program to a representation of the set of its possible results. In all these presentations, nondeterminism is *bounded*: only a finite number of choices are allowed.

Subsequent work generalizes the powerdomain interpretation to *unbounded nondeterminism*. For example, Back [1983] considers a language construct $x := \epsilon P$ that assigns x to an arbitrary value satisfying the predicate P—the program has undefined behavior if no such value exists. Apt and Plotkin [1986] address the lack of *continuity* in the models presented in earlier work, still leveraging the notion of powerdomains. Their presentation includes a (countable) nondeterministic assignment operator, written $x := ?$, that sets x to an arbitrary integer in $\mathbb{Z}$.

More recently, powerdomains have been employed for assigning denotational semantics to probabilistic programs, e.g., by Wang et al. [2019]. An interesting question is whether CPS semantics could provide inductively defined operational semantics that account for probabilities, by relating configurations not to sets of outcomes but instead to probability distributions of outcomes.

Concurrent semantics. A concurrent programming language is generally associated with a larger amount of nondeterminism, due to interleavings, and generally includes many more undefined behaviors, due in particular to data races. The work on CompCert TSO [Ševčík et al. 2013] shows how to deal with this additional complexity in a compiler-correctness proof. A direction for future work is to investigate the extent to which CPS semantics help simplifying proofs from CompCert TSO. It would be worth trying both CPS-small-step semantics, which would minimize the changes, and CPS-big-step semantics, which might allow for major simplifications in several proofs.

Semantics of reactive programs. One key question is how much of a program's internal nondeterminism should be reflected in its *execution trace*. At one extreme, one could include a *delay* event, a.k.a. a *tick*, to reflect in the trace each transition performed by the program, following the approaches of Danielsson [2012]. More recent work on interaction trees [Koh et al. 2019; Xia et al. 2019] maps each program to a coinductive structure featuring ticks in addition to I/O steps. Yet, these approaches come at the cost of reasoning "up to removal of a finite number of ticks".

A promising route to avoiding ticks is the *mixed inductive-coinductive* approach of Nakata and Uustalu [2010], for distinguishing between *reactive* programs that always eventually perform I/O operations and *silently diverging* programs that eventually continue executing forever without performing any I/O. So far, the mixed inductive-coinductive approach has not been proved to scale up to realistic compiler proofs. Yet, perhaps such a result could be achieved by combining the mixed inductive-coinductive approach with the CPS-semantics approach.

Improvements to big-step semantics. Leroy and Grall [2009] argue that fairly complex, optimizing compilation passes can be proved correct more easily using big-step semantics than using small-step semantics. These authors propose to reason about diverging executions using *coinductive big-step semantics*, following up on an earlier idea by Cousot and Cousot [1992]. Leroy and Grall's judgment, written $t/s \Uparrow^{\text{co}}$, asserts that there exists a diverging execution of t/s. This judgment is defined coinductively, and a number of its rules refer to the standard big-step judgment. For example, consider the two rules associated with divergence of a let-binding. An expression $\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$ diverges either because t_1 diverges (rule DIV-LET-1) or because t_1 terminates on a value v_1 and the term $[v_1/x]\, t_2$ diverges (rule DIV-LET-2).

$$\frac{t_1/s \Uparrow^{\text{co}}}{(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)/s \Uparrow^{\text{co}}}\ \textsc{div-let-1} \qquad \frac{t_1/s \Downarrow v_1/s' \qquad ([v_1/x]\, t_2)/s' \Uparrow^{\text{co}}}{(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)/s \Uparrow^{\text{co}}}\ \textsc{div-let-2}$$

In contrast, the coinductive CPS-big-step judgment involves a single rule, reproduced below.

$$\frac{t_1/s \Downarrow^{\text{co}} Q_1 \qquad \big(\forall (v_1, s') \in Q_1.\ \ ([v_1/x]\, t_2)/s' \Downarrow^{\text{co}} Q\big)}{(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)/s \Downarrow^{\text{co}} Q}\ \textsc{cps-big-let}$$

If Q_1 is instantiated as the empty set, the second premise of CPS-BIG-LET becomes vacuous, and we recover the rule DIV-LET-1. Otherwise, if Q_1 is nonempty, then it describes the values v_1 that t_1 may evaluate to. For each possible value v_1, the second premise of CPS-BIG-LET requires the term $[v_1/x]\, t_2$ to diverge, just like in the rule DIV-LET-2. Thus, CPS-BIG-LET indeed captures at once both the logic of DIV-LET-1 and DIV-LET-2.

Avoiding duplication in big-step semantics. One issue with coinductive big-step semantics, and also with any standard big-step semantics for a language that includes propagation of exceptions, is the fact that many premises and rules need to be duplicated. This duplication problem is described by Charguéraud [2013], who presents *pretty-big-step semantics* as a way to handle it. The idea is to augment the grammar of terms with *intermediate terms* for representing terms in which a number of subterms have been evaluated. Follow-up work by Bach Poulsen and Mosses [2017] describes an alternative solution that avoids the need for intermediate terms, only at the cost of introducing additional *flag* arguments (Boolean values) both as input and as output of the evaluation judgment. We leave it to future work to investigate the extent to which CPS-big-step semantics handle the duplication problem.

REFERENCES

K. R. Apt and G. D. Plotkin. 1986. Countable Nondeterminism and Random Assignment. *J. ACM* 33, 4 (Aug. 1986), 724–767. https://doi.org/10.1145/6490.6494

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. https://doi.org/10.1016/j.jlamp.2016.05.001

R.J.R. Back. 1983. A continuous semantics for unbounded nondeterminism. *Theoretical Computer Science* 23, 2 (1983), 187–210. https://doi.org/10.1016/0304-3975(83)90055-5

Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP'13)*. Springer-Verlag, Rome, Italy, 41–60. https://doi.org/10.1007/978-3-642-37036-6_3 https://doi.org/10.1007/978-3-642-37036-6_3.

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* (Albuquerque, New Mexico, USA) *(POPL '92)*. Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/143165.143184

Nils Anders Danielsson. 2012. Operational Semantics Using the Partiality Monad. *SIGPLAN Not.* 47, 9 (Sept. 2012), 127–138. https://doi.org/10.1145/2398856.2364546

Edsger W. Dijkstra. 1976. *A Discipline of Programming.* Prentice-Hall. I–XVII, 1–217 pages.

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification Across Software and Hardware for a Simple Embedded System. *PLDI'21* (2021). https://doi.org/10.1145/3453483.3454065.

Nissim Francez, C. A. R. Hoare, Daniel J. Lehmann, and Willem P. De Roever. 1979. Semantics of Nondeterminism, Concurrency, and Communication. *J. Comput. System Sci.* 19, 3 (Dec. 1979), 290–308. https://doi.org/10.1016/0022-0000(79)90006-0 http://www.sciencedirect.com/science/article/pii/0022000079900060.

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2019)*. Association for Computing Machinery, Cascais, Portugal, 234–248. https://doi.org/10.1145/3293880.3294106 https://doi.org/10.1145/3293880.3294106.

Xavier Leroy. 2009. A Formally Verified Compiler Back-End. *Journal of Automated Reasoning* 43, 4 (Dec. 2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4 http://link.springer.com/10.1007/s10817-009-9155-4.

Xavier Leroy and Hervé Grall. 2009. Coinductive big-step operational semantics. *Information and Computation* 207, 2 (2009), 284–304. https://doi.org/10.1016/j.ic.2007.12.004 Special issue on Structural Operational Semantics (SOS).

Robin Milner. 1975. Processes: a mathematical model of computing agents. In *Studies in Logic and the Foundations of Mathematics*. Vol. 80. Elsevier, 157–173.

Jean-François Monin and Xiaomu Shi. 2013. Handcrafted Inversions Made Operational on Operational Semantics. In *Interactive Theorem Proving*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Vol. 7998. Springer Berlin Heidelberg, Berlin, Heidelberg, 338–353. https://doi.org/10.1007/978-3-642-39634-2_25 http://link.springer.com/10.1007/978-3-642-39634-2_25.

Keiko Nakata and Tarmo Uustalu. 2010. Mixed Induction-Coinduction at Work for Coq. *2nd Workshop of Coq users, developers, and contributors* (2010). http://www.cs.ioc.ee/~keiko/papers/Coq2.pdf.

Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press.

G. D. Plotkin. 1976. A Powerdomain Construction. *Siam J. of Computing* (1976).

Tiark Rompf and Nada Amin. 2016. Type Soundness for Dependent Object Types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* ACM, Amsterdam Netherlands, 624–641. https://doi.org/10.1145/2983990.2984008 https://dl.acm.org/doi/10.1145/2983990.2984008.

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (June 2013), 1–50. https://doi.org/10.1145/2487241.2487248 http://dl.acm.org/citation.cfm?doid=2487241.2487248.

Di Wang, Jan Hoffmann, and Thomas Reps. 2019. A Denotational Semantics for Low-Level Probabilistic Programs with Nondeterminism. *Electronic Notes in Theoretical Computer Science* 347 (2019), 303–324. https://doi.org/10.1016/j.entcs.2019.09.016 Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics.

Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. Compiler Verification Meets Cross-Language Linking via Data Abstraction. In *OOPSLA*. ACM Press, 675–690. https://doi.org/10.1145/2660193.2660201

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (Nov. 1994), 38–94. https://doi.org/10.1006/inco.1994.1093 https://doi.org/10.1006/inco.1994.1093.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proceedings of the ACM on Programming*

Languages 4, POPL (Dec. 2019), 51:1–51:32. https://doi.org/10.1145/3371119 https://doi.org/10.1145/3371119.

A EVALUATION RULES WITH MORE THAN TWO PREMISES

So far, we have presented evaluation rules for terms in A-normal form, with applications of the form $(v_1\ v_2)$. The evaluation rule CPS-BIG-LET involves quantifying over one intermediate postcondition Q_1. But what would be the statement of a CPS-big-step reasoning rule for a general application of the form $(t_1\ t_2)$? How would we quantify over the two intermediate postconditions involved?

One possible solution is to say that we can avoid the problem altogether by applying the technique of the pretty-big-step semantics [Charguéraud 2013], which decomposes rules in such a way that no rule contains more than two evaluation premises, and thus no more than one intermediate postcondition. It what follows, we present two other solutions for stating the semantics of $(t_1\ t_2)$ in CPS-big-step form.

A solution based on the evaluation of pairs. Consider the predicate $(t_1, t_2)/s \Downarrow Q$, and the associated evaluation rule shown below.

$$\frac{t_1/s \Downarrow Q_1 \qquad \Big(\forall (v_1, s') \in Q_1.\ \ t_2/s' \Downarrow \{(v_2, s'') \mid ((v_1, v_2), s'') \in Q\}\Big)}{(t_1, t_2)/s \Downarrow Q}\ \text{CPS-BIG-PAIR}$$

If pairs are a construct of the language, then this predicate is just an instance of $t/s \Downarrow Q$. Otherwise, either pairs can be added as an artificial feature of the language, or the predicate $(t_1, t_2)/s \Downarrow Q$ can be viewed as a separate judgment, defined in mutual induction with $t/s \Downarrow Q$. Then the rule for applications could be stated as follows.

CPS-BIG-APP-TERMS-VIA-PAIRS

$$\frac{(t_1, t_2)/s \Downarrow Q_0 \qquad \Big(\forall ((v_1, v_2), s') \in Q_0.\ \ \exists f x t_3.\ \ v_1 = \mu f.\lambda x.t_3\ \wedge\ ([v_1/f]\,[v_2/x]\,t_3)/s' \Downarrow Q\Big)}{(t_1\ t_2)/s \Downarrow Q}$$

One caveat, thought, is that the implementation of Coq (as of 2021) does not automatically generate appropriate induction principles for premises that involve existential quantifiers and conjunctions. A work-around that applies to the rule above is to require Q_0 to only include closures in its first component.

CPS-BIG-APP-TERMS-VIA-PAIRS-WITH-INDUCTION-PRINCIPLE

$$\frac{\begin{array}{c}(t_1, t_2)/s \Downarrow Q_0 \qquad \Big(\forall ((v_1, v_2), s') \in Q_0.\ \ \exists f x t_3.\ \ v_1 = \mu f.\lambda x.t_3\Big) \\ \Big(\forall ((v_1, v_2), s') \in Q_0.\ \ \forall f x t_3.\ \ v_1 = \mu f.\lambda x.t_3\ \Rightarrow\ ([v_1/f]\,[v_2/x]\,t_3)/s' \Downarrow Q\Big)\end{array}}{(t_1\ t_2)/s \Downarrow Q}$$

A solution that does not involve pairs. If we really want to avoid the introduction of pairs or the introduction of an auxiliary judgment, then we need the statement of the evaluation rule for $t_1\ t_2$ to involve existential quantification of intermediate postconditions.

CPS-BIG-APP-TERMS

$$\frac{\begin{array}{l}\qquad\qquad\qquad\qquad t_1/s \Downarrow Q_1 \\ \Big(\forall (v_1, s') \in Q_1.\ \ \exists Q_2.\ \ t_2/s' \Downarrow Q_2\ \wedge \\ \qquad \Big(\forall (v_2, s'') \in Q_2.\ \ \exists f x t_3.\ \ v_1 = \mu f.\lambda x.t_3\ \wedge\ ([v_1/f]\,[v_2/x]\,t_3)/s'' \Downarrow Q\Big)\Big)\end{array}}{(t_1\ t_2)/s \Downarrow Q}$$

Here again, we need to adapt the statement of the rule if we want Coq to generate a usable induction principle. The idea is to pull the existential quantifiers on Q_2 at the top of the rule, by making Q_2 be a function of s'. Then, we are able to split the conjuncts, at the cost of duplicating a few quantifiers.

CPS-BIG-APP-TERMS-WITH-INDUCTION-PRINCIPLE

$$\frac{\begin{array}{c} t_1/s \Downarrow Q_1 \qquad \Big(\forall (v_1, s') \in Q_1.\ \ t_2/s' \Downarrow Q_2(s')\Big) \\ \Big(\forall s'.\ \forall (v_1, s'') \in Q_1(s').\ \ \exists f x t_3.\ \ v_1 = \mu f.\lambda x.t_3\Big) \\ \Big(\forall (v_1, s') \in Q_1.\ \ \forall (v_2, s'') \in Q_2(s').\ \ \forall f x t_3.\ \ v_1 = \mu f.\lambda x.t_3 \ \Rightarrow\ ([v_1/f]\,[v_2/x]\,t_3)/s'' \Downarrow Q\Big) \end{array}}{(t_1\, t_2)/s \Downarrow Q}$$

Remark: to prove this reformulated rule equivalent to CPS-BIG-APP-TERMS, one needs to exploit classical logic and the axiom of choice. This encoding is quite involved, thus we would recommand following one of the other approaches described: either assume A-normal form (input programs can be A-normalized once and for all, at the very start of a verified tool chain), or assume a language with pairs, or extend the language with pairs, or use an auxiliary judgment for evaluating a pair of terms, or follow the pretty-big-step presentation to evaluate subterms one at a time.

B INDUCTIVE DEFINITIONS OF WP (LET CASE)

The reader may wonder whether it would be possible to define wp directly as an inductive judgment. For example, one might attempt to define wp for let-bindings as follows:

$$\mathsf{wp}\ t_1\ (\lambda v'.\, \mathsf{wp}\ ([v'/x]\ t_2)\ Q) \ \ \vdash\ \ \mathsf{wp}\ (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\, Q$$

where $(\vdash)$ denotes the entailment relation ($H \vdash H'$ is defined as $\forall s.\, H\, s \Rightarrow H'\, s$). Yet, such a rule is not accepted by Coq as part of an inductive definition for wp because the nested occurrence of wp in the premise is not a *positive occurence*. The relevant 3 candidate rules are shown below, using Coq syntax. The rule `wp_let_invalid` features the non-positive occurrence. The rule `wp_let_valid` avoids the issue using an existential quantifier for the intermediate postcondition Q_1, however Coq generates no useful induction principle. The rule `wp_let_good` corresponds to the rule CPS-BIG-LET, where the intermediate postcondition Q_1 is quantified at the level of the rule, and for which Coq can produce a useful induction principle.

```
Notation "H1 ⊢H2" := (∀ s, H1 s → H2 s).
Inductive wp : trm → (val→ state→ Prop) → (state→ Prop) :=
  | wp_let_invalid : ∀x t1 t2 Q, (* non strictly positive occurrence of [wp]. *)
      wp t1 (fun v1 ⇒ wp (subst x v1 t2) Q)
      ⊢wp (trm_let x t1 t2) Q
  | wp_let_valid : ∀x t1 t2 Q, (* no induction hypothesis generated by Coq *)
      (fun s ⇒ ∃Q1, wp t1 Q1 s ∧ (∀ v1, Q1 v1 ⊢wp (subst x v1 t2) Q))
      ⊢wp (trm_let x t1 t2) Q
  | wp_let_good : ∀x t1 t2 Q1 Q, (* comes with an induction hypothesis *)
      wp t1 Q1 s →
      (∀ v1 s2, Q1 v1 s2 → wp (subst x v1 t2) Q s2)) →
      wp (trm_let x t1 t2) Q s
```

C DEFINITION OF THE STANDARD SMALL-STEP JUDGMENT

$$\textsc{small-app}\quad \frac{v_1 = (\mu f.\lambda x.t)}{(v_1\, v_2)/s \longrightarrow ([v_2/x]\,[v_1/f]\,t)/s}$$

$$\textsc{small-if-true}\quad \frac{}{(\text{if true then } t_1 \text{ else } t_2)/s \longrightarrow t_1/s}$$

$$\textsc{small-if-false}\quad \frac{}{(\text{if false then } t_1 \text{ else } t_2)/s \longrightarrow t_2/s}$$

$$\textsc{small-let-ctx}\quad \frac{t_1/s \longrightarrow t_1'/s'}{(\text{let}\, x = t_1 \,\text{in}\, t_2)/s \longrightarrow (\text{let}\, x = t_1' \,\text{in}\, t_2)/s'}$$

$$\textsc{small-let-val}\quad \frac{}{(\text{let}\, x = v_1 \,\text{in}\, t_2)/s \longrightarrow ([v_1/x]\, t_2)/s}$$

$$\textsc{small-add}\quad \frac{}{(\text{add}\, n_1\, n_2)/s \longrightarrow (n_1 + n_2)/s}$$

$$\textsc{small-rand}\quad \frac{0 \le m < n}{(\text{rand}\, n)/s \longrightarrow m/s}$$

$$\textsc{small-ref}\quad \frac{p \notin \text{dom}\, s}{(\text{ref}\, v)/s \longrightarrow (s[p := v])/s}$$

$$\textsc{small-free}\quad \frac{p \in \text{dom}\, s}{(\text{free}\, p)/s \longrightarrow tt/(s \setminus p)}$$

$$\textsc{small-get}\quad \frac{p \in \text{dom}\, s}{(\text{get}\, p)/s \longrightarrow (s[p])/s}$$

$$\textsc{small-set}\quad \frac{p \in \text{dom}\, s}{(\text{set}\, p\, v)/s \longrightarrow tt/(s[p := v])}$$

D DEFINITION OF THE TYPING JUDGMENT

We define the typing rules for terms in A-normal form. The judgment $\vdash\, v \,:\, T$ asserts that the closed value t admits the type T. The judgment $E \,\vdash\, t \,:\, T$ asserts that the term t admits type T in the environment E. Finally, $\mathbb{V}$ denotes the set of terms that are either values or variables.

$$\textsc{vtyp-unit}\quad \frac{}{\vdash\, tt \,:\, \text{unit}}$$

$$\textsc{vtyp-bool}\quad \frac{}{\vdash\, b \,:\, \text{bool}}$$

$$\textsc{vtyp-int}\quad \frac{}{\vdash\, n \,:\, \text{int}}$$

$$\textsc{vtyp-fix}\quad \frac{f : (T_1 \to T_2),\, x : T_1 \,\vdash\, t \,:\, T_2}{\vdash\, ((\mu f.\lambda x.t)) \,:\, (T_1 \to T_2)}$$

$$\textsc{typ-val}\quad \frac{\vdash\, v \,:\, T}{E \,\vdash\, v \,:\, T}$$

$$\textsc{typ-var}\quad \frac{x \in \text{dom}\, E \qquad E[x] = T}{E \,\vdash\, x \,:\, T}$$

$$\textsc{typ-fix}\quad \frac{E,\, f : (T_1 \to T_2),\, x : T_1 \,\vdash\, t \,:\, T_2}{E \,\vdash\, (\mu f.\lambda x.t) \,:\, T}$$

$$\textsc{typ-app}\quad \frac{E \,\vdash\, t_1 \,:\, (T_1 \to T_2) \qquad E \,\vdash\, t_2 \,:\, T_1 \qquad t_1, t_2 \in \mathbb{V}}{E \,\vdash\, (t_1\, t_2) \,:\, T_2}$$

$$\textsc{typ-if}\quad \frac{E \,\vdash\, t_0 \,:\, \text{bool} \qquad E \,\vdash\, t_1 \,:\, T \qquad E \,\vdash\, t_2 \,:\, T \qquad t_0 \in \mathbb{V}}{E \,\vdash\, (\text{if } t_0 \text{ then } t_1 \text{ else } t_2) \,:\, T}$$

$$\textsc{typ-let}\quad \frac{E \,\vdash\, t_1 \,:\, T_1 \qquad E,\, x : T_1 \,\vdash\, t_2 \,:\, T_2}{E \,\vdash\, (\text{let}\, x = t_1 \,\text{in}\, t_2) \,:\, T_2}$$

$$\textsc{typ-add}\quad \frac{E \,\vdash\, t_1 \,:\, \text{int} \qquad E \,\vdash\, t_2 \,:\, \text{int} \qquad t_1, t_2 \in \mathbb{V}}{E \,\vdash\, (\text{add}\, t_1\, t_2) \,:\, \text{int}}$$

$$\textsc{typ-rand}\quad \frac{E \,\vdash\, t_1 \,:\, \text{int} \qquad t_1 \in \mathbb{V}}{E \,\vdash\, (\text{rand}\, t_1) \,:\, \text{int}}$$

E EVALUATION OF UNARY AND BINARY OPERATORS

$$\frac{}{\mathsf{evalunop}(\mathsf{fst}, (v_1, v_2), v_1)} \qquad \frac{}{\mathsf{evalunop}(\mathsf{snd}, (v_1, v_2), v_2)} \qquad \frac{}{\mathsf{evalunop}(\mathsf{not}, 1, 0)}$$

$$\frac{}{\mathsf{evalunop}(\mathsf{not}, 0, 1)} \qquad \frac{}{\mathsf{evalbinop}(+, n_1, n_2, n_1 + n_2)} \qquad \frac{}{\mathsf{evalbinop}(\mathsf{mkpair}, v_1, v_2, (v_1, v_2))}$$

F CPS EVALUATION RULES IN THE PRESENCE OF EXCEPTIONS

For a programming language that features exceptions, the reasoning rule for let-bindings needs to be adapted in two ways. Indeed, if the body of the let-binding raises an exception, then the continuation should not be evaluated, and that exception should be included in the set of results that the let-binding can produce.

Assume the grammar of values to be extended with an exception construct exn, which we here assume to not carry a value, for simplicity. The CPS-big-step evaluation rule can be stated as follows.

CPS-BIG-LET-WITH-EXCEPTIONS

$$\frac{t_1/s \Downarrow Q_1 \quad \Big(\forall (v', s') \in Q_1.\ \ v' \neq \mathsf{exn} \ \Rightarrow\ ([v'/x]\, t_2)/s' \Downarrow Q\Big) \quad \Big(\forall s'.\ \ Q_1\, \mathsf{exn}\, s' \ \Rightarrow\ Q\, \mathsf{exn}\, s'\Big)}{(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)/s \Downarrow Q}$$

Observe that a single rule handles both the case where t_1 produces a normal value and the case where it produces an exception.

We proved the equivalence of the above rule with the standard small-step and big-step semantics for a language with (non-catchable, argument-free) exceptions. We leave to future work the set up of CPS-semantics for a language that includes full-featured exceptions.

G UNSPECIFIED EVALUATION ORDER

For a language that does not specify the order of evaluation for arguments of, e.g., pairs, or applications, we can consider a generalized version of the rule CPS-BIG-PAIR from the previous section. Essentially, we duplicate the premises to account for the two possible evaluation orders.

CPS-BIG-PAIR-UNSPECIFIED-ORDER

$$\frac{\begin{array}{ll} t_1/s \Downarrow Q_1 & \Big(\forall (v_1, s') \in Q_1.\ \ t_2/s' \Downarrow \{(v_2, s'') \mid ((v_1, v_2), s'') \in Q\}\Big) \\ t_2/s \Downarrow Q_2 & \Big(\forall (v_2, s') \in Q_2.\ \ t_1/s' \Downarrow \{(v_1, s'') \mid ((v_1, v_2), s'') \in Q\}\Big) \end{array}}{(t_1, t_2)/s \Downarrow Q}$$

To avoid duplication in the premises, one can follow the approach described in §5.5 of the paper on the pretty-big-step semantics [Charguéraud 2013], which presents a general rule for evaluating a list of subterms in arbitrary order.

H AN ALTERNATIVE EVENTUALLY JUDGMENT, EXPRESSED WITH THE STANDARD SMALL-STEP JUDGMENT

The judgment $t/s \longrightarrow^{\Diamond} P$ captures the property that every possible evaluation of t/s is safe and eventually reaches a configuration in the set P. In the paper, we define it in terms of the CPS-small-step judgment. We here note that it can also be defined directly in terms of the standard small-step judgment. The variant definition involves the following two rules. The first one asserts, as before, that the judgment is satisfied if t/s belongs to P. The second rule asserts that the judgment is satisfied if t/s is not stuck and that for any configuration t'/s' that it may reduce to (with respect to the small-step evaluation judgments), the predicate $t'/s' \longrightarrow^{\Diamond} P$ holds.

$$\textsc{eventually-here}\quad \frac{(t, s) \in P}{t/s \longrightarrow^{\Diamond} P} \qquad \textsc{eventually-step-using-standard-small-step}\quad \frac{\big(\exists t's'.\;\; t/s \longrightarrow t'/s'\big) \qquad \big(\forall t's'.\;\; (t/s \longrightarrow t'/s') \;\Rightarrow\; (t'/s' \longrightarrow^{\Diamond} P)\big)}{t/s \longrightarrow^{\Diamond} P}$$

This alternative definition might be of interest if one already has at hand a small-step judgment at hand and does not with to state a CPS-small-step judgment. However, the original definition presented in the paper is both more concise and more practical for carrying out proofs.