Omnisemantics: Smooth Handling of Nondeterminism

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 gives an in-depth presentation of the omni-big-step and omni-small-step styles of semantic judgments. These styles 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 (hence the name *omni*), whereas in traditional small-step and big-step semantics, each derivation only talks about one single execution. This restructuring allows for straightforward modeling of both nondeterminism and undefined behavior as commonly encountered in sequential functional and imperative programs. Specifically, omnisemantics 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.

Omnisemantics can be understood as an inductively defined weakest-precondition semantics (or more generally, predicate-transformer semantics) that does not involve invariants for loops and recursion but instead uses unrolling rules like in traditional small-step and big-step semantics. Omnisemantics were previously described in association with several projects, but we believe the technique has been underappreciated and deserves a well-motivated, extensive, and pedagogical presentation of its benefits. We also explore several novel aspects associated with these semantics, in particular their use in type-safety proofs for lambda calculi, partial-correctness reasoning, and forward proofs of compiler correctness for terminating but potentially nondeterministic programs being compiled to nondeterministic target languages. All results in this paper are formalized in Coq.

CCS Concepts: • **Theory of computation → Operational semantics**; **Axiomatic semantics**; **Pre- and post-conditions**; **Program verification**; **Hoare logic**; **Separation logic**.

Additional Key Words and Phrases: Nondeterminism, Termination, Compiler Correctness Proofs

ACM Reference Format:
Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2018. Omnisemantics: Smooth Handling of Nondeterminism. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2018), 43 pages. https://doi.org/10.1145/3579834

1 INTRODUCTION

Today, a typical project in rigorous reasoning about programming languages begins with an operational semantics (or maybe several), with proofs of key lemmas proceeding by induction on derivations of the semantics judgment. 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. Yet, a style that we call *omnisemantics* has emerged in recent years as a new, powerful technique with numerous applications.

In short, omnisemantics relate starting states to their sets of possible outcomes, rather than to individual outcomes. The omni-big-step judgment takes the form $t/s \Downarrow Q$ and asserts that every

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.

2018. 0164-0925/2018/1-ART1 $15.00
https://doi.org/10.1145/3579834

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. The omni-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, one may 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.

On the one hand, omnisemantics can be viewed as *operational semantics*, because they are not far from traditional operational semantics or executable interpreters. On the other hand, omnisemantics can be viewed as *axiomatic semantics*, because they are not far form reasoning rules; in particular, they directly give a practical, usable definition of a weakest-precondition judgment, which can be used for verifying concrete programs. The fact that they are both closely related to operational semantics and to axiomatic semantics is precisely the strength of omnisemantics.

To the best of our knowledge, the ideas of omnisemantics have been studied prior to the writing of this paper by three different groups of researchers. First, Schäfer et al. [2016] present an omni-big-step judgment for a nondeterministic source language of guarded commands, as well as for a deterministic target language with named continuations, using the term *axiomatic semantics* to refer to this style of semantics. They establish the correctness of a function that compiles terminating programs from the source language into the target language. Their proof is by induction on the derivation of an omni-big-step judgment for the *source* program rather than on a derivation for the *target* program, a key insight that we will discuss in Sections 1.3 and 6. They also present characterizations of program equivalence and present a proof of equivalence with traditional small-step semantics, though only in the case of a deterministic semantics. Second, Erbsen et al. [2021] make use of both omni-big-step semantics, applied to a high-level, core imperative language with external calls; and of omni-small-step semantics, applied to a low-level, RISC-V machine language. They call this style of semantics *CPS semantics*. They establish end-to-end compiler-correctness results for terminating programs. They also set up Separation Logic reasoning rules in weakest-precondition style. Third, Charguéraud [2020]'s course notes make use of omni-big-step semantics for the purpose of deriving Separation Logic triples, for both partial and total correctness. The language considered is a nondeterministic, imperative λ-calculus, with a substitution-based semantics. In particular, that work establishes the relationship between omni-big-step semantics and traditional big-step semantics, in the presence of nondeterminism.

Throughout the three pieces of work, the fundamental feature of omnisemantics being exploited is the ability to carry out proofs by induction on derivations that follow the flow of program execution, *with smooth handling of nondeterminism*. Indeed, nondeterministic choices result in universally quantified induction hypotheses at steps where nondeterministic choices are made. Before further presenting omnisemantics, we believe that it is useful to begin by presenting in more detail the several important problems that omnisemantics solve.

1.1 Feature #1: Stuck Terms and Nondeterminism

In an impure language, an execution may get stuck, for instance due to a division by zero or an out-of-bounds array access. In a nondeterministic language, some executions may get stuck 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 getting stuck is impossible.

How to fix the problem? A popular but cumbersome approach is to add errors as explicit outcomes (written err in the rules below), so that we can state theorems ruling out stuck terms. 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. Omni-big-step semantics provide a way to reason, in big-step style, about the absence of stuck terms in nondeterministic languages without introducing error-propagation rules.

1.2 Feature #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 foundational approaches, Hoare triples must be defined in terms of or otherwise formally related to the operational semantics of languages.

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, but 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 becomes 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 traditional big-step judgment, we do not know of any direct way to capture the fact that *all* executions terminate. Omni-big-step semantics provide a direct definition of 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.

1.3 Feature #3: 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 showing *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 [Blazy and Leroy 2009]. 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. Omnisemantics make it possible to retain the simplicity of forward simulation, while keeping nondeterminism explicit.

1.4 Feature #4: Linear-Size Type-Safety Proofs

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

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

The Wright and Felleisen approach, although widely used, suffers from two limitations that can be problematic at the scale of real-world languages 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. The trouble is that 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[1]. As we expect to handle each possible transition at most once, a proof that takes only linear work would be more satisfying. It would also avoid potential blow-up in the proof-checking time, for languages involving hundreds of constructs.

Interestingly, in the particular case of a deterministic language, there exists a strategy [Rompf and Amin 2016] for deriving type safety 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 can step to a term that admits the same type.

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

Omnisemantics allow to generalize this approach to the case of nondeterministic languages. As we show in one of this paper's original contributions, practical proofs of type safety can be carried out with respect to both omni-small-step and omni-big-step semantics.

1.5 Contributions and Contents of the Paper

The contributions of this paper are as follows.

- We present big-step and small-step omnisemantics for a standard imperative λ-calculus as well as for a standard imperative while language, which we believe should make the presentation

[1] Lean matches Coq, and a proof based on Agda's flexible dependent pattern matching still takes superlinear time to check.

more accessible than in prior publications. Moreover, we accompany this presentation with a Coq formalization of all definitions and proofs.[2]

- We explain four key beneficial features of omnisemantics: They provide a convenient way to reason about the absence of stuck terms (feature #1) and the absence of diverging terms (feature #2) in nondeterministic languages, they enable forward-simulation-based correctness proofs for compilers with nondeterministic target languages (feature #3), and they enable type-safety proofs that avoid quadratic case inspection even in the case of a nondeterministic language (feature #4).
- We introduce the coinductive variant of omni-big-step semantics, which yields a partial-correctness judgment. This possibility was left as future work by Schäfer et al. [2016].
- We present numerous properties of omnisemantics, as well as their relationship to traditional operational semantics. Some of these properties were described in Erbsen et al. [2021] but only briefly. For example, the connection between traditional and omnisemantics only covered traditional small-step semantics with no undefined behavior, and small-step omnisemantics themselves were given one paragraph of description.
- We present in detail the proof techniques from two case studies on compiler-correctness results, adapted from Erbsen et al. [2021]'s prior work.
- We present a new case study illustrating an example of a correctness proof for a compiler transformation that *increases* the amount of nondeterminism. In contrast, work by Schäfer et al. [2016] and Erbsen et al. [2021] only considered transformations that *decrease* the amount of nondeterminism.

The paper is organized as follows.

- In Section 2, we introduce the omni-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 omni-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-safety proofs carried out with respect to either omni-small-step or omni-big-step semantics. We explain the improvement over the prior state of the art, as suggested in the earlier discussion of features #1 and #4.
- In Section 5, we explain how the omni-big-step judgment or the omni-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 (recall feature #2). Moreover, we explain how one may derive the frame rule from Separation Logic.
- In Section 6, we demonstrate how omnisemantics can be used to prove that a compiler correctly compiles terminating programs, via forward-simulation proofs (recall feature #3). 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 omni-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 an omni-*big*-step semantics for the source language and an omni-*small*-step semantics for the target language.

[2]The present paper would, in particular, provide a formal publication of the results covered by the chapter on nondeterminism and the chapter on partial correctness from Charguéraud's *Separation Logic Foundations* course, Volume 6 of the Software Foundations series. These results originally covered only omni-big-step semantics but have been extended in 2021 to cover omni-small-step semantics as well.

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

2 OMNI-BIG-STEP SEMANTICS

In the section, we introduce the omni-big-step judgment, written $t/s \Downarrow Q$. We use this judgment in particular for establishing type safety (§4.3), for setting up program logics (§5), and for establishing compiler-verification results (§6). To present the definition of this judgment, we consider an imperative, nondeterministic lambda-calculus, for which we first present the semantics in standard big-step style (§2.1). We then discuss the properties and interpretation of the omni-big-step judgment (§2.2). In particular, we focus on why the set Q that appears in $t/s \Downarrow Q$ is interpreted as an *overapproximation* of the set of possible results, rather than as the *exact* set of possible results. We next present the corresponding *coinductive* judgment, written $t/s \Downarrow^{\text{co}} Q$, which captures partial correctness in the sense that it allows for diverging executions (§2.4). We conclude this section by presenting the *bind rule* for handling programs that are not in A-normal form (§2.5).

2.1 Definition of the Omni-Big-Step Judgment

Syntax. As a running example, we consider an imperative lambda-calculus, including a random-number generator rand. Both this operator and allocation are nondeterministic.

The grammar of the language appears next. The metavariable π ranges over primitive operations, v ranges over 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, a primitive operator, or a closure[3].

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

For simplicity, we present evaluation rules by focusing first on programs in A-normal form: the let-binding construct is the only one that involves evaluation under a context. 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. In §2.5, we present the *bind* rule, which enables the evaluation of subterms under all valid evaluation contexts.

Evaluation judgments. The standard big-step-semantics judgment for this language appears in Figure 1. States s are finite 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 omni-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 omni-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 in §2.3.

The set Q contains pairs made of values and states. 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

[3] In our Coq formalization, the grammar of values is restricted to *closed* values (i.e., values without free variables). This design choice significantly simplifies the reasoning about substitutions. One minor consequence is that the function construct needs to appear twice: once in the grammar of closed values and once in the grammar of terms.

$$\text{BIG-VAL}\ \frac{}{v/s \Downarrow v/s} \qquad \text{BIG-APP}\ \frac{v_1 = (\mu f.\lambda x.t) \qquad ([v_2/x]\,[v_1/f]\,t)/s \Downarrow v'/s'}{(v_1\, v_2)/s \Downarrow v'/s'} \qquad \text{BIG-IF-TRUE}\ \frac{t_1/s \Downarrow v'/s'}{(\text{if true then } t_1 \text{ else } t_2)/s \Downarrow v'/s'}$$

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

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

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

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

$$\text{OMNI-BIG-VAL}\ \frac{(v, s) \in Q}{v/s \Downarrow Q} \qquad \text{OMNI-BIG-IF-TRUE}\ \frac{t_1/s \Downarrow Q}{(\text{if true then } t_1 \text{ else } t_2)/s \Downarrow Q} \qquad \text{OMNI-BIG-IF-FALSE}\ \frac{t_2/s \Downarrow Q}{(\text{if false then } t_1 \text{ else } t_2)/s \Downarrow Q}$$

$$\text{OMNI-BIG-APP}\ \frac{\begin{array}{c} v_1 = \mu f.\lambda x.t_1 \\ ([v_1/f]\,[v_2/x]\,t_1)/s \Downarrow Q \end{array}}{(v_1\, v_2)/s \Downarrow Q} \qquad \text{OMNI-BIG-LET}\ \frac{\begin{array}{c} t_1/s \Downarrow Q_1 \\ \big(\forall (v', s') \in Q_1.\ \ ([v'/x]\,t_2)/s' \Downarrow Q\big) \end{array}}{(\text{let } x = t_1 \text{ in } t_2)/s \Downarrow Q} \qquad \text{OMNI-BIG-ADD}\ \frac{(n_1 + n_2,\ s) \in Q}{(\text{add}\, n_1\, n_2)/s \Downarrow Q}$$

$$\text{OMNI-BIG-RAND}\ \frac{n > 0 \qquad \big(\forall m.\ \ 0 \le m < n \ \Rightarrow\ (m, s) \in Q\big)}{(\text{rand}\, n)/s \Downarrow Q} \qquad \text{OMNI-BIG-REF}\ \frac{\forall p \notin \text{dom}\, s.\ \ (p,\ s[p := v]) \in Q}{(\text{ref}\, v)/s \Downarrow Q}$$

$$\text{OMNI-BIG-FREE}\ \frac{p \in \text{dom}\, s \qquad (tt,\ s \setminus p) \in Q}{(\text{free}\, p)/s \Downarrow Q} \qquad \text{OMNI-BIG-GET}\ \frac{p \in \text{dom}\, s \qquad (s[p],\ s) \in Q}{(\text{get}\, p)/s \Downarrow Q} \qquad \text{OMNI-BIG-SET}\ \frac{p \in \text{dom}\, s \qquad (tt,\ s[p := v]) \in Q}{(\text{set}\, p\, v)/s \Downarrow Q}$$

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

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. (The type of Q may be generalized for languages that include exceptions; see Appendix C.)

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

The let-binding rule OMNI-BIG-LET ensures that all possible evaluations of an expression let $x = t_1$ 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 $t_1/s \Downarrow Q_1$. One can think of Q_1 as the type of t_1, in a very precise type system where any set of values can be treated as a type. The second premise asserts that, 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.

The evaluation rule OMNI-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 OMNI-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 OMNI-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, OMNI-BIG-FREE, OMNI-BIG-GET and OMNI-BIG-SET, are deterministic and follow the same pattern as OMNI-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 Omni-Big-Step Judgment

In this section, we discuss some key properties of the omni-big-step judgment $t/s \Downarrow Q$. Recall that the metavariable 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 Appendix D). We prove:

OMNI-BIG-STEP-IFF-TERMINATES-AND-CORRECT :
$$t/s \Downarrow Q \quad\Longleftrightarrow\quad \mathsf{terminates}(t, s) \;\wedge\; \Big(\forall v s'.\; (t/s \Downarrow v/s') \Rightarrow (v, s') \in Q\Big).$$

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.

$$\text{OMNI-BIG-CONSEQUENCE :} \qquad t/s \Downarrow Q \quad\wedge\quad Q \subseteq Q' \quad\Rightarrow\quad t/s \Downarrow Q'$$

Strongest postcondition. If the omni-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.

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

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 we have: $\forall Q.\; \neg\, (t/s \Downarrow Q)$.

$$\text{PRECISE-BIG-VAL}\quad \frac{}{v/s \Downarrow' \{(v,s)\}} \qquad \text{PRECISE-BIG-REF}\quad \frac{}{(\mathsf{ref}\,v)/s \Downarrow' \{(p, s[p := v]) \mid p \notin \mathrm{dom}\, s\}}$$

$$\text{PRECISE-BIG-RAND}\quad \frac{n > 0}{(\mathsf{rand}\,n)/s \Downarrow' \{(m,s) \mid 0 \le m < n\}} \qquad \text{PRECISE-BIG-LET}\quad \frac{t_1/s \Downarrow' Q_1 \qquad \forall (v',s') \in Q_1.\ ([v'/x]\,t_2)/s' \Downarrow Q'_{(v',s')}}{(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)/s \Downarrow' \bigcup_{(v',s') \in Q_1} Q'_{(v',s')}}$$

Fig. 3. Selected rules defining a *precise* variant of omni-big-step semantics, written $t/s \Downarrow' Q$.

Relationship to standard big-step semantics. The standard big-step judgment $t/s \Downarrow v/s'$ relates one input configuration t/s to one single result configuration v/s'. The omni-big-step judgment, which relates inputs to sets of results, thus appears as an immediate generalization of the standard big-step judgment. The following two results formalizes their relationship.

First, if $t/s \Downarrow Q$ holds, then any final configuration for which the standard big-step judgment holds necessarily belongs to the set Q.

$$\text{OMNI-BIG-AND-BIG-INV:} \qquad t/s \Downarrow Q \quad \wedge \quad t/s \Downarrow v/s' \quad \Rightarrow \quad (v,s') \in Q$$

Second, if $t/s \Downarrow Q$ holds, then there exists at least one evaluation according to the standard big-step judgment whose final configuration belongs to the set Q.

$$\text{OMNI-BIG-TO-ONE-BIG:} \qquad t/s \Downarrow Q \quad \Rightarrow \quad \exists vs'.\ \ t/s \Downarrow v/s' \ \wedge\ (v,s') \in Q$$

A corollary asserts that if $t/s \Downarrow Q$ holds with Q being a singleton set made of a unique final configuration v/s', then the standard big-step judgment holds for that configuration.

$$\text{OMNI-BIG-SINGLETON:} \qquad t/s \Downarrow \{(v,s')\} \quad \Rightarrow \quad t/s \Downarrow v/s'$$

Particular case of deterministic languages. In a deterministic language, an input configuration t/s may evaluate to at most one configuration v/s'. In such a case, the strongest postcondition is reduced to the singleton set $\{(v,s')\}$.

Nonempty outcome sets. Observe that the judgment $t/s \Downarrow Q$, as defined in Fig. 2, can only hold for a nonempty set Q. When designing omni-big-step rules for a new language, one has to be careful not to accidentally include rules that allow derivations of empty outcome sets for some programs. To illustrate the matter, consider the term "rand 0". According to the standard big-step semantics, this term is stuck because the rule BIG-RAND requires a positive argument to rand. In the omni-big-step semantics, if we were to omit the premise $n > 0$ in the rule OMNI-BIG-RAND, we would be able to derive $(\mathsf{rand}\,0)/s \Downarrow Q$ for any s and Q. Indeed, the premise $\forall m.\ 0 \le m < n \Rightarrow (m,s) \in Q$ becomes vacuously true when n is nonpositive.

A similar subtlety appears in the rule OMNI-BIG-REF, where the fresh location p must be picked fresh from the domain of s. This quantification could become vacuously true if the semantics allowed for infinite states or if the set of memory locations were finite. (We discuss in §6.5 the treatment of a language whose semantics account for a finite memory.)

The likelihood of unadequate formalization due to missing premises might be viewed as the main weakness of omnisemantics. Yet, if needed, additional confidence can easily be restored at the cost of minor additional work: one may consider a standard small-step semantics as reference (i.e., as part of the trusted code base), then relate it to the corresponding omni-big-step semantics and use the latter to carry out big-step style, inductive proofs on nondeterministic executions.

2.3 About the Overapproximation of the Set of Results

The omni-big-step judgment $t/s \Downarrow Q$ associates an initial configuration t/s with a postcondition Q, which denotes an *overapproximation* of the set of possible final configurations. One may thus wonder: why not associate it with a *precise* set of results? In this section, we show that it is technically possible to define a *precise* judgment, but at the same time we argue why that judgment is much less practical to work with than the *overapproximating* omni-big-step judgment.

The precise judgment, written $t/s \Downarrow' Q$, is precise in the sense that it relates a configuration t/s to *at most one* set of results Q. This precise judgment, like the overapproximating omni-big-step judgment, guarantees safety: a judgment $t/s \Downarrow' Q$ can be derived for some Q if and only if none of the possible executions of t/s can get stuck. Thus, the precise judgment relates a *safe* configuration t/s to exactly one Q.

Figure 3 shows selected rules from the definition of the precise judgment, written $t/s \Downarrow' Q$. The rule PRECISE-BIG-VAL relates a value v in a state s to the singleton set made of the pair (v, s). The rule PRECISE-BIG-REF relates the term $(\mathsf{ref}\, v)$ in a state s to the set of pairs made of a location p fresh from s and of the state s updated at location p with the value v. Observe how this compares with the rule OMNI-BIG-REF, which only requires that set of pairs to be included in the result set Q. The rule PRECISE-BIG-RAND follows a similar pattern, only with the premise $n > 0$ to ensure that the term is not stuck.

Most interesting is the rule PRECISE-BIG-LET. Its first premise involves an intermediate set Q_1, which denotes *exactly* the set of results that t_1 can produce when executed in the input state s. The second premise describes, for each result (v', s') from the set Q_1, the evaluation of $([v'/x]\, t_2)$ in state s'. The result of the execution is asserted to be exactly a set of configurations written $Q'_{(v',s')}$. Here Q' denotes a (possibly infinite) family of postconditions, indexed by the possible results of t_1. The final postcondition of the term $(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)$ is obtained by taking the union over that family of postconditions.[4]

In practice, working with indexed families of postconditions introduces significant overhead, compared with the overapproximating omni-big-step judgment. Moreover, for practical applications such as type-checking or program verification (either using weakest preconditions or Hoare triples), we are only interested in overapproximations of the semantics. For such applications, building the overapproximation on top of a precise judgment would only introduce a level of indirection. For other situations where a notion of exact set of results might be desirable, typically for metatheoretical results (e.g., completeness results), we can always refer to the *strongest postcondition*, which, as explained earlier, can be formalized as the intersection of all valid postconditions.

In summary, we believe that it is interesting to know that a precise judgment can be defined, as it might be useful in other contexts, but for the applications that we have in mind the overapproximating omni-big-step judgment appears much better suited.

2.4 Coinductive Interpretation of the Omni-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 omni-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 in a final configuration satisfying Q. In particular, this judgment

[4]In Coq, we model sets with elements of type A as functions from A to propositions, thus Q_1 is represented as a function that takes a value and a state and returns a proposition, Q' is a function that takes a value, a state, another value, another state and returns a proposition, and the union over the family of results is written $\lambda\, v''\, s''.\ \exists\, v'\, s'.\ Q_1\, v'\, s' \,\wedge\, Q'\, v'\, s'\, v''\, s''$.

rules out the possibility for an execution of t/s to get stuck, and it can be used to express type safety, as detailed in §4. The judgment $t/s \Downarrow^{\text{co}} Q$ can also be used to define partial-correctness Hoare triples, as detailed in §5.

Formally, we can relate the meaning of $t/s \Downarrow^{\text{co}} Q$ 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 reduced further. Below, $t/s \longrightarrow t'/s'$ denotes the standard small-step evaluation judgment (defined in Appendix G), and val denotes the constructor that injects values into the grammar of terms.

CO-OMNI-BIG-IFF-SAFE-AND-CORRECT

$$t/s \Downarrow^{\text{co}} Q \quad\Longleftrightarrow\quad \forall s't'.\ (t/s \longrightarrow^{*} t'/s') \Rightarrow \begin{array}{l}\big(\exists v.\ t' = \mathsf{val}\,v \wedge (v, s') \in Q\big)\\ \vee\big(\exists t''s''.\ t'/s' \longrightarrow t''/s''\big)\end{array}$$

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

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

2.5 The Bind Rule for Reasoning about Evaluation Contexts

In this section, we explain how to reason about programs that are not in A-normal form. We follow the approach of the *bind rule*, popularized by Iris [Jung et al. 2018] in the context of program logics. The bind rule follows the pattern of the let-binding rule but allows for evaluation of a subterm t that appears in an *evaluation context* E. For the syntax introduced in §2.1, we can define evaluation contexts by the following grammar, where $\square$ denotes the *hole*, i.e., the empty context.

$$E \quad := \quad \square \quad|\quad \mathsf{let}\,x = E\,\mathsf{in}\,t \quad|\quad (E\,t) \quad|\quad (v\,E) \quad|\quad \mathsf{if}\,E\,\mathsf{then}\,t\,\mathsf{else}\,t$$

We write $E[t]$ for the context E whose hole is filled with the term t. We write value t for the predicate that asserts that t is a value. The bind rule describes how to evaluate or reason about subterms that appear in evaluation contexts and that are not already values. The omni-big-step bind rule takes the following form.

$$\frac{\neg\,\mathsf{value}\,t \qquad t/s \Downarrow Q_1 \qquad \big(\forall v s'.\ Q_1\,v\,s' \Rightarrow E[v]\,/\,s' \Downarrow Q\big)}{E[t]\,/\,s \Downarrow Q}\ \textsc{omni-big-bind}$$

The premise $\neg$ value t could be omitted for the inductive interpretation of the omni-big-step rules. It is required, however, for the coinductive interpretation, to prevent the construction of infinite derivations for terms that do not diverge.

3 OMNI-SMALL-STEP SEMANTICS

In this section, we present the omni-small-step judgment, written $t/s \longrightarrow P$. Here, P denotes a set of pairs each made of a term and a state. We then present the *eventually* judgment, written $t/s \longrightarrow^{\Diamond} P$. We use these judgments in particular for establishing type-safety (§4.1) and compiler-verification (§6.6) results.

3.1 The Omni-Small-Step Judgment

The omni-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. 4. There is one per small-step transition. The interesting rules are

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

$$\text{OMNI-SMALL-IF-TRUE}\quad \frac{P\,(t_1, s)}{(\text{if true then } t_1 \text{ else } t_2)/s \longrightarrow P} \qquad \text{OMNI-SMALL-IF-FALSE}\quad \frac{P\,(t_2, s)}{(\text{if false then } t_1 \text{ else } t_2)/s \longrightarrow P}$$

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

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

$$\text{OMNI-SMALL-ADD}\quad \frac{(n_1 + n_2,\ s) \in P}{(\text{add}\, n_1\, n_2)/s \longrightarrow P} \qquad \text{OMNI-SMALL-RAND}\quad \frac{n > 0 \qquad \Big(\forall m \in [0, n).\ \ (m, s) \in P\Big)}{(\text{rand}\, n)/s \longrightarrow P}$$

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

$$\text{OMNI-SMALL-FREE}\quad \frac{p \in \text{dom}\, s \qquad (tt,\ s \setminus p) \in P}{(\text{free}\, p)/s \longrightarrow P} \qquad \text{OMNI-SMALL-GET}\quad \frac{p \in \text{dom}\, s \qquad (s[p],\ s) \in P}{(\text{get}\, p)/s \longrightarrow P} \qquad \text{OMNI-SMALL-SET}\quad \frac{p \in \text{dom}\, s \qquad (tt,\ s[p := v]) \in P}{(\text{set}\, p\, v)/s \longrightarrow P}$$

Fig. 4. Omni-small-step semantics (for terms in A-normal form)

those involving nondeterminism, namely OMNI-SMALL-RAND and OMNI-SMALL-REF, which follow a pattern similar to the corresponding omni-big-step rules. Observe also how the rule OMNI-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.

$$\text{OMNI-SMALL-STEP-IFF-PROGRESS-AND-CORRECT}$$
$$t/s \longrightarrow P \quad\iff\quad \Big(\exists t's'.\ \ t/s \longrightarrow t'/s'\Big) \wedge \Big(\forall t's'.\ \ t/s \longrightarrow t'/s' \ \Rightarrow\ (t', s') \in P\Big)$$

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 omni-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.

$$\text{EVENTUALLY-HERE}\quad \frac{(t, s) \in P}{t/s \longrightarrow^{\Diamond} P} \qquad \text{EVENTUALLY-STEP}\quad \frac{t/s \longrightarrow P' \qquad \Big(\forall (t', s') \in P'.\ \ t'/s' \longrightarrow^{\Diamond} P\Big)}{t/s \longrightarrow^{\Diamond} P}$$

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 omni-big-step judgment $t/s \Downarrow Q$.

$$\text{EVENTUALLY-IFF-OMNI-BIG-STEP:} \qquad t/s \longrightarrow^{\Diamond} Q \quad\iff\quad t/s \Downarrow Q$$

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 omni-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.

$$\text{EVENTUALLY-STEP-CHAINED}: \qquad t/s \longrightarrow \left\{(t', s') \;\middle|\; t'/s' \longrightarrow^{\Diamond} P\right\} \quad \Rightarrow \quad t/s \longrightarrow^{\Diamond} P$$

One may wonder why we did not use this rule directly in the inductively defined judgment, and the reason is Coq's *strict positivity* requirement. The considerations for encoding sequencing here are similar to those discussed in Appendix A in the context of the omni-big-step let-binding rule.

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.

$$\text{EVENTUALLY-CUT}: \qquad t/s \longrightarrow^{\Diamond} P' \quad \wedge \quad \left(\forall (t', s') \in P'.\;\; t'/s' \longrightarrow^{\Diamond} P\right) \quad \Rightarrow \quad t/s \longrightarrow^{\Diamond} P$$

This cut rule also admits a *chained* version, which 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.

$$\text{EVENTUALLY-CUT-CHAINED}: \qquad t/s \longrightarrow^{\Diamond} \left\{(t', s') \;\middle|\; t'/s' \longrightarrow^{\Diamond} P\right\} \quad \Rightarrow \quad t/s \longrightarrow^{\Diamond} P$$

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 Omni-Small-Step Judgment

Let $t/s \longrightarrow^{\Diamond}_{\text{co}} P$ denote the coinductive interpretation of the two rules that define $t/s \longrightarrow^{\Diamond} P$. Divergence can be captured by instantiating P as the empty set. We prove that the judgment $t/s \longrightarrow^{\Diamond}_{\text{co}} \emptyset$ 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.

$$\begin{array}{l}\text{CO-EVENTUALLY-EMPTY-IFF-SMALL-STEP-DIVERGES} \\ t/s \longrightarrow^{\Diamond}_{\text{co}} \emptyset \qquad \Longleftrightarrow \qquad \forall s't'.\;\; (t/s \longrightarrow^{*} t'/s') \;\Rightarrow\; \left(\exists t''s''.\;\; t'/s' \longrightarrow t''/s''\right)\end{array}$$

Besides, we can relate the coinductive omni-small-step judgment $t/s \longrightarrow^{\Diamond}_{\text{co}} P$ to the coinductive omni-big-step judgment $t/s \Downarrow^{\text{co}} Q$ defined in §2.4. Here again, we let Q denote a set of *final* configurations. We prove the following equivalence.

$$\text{CO-EVENTUALLY-IFF-CO-OMNI-BIG-STEP}: \qquad t/s \longrightarrow^{\Diamond}_{\text{co}} Q \quad \Longleftrightarrow \quad t/s \Downarrow^{\text{co}} Q$$

The proofs of these two equivalences CO-EVENTUALLY-IFF-CO-OMNI-BIG-STEP, CO-EVENTUALLY-EMPTY-IFF-SMALL-STEP-DIVERGES, as well as the proof of CO-OMNI-BIG-IFF-SAFE-AND-CORRECT from §3.4, are interesting in that they involve yet another judgment. This judgment, written $t/s \twoheadrightarrow^{*}_{\text{co}} Q$, is defined in terms of the standard small-step semantics, by taking the coinductive interpretation of the following two rules.

$$\text{EVENTUALLY'-HERE}\;\; \frac{(v, s) \in Q}{v/s \twoheadrightarrow^{*}_{\text{co}} Q} \qquad \text{EVENTUALLY'-STEP}\;\; \frac{\left(\exists t's'.\;\; t/s \longrightarrow t'/s'\right) \qquad \left(\forall t's'.\;\; (t/s \longrightarrow t'/s') \;\Rightarrow\; (t'/s' \twoheadrightarrow^{*}_{\text{co}} Q)\right)}{t/s \twoheadrightarrow^{*}_{\text{co}} Q}$$

The desired equivalences are established in three steps. First, we prove that the standard small-step characterization of partial correctness that appears in the statement of CO-OMNI-BIG-IFF-SAFE-AND-CORRECT (§3.4) is equivalent to this new coinductive judgment $t/s \twoheadrightarrow^*_{\text{co}} Q$. The proof is relatively straightforward because both of these characterizations are expressed using small-step transitions.

Second, we prove that the *co-eventually* judgment $t/s \longrightarrow^{\Diamond}_{\text{co}} Q$ is equivalent to $t/s \twoheadrightarrow^*_{\text{co}} Q$. The proof is relatively straightforward because the coinductive definitions for these two judgments share a similar structure. As a corollary, by instantiating Q as the empty set, we establish CO-EVENTUALLY-EMPTY-IFF-SMALL-STEP-DIVERGES.

Third, we prove that the *co-omni-big-step* judgment $t/s \Downarrow^{\text{co}} Q$ is equivalent to $t/s \twoheadrightarrow^*_{\text{co}} Q$. This third proof is the most challenging, especially for establishing the implication from the small-step style judgment to the big-step style judgment. The proof involves a key intermediate lemma, which consists of an inversion rule for let-bindings: if $(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)/s \twoheadrightarrow^*_{\text{co}} Q$ holds, then there exists a set Q_1 such that $t_1/s \twoheadrightarrow^*_{\text{co}} Q_1$ and $\forall (v_1, s') \in Q_1.\ ([v_1/x]\, t_2)/s' \twoheadrightarrow^*_{\text{co}} Q$ hold. The proof of this key lemma itself relies on two auxiliary results, whose purpose is to justify that we can take as witness for Q_1 the strongest postcondition of t_1/s. The first one asserts that $(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)/s \twoheadrightarrow^*_{\text{co}} Q$ implies $t_1/s \twoheadrightarrow^*_{\text{co}} \{(v_1, s') \mid t_1/s \longrightarrow^* v_1/s'\}$. The second one asserts that $(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)/s \twoheadrightarrow^*_{\text{co}} Q$ and $t_1/s \longrightarrow^* v_1/s'$ imply $([v_1/x]\, t_2)/s' \twoheadrightarrow^*_{\text{co}} Q$. We refer to our Coq development for details.

A key observation about all the proofs involved in §2 and §3 is that they are constructive[5]. In particular, we are able to establish equivalences betweeen *coinductive omni-big-step semantics* and small-step style semantics *without* recourse to classical logic. This constrast with *coinductive big-step semantics* [Leroy and Grall 2009], whose connection to small-step semantics requires classical logic. We discuss this aspect further in the related work section (§8).

4 TYPE-SAFETY PROOFS USING OMNISEMANTICS

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

For this section, we need to consider a different semantics for the random-number generator. Indeed, the current rule OMNI-BIG-RAND asserts that the program is stuck if $\mathsf{rand}\, n$ 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 $\mathsf{rand}\, n$ is turned into a total function that returns 0 when $n \le 0$.

OMNI-BIG-RAND-COMPLETE

$$\frac{\forall m.\ \ 0 \le m < \max(n, 1) \ \Rightarrow\ (m, s) \in Q}{(\mathsf{rand}\, n)/s \Downarrow Q}$$

OMNI-SMALL-RAND-COMPLETE

$$\frac{\forall m.\ \ 0 \le m < \max(n, 1) \ \Rightarrow\ (m, s) \in P}{(\mathsf{rand}\, n)/s \longrightarrow P}$$

Additionally, 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 are essentially

[5] The proofs that we present do not exploit classical logic axioms. However, we do not provide a *machine-checked proof* that our proofs are constructive. Indeed, our Coq development is building on top of general-purpose libraries that exploit classical logic in various places.

standard, apart from the fact that they involve side conditions of the form $t \in \mathbb{V}$ to constrain terms to be in A-normal form. We include here two example rules; the other rules are given in appendix E.

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

4.1 Omni-Small-Step Type-Safety Proof for a State-Free Language

A *stuck term* is a term that is not a value and that cannot take a step. Type safety 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 can further reduce.

TYPE-SAFETY (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 safety is by proving the *preservation* and *progress* properties [Pierce 2002; Wright and Felleisen 1994].

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

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

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., of 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-size proof term. This combined statement, shown below, asserts that a well-typed term t is either a value or can make a step towards a term t' that admits the same type.

INDUCTION-FOR-TYPE-SAFETY, STATE-FREE, STANDARD SMALL-STEP, DETERMINISTIC

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

As we explain next, this approach can be generalized to the case of nondeterministic languages using the omni-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 safety by considering for the postcondition P the set of terms t' that admit the same type as t.

LEMMA 4.1 (INDUCTION-FOR-TYPE-SAFETY, STATE-FREE, OMNI-SMALL-STEP, NONDETERMINISTIC).

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

PROOF. 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. We next detail two representative proof cases.

CASE 1: the term t has been typed using rule TYP-RAND. In this case, the term t has the form "$\mathsf{rand}\, t_1$". The rule concludes $\emptyset \vdash (\mathsf{rand}\, t_1) : \mathsf{int}$, from the premise $\emptyset \vdash t_1 : \mathsf{int}$ and the premise $t_1 \in \mathbb{V}$. The latter means that t_1 is either a value or a variable (recall that we assume A-normal form to simplify the presentation). Because t_1 typechecks in the empty environment, it cannot be a variable. Thus, it must be a value, and because this value has type int, it must be an integer value. (In other words, $\emptyset \vdash t_1 : \mathsf{int}$ must have been derived using the rules TYP-VAL and VTYP-INT stated in appendix E.) Let us call n this integer. We need to establish: $(\mathsf{rand}\, n) \longrightarrow \left\{t' \;\middle|\; (\emptyset \vdash t' : \mathsf{int})\right\}$. Recall the rule OMNI-SMALL-RAND-COMPLETE introduced at the start of §4. We apply this rule (ignoring the state component), and need to establish its premise: $\forall m.\ 0 \le m < \max(n, 1) \Rightarrow m \in \left\{t' \;\middle|\; (\emptyset \vdash t' : \mathsf{int})\right\}$. Consider an integer m such that $0 \le m < \max(n, 1)$. We are left to prove $\emptyset \vdash m : \mathsf{int}$, which is derivable from the rules TYP-VAL and VTYP-INT.

CASE 2: the term t has been typed using rule TYP-LET. In this case, the term t has the form "$\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$". The rule concludes $\emptyset \vdash (\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2) : T$, from the two premises $\emptyset \vdash t_1 : T_1$ and $x : T_1 \vdash t_2 : T$. We need to prove $(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2) \longrightarrow \left\{t' \;\middle|\; (\emptyset \vdash t' : T)\right\}$. By the induction hypothesis applied to the first assumption, either t_1 is a value, or $t_1 \longrightarrow \left\{t_1' \;\middle|\; (\emptyset \vdash t_1' : T_1)\right\}$.

In the first subcase, t_1 is a value; let us call it v_1. We exploit OMNI-SMALL-LET, and are left to justify $([v_1/x]\, t_2) \in \left\{t' \;\middle|\; (\emptyset \vdash t' : T)\right\}$, that is, $\emptyset \vdash ([v_1/x]\, t_2) : T$. This result follows from the standard substitution lemma applied to $x : T_1 \vdash t_2 : T$ and to $\emptyset \vdash v_1 : T_1$.

In the second subcase, we have $t_1 \longrightarrow \left\{t_1' \;\middle|\; (\emptyset \vdash t_1' : T_1)\right\}$. To prove $(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2) \longrightarrow \left\{t' \;\middle|\; (\emptyset \vdash t' : T)\right\}$, we exploit OMNI-SMALL-LET-CTX with $P_1 = \left\{t_1' \;\middle|\; (\emptyset \vdash t_1' : T_1)\right\}$. We need to justify the second premise of that rule: $\forall t_1' \in P_1.\ (\mathsf{let}\, x = t_1' \,\mathsf{in}\, t_2) \in \left\{t' \;\middle|\; (\emptyset \vdash t' : T)\right\}$. Consider a particular t_1'. The assumption $t_1' \in P_1$ is equivalent to $\emptyset \vdash t_1' : T_1$. The proof obligation $(\mathsf{let}\, x = t_1' \,\mathsf{in}\, t_2) \in \left\{t' \;\middle|\; (\emptyset \vdash t' : T)\right\}$ is equivalent to $\emptyset \vdash (\mathsf{let}\, x = t_1' \,\mathsf{in}\, t_2) : T$. This result follows from the rule TYP-LET applied to the facts $\emptyset \vdash t_1' : T_1$ and $x : T_1 \vdash t_2 : T$. □

The statement INDUCTION-FOR-TYPE-SAFETY 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-SAFETY entails the TYPE-SAFETY property.[6]

4.2 Omni-Small-Step Type-Safety 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 rules for typing memory locations and memory operations are standard; they appear in Appendix F.

The type-safety 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-SAFETY:

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

To establish a type-safety 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:

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

[6]The generic entailment from INDUCTION-FOR-TYPE-SAFETY to TYPE-SAFETY holds for any typing judgment of the form $\emptyset \vdash t : T$ 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 OMNI-SMALL-STEP-IFF-PROGRESS-AND-CORRECT from §3.2.

The preservation and progress lemmas associated with the traditional approach to proving type safety 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 ($S' \supseteq S$).

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

In contrast, using the omni-small-step judgment, we can establish type safety 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-safety (omni-small-step, with state)

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

4.3 Omni-Big-Step Type-Safety Proof for an Imperative Language

Traditionally, a big-step safety proof can only be carried out if the semantics is completed using error-propagation rules. Here, we demonstrate how to establish type safety with respect to an omni-big-step judgment, without any need for error-propagation rules. First, we introduce the construct $[\![T/S]\!]$ to denote the set of possible outputs produced by a term of type T, well-typed in a store of type S. Second, we describe the statement and proof for type safety.

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'. The extension S' involved here accounts for the fact that the evaluation of a term t of type T may perform allocation operations that extend the store in which t is well-typed.

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

A key lemma involved in the type soundness proof asserts that, if S' is a store typing that enforces more constraints than another store typing S, then $[\![T/S']\!]$ is a smaller set than $[\![T/S]\!]$.

Lemma 4.2 (configuration-typing-subset).

$$S' \supseteq S \quad \Rightarrow \quad [\![T/S']\!] \subseteq [\![T/S]\!]$$

Proof. Assume $S' \supseteq S$. Consider a pair $(v,s) \in [\![T/S']\!]$. By definition, there exists S'' such that $S'' \supseteq S'$ and $\vdash s : S''$ and $S'' \vdash v : T$. By transitivity, $S'' \supseteq S$. We conclude that $(v,s) \in [\![T/S]\!]$ holds, by taking S'' as witness for the existential quantifier in the definition of $[\![T/S]\!]$. □

We are now ready to state type safety. The coinductive omni-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 $t/\varnothing \Downarrow^{\mathrm{co}} [\![T/\varnothing]\!]$ thus captures exactly the type-safety property associated with the typing judgment $\varnothing;\emptyset \vdash t : T$. Type safety may be established by proving the following statement by coinduction.

Lemma 4.3 (coinduction-for-type-safety, omni-big-step, nondeterministic).

$$S;\emptyset \vdash t : T \quad \wedge \quad \vdash s : S \quad \Rightarrow \quad t/s \Downarrow^{co} [\![T/S]\!]$$

Proof. For technical reasons, the Coq coinduction tactic needs to be applied to the following statement, which introduces an intermediate set Q.

$$S;\emptyset \vdash t : T \quad \wedge \quad \vdash s : S \quad \wedge \quad [\![T/S]\!] \subseteq Q \quad \Rightarrow \quad t/s \Downarrow^{\text{co}} Q$$

Observe that this alternative statement is logically equivalent to the previous one: on the one hand, we can instantiate Q as $[\![T/S]\!]$; on the other hand, we can exploit omni-big-conseqence to prove $t/s \Downarrow^{\text{co}} Q$ from $t/s \Downarrow^{\text{co}} [\![T/S]\!]$ and $[\![T/S]\!] \subseteq Q$.

We carry out a proof by coinduction on that alternative statement. The coinduction hypothesis asserts that we can *assume* the alternative statement to hold, provided that we have already applied at least one evaluation rule (i.e., a *coinductive constructor*) to the conclusion at hand ($t/s \Downarrow^{\text{co}} Q$).

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. Let us consider two representative proof cases: the case of rand and the case of a let-binding. In each case, the assumptions are $S;\emptyset \vdash t : T$ and $\vdash s : S$ and $[\![T/S]\!] \subseteq Q$; and the goal is to prove $t/s \Downarrow^{\text{co}} Q$.

Case 1: the term t has been typed using rule typ-rand. In this case, the term t has the form "$\mathsf{rand}\, t_1$", and T is int. The rule concludes $S;\emptyset \vdash (\mathsf{rand}\, t_1) : \mathsf{int}$, from the premise $S;\emptyset \vdash t_1 : \mathsf{int}$ and the premise $t_1 \in \mathbb{V}$. Because t_1 typechecks in the empty environment, it must be a value. Because this value has type int, it must be an integer value, let us call it n. We need to establish: $(\mathsf{rand}\, n)/s \Downarrow^{\text{co}} Q$. We apply the rule co-omni-big-rand-complete, which is like omni-big-rand-complete but part of the coinductive interpretation of the set of evaluation rules. We need to prove its premise: $\forall m.\ 0 \leq m < \max(n, 1) \Rightarrow (m, s) \in Q$. Consider a particular m in that range. We have $[\![\mathsf{int}/S]\!] \subseteq Q$. Thus, to show $(m, s) \in Q$ it suffices to show $(m, s) \in [\![\mathsf{int}/S]\!]$. By definition of the operator $[\![T/S]\!]$, this amounts to proving $\exists S' \supseteq S.\ (\vdash s : S') \wedge (S' \vdash m : \mathsf{int})$. We conclude by taking $S' = S$ and checking that $\vdash s : S$ and $S' \vdash m : \mathsf{int}$ indeed hold.

Case 2: the term t has been typed using rule typ-let. In this case, the term t has the form "$\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$". The rule concludes $S;\emptyset \vdash (\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2) : T$, from the two premises $S;\emptyset \vdash t_1 : T_1$ and $S;(x : T_1) \vdash t_2 : T$. We need to establish: $(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)/s \Downarrow^{\text{co}} Q$. We apply the rule co-omni-big-let (which is like omni-big-let but part of the coinductive interpretation of the set of evaluation rules) with Q_1 instantiated as $[\![T_1/S]\!]$. We have to establish the two premises: $t_1/s \Downarrow [\![T_1/S]\!]$, and $\forall (v', s') \in [\![T_1/S]\!].\ ([v'/x]\, t_2)/s' \Downarrow Q$. The first premise follows directly from the coinduction hypothesis applied to $S;\emptyset \vdash t_1 : T_1$ and to $[\![T_1/S]\!] \subseteq [\![T_1/S]\!]$. For the second premise, consider a pair $(v', s') \in [\![T_1/S]\!]$. This amounts to assuming the existence of some S' such that $S' \supseteq S$ and $\vdash s' : S'$ and $S' \vdash v : T_1$. There remains to show $([v'/x]\, t_2)/s' \Downarrow Q$. A standard "type preservation upon store typing extension" lemma shows that, because $S' \supseteq S$, we can refine $S\,;(x : T_1) \vdash t_2 : T$ into $S'\,;(x : T_1) \vdash t_2 : T$. Then, by the standard substitution lemma applied to $S'\,;(x : T_1) \vdash t_2 : T$ and to $S' \vdash v : T_1$, we derive $S';\emptyset \vdash ([v'/x]\, t_2) : T$. Besides, the lemma configuration-typing-subset applied to $S' \supseteq S$ gives $[\![T/S']\!] \subseteq [\![T/S]\!]$. Composing this subset relation by transitivity with $[\![T/S]\!] \subseteq Q$ yields $[\![T/S']\!] \subseteq Q$. The conclusion $([v'/x]\, t_2)/s' \Downarrow Q$ then follows from the coinduction hypothesis applied to $S';\emptyset \vdash ([v'/x]\, t_2) : T$ and $\vdash s' : S'$ and $[\![T/S']\!] \subseteq Q$.

Note that most of these arguments are easily handled by automated proof search in Coq. □

Like for the small-step settings, we proved once-and-for-all that the statement coinduction-for-type-safety entails type-safety.

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

Regarding the comparison with the standard preservation-and-progress approach, at this stage we cannot draw general conclusions on whether omni-big-step and omni-small-step type-safety 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 PROGRAM PROOF RULES

This section discusses the construction of a *foundational* program logic, that is, a program logic whose reasoning rules are derived through mechanized proofs from the formal semantics of the targeted programming language. We specifically focus on Separation Logic [O'Hearn et al. 2001; Reynolds 2002], which has proved in the past two decades to be an invaluable tool for carrying out practical, modular program verification, both for low-level and high-level languages—see the broad survey by O'Hearn [2019] and the survey by Charguéraud [2020] that focuses on sequential programs.

We first review the properties that a program logic might capture, and we describe the key challenges in deriving a foundational Separation Logic that captures total correctness with respect to a standard big-step semantics (§5.1). We then explain how omnisemantics overcome these challenges, allowing one to derive a foundational, total-correctness Separation Logic judgment in a straightforward, direct manner (§5.2). Moreover, by referring to the coinductive omni-big-step judgment instead of the inductive one, one can similarly define partial-correctness triples. We explain how to derive the reasoning rules (§5.3) and in particular the frame rule of Separation Logic (§5.4). Finally, we present reasoning rules in weakest-precondition style (§5.5), which have proved very useful to set up practical tools, and which turn out to be even easier to derive.

5.1 Challenges in Defining Foundational Separation Logic Triples

A 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, correctness, and termination hold.

When targeting total correctness, one key challenge in defining triples with respect to a standard big-step semantics is that the direct definition of Hoare triples yields a judgment that does not satisfy the *frame rule* of Separation Logic. The frame rule asserts that if a triple $\{H\}\ t\ \{Q\}$ holds, then the pre- and the postcondition may be extended with an arbitrary predicate H', yielding the valid triple $\{H \star H'\}\ t\ \{Q \star H'\}$. Here, $Q \star H$ denotes the postcondition $\lambda v.\ (Q\,v \star H)$.

Concretely, consider the following definition of a Hoare triple with respect to a standard big-step, deterministic semantics. It asserts that, for any input state s satisfying the precondition H, there exists a result value v and a final state s' such that the configuration t/s evaluates to a final configuration v/s' that satisfies the postcondition Q.

$$^{\mathsf{Hoare}}\{H\}\,t\,\{Q\} \quad \equiv \quad \forall s.\ H\,s \;\Rightarrow\; \exists v.\,\exists s'.\ (t/s \Downarrow v/s') \;\wedge\; (Q\,v\,s').$$

For such a judgment, one can prove that, starting from an empty heap, the allocation of a reference returns a *specific* memory location, say 0. For example, if the reference contains 3 and the location l denotes its address, one can prove: ${}^{\text{Hoare}}\,\{[\,]\}\,(\mathsf{ref}\,3)\;\{\lambda l.\;\;[l = 0] \star (0 \hookrightarrow 3)\}$. To see why the judgment does not satisfy the frame rule, let us attempt to extend the pre- and the postcondition of this triple with the heap predicate $(0 \hookrightarrow 1)$, which denotes a reference at location 0 storing the value 1. We obtain: ${}^{\text{Hoare}}\,\{0 \hookrightarrow 1\}\,(\mathsf{ref}\,3)\;\{\lambda l.\;\;[l = 0] \star (0 \hookrightarrow 3) \star (0 \hookrightarrow 1)\}$. This triple does not hold, because the separating conjunction $(0 \hookrightarrow 3) \star (0 \hookrightarrow 1)$ is equivalent to False.

To derive a Separation Logic judgment that *does* satisfy the frame rule, one can exploit the classic technique of the *baked-in frame rule* [Birkedal et al. 2005]—for technical and historical details, we refer to Charguéraud [2020, §5.1 and §10.2]. Separation Logic triples are defined as follows.

$$^{\text{Sep. Logic via baked-in frame rule}}\,\{H\}\,t\,\{Q\} \quad\equiv\quad \forall H'.\;\; ^{\text{Hoare}}\,\{H \star H'\}\,t\,\{Q \star H'\}$$

This definition quantifies over a heap predicate H' that describes the "rest of the world." The resulting triples inherently satisfy the frame rule, as a direct consequence of the associativity of the separating-conjunction operator. Indirectly, the introduction of H' rules out the judgments whose postconditions refer to *specific* locations, such as in the aforementioned counterexample.

The two-stage construction presented above, for building Separation Logic triples on top of the standard big-step judgment via the *baked-in frame rule* technique, can be applied to deterministic languages or to languages that are deterministic up to the choice of memory addresses. In what follows, we show that, by grounding Separation Triples not on top of standard big-step semantics but instead on top of omnisemantics, we can avoid the need to go through the two-stage construction associated with the *baked-in frame rule* technique. Moreover, the omnisemantics construction applies to the general case of *nondeterministic* semantics, and it unfolds similarly for both total- and partial-correctness triples.

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

Consider a possibly nondeterministic semantics. A total-correctness Hoare triple $\{H\}\;t\;\{Q\}$ asserts that, for any input state s satisfying the precondition H, every possible execution of t/s terminates and satisfies the postcondition Q. This property can be captured using the *inductive* omni-big-step judgment as follows:

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

Note that an omni-big-step judgment may be interpreted as a particular Hoare triple, featuring a singleton precondition to constrain the input state:

$$\big(t/s \Downarrow Q\big) \qquad\Longleftrightarrow\qquad ^{\text{Hoare}}\,\{(\lambda s'.\;\; s' = s)\}\,t\,\{Q\}.$$

A partial-correctness Hoare triple asserts that, for any input state s satisfying the precondition H, every possible execution of t/s either diverges or terminates and satisfies the postcondition. This property can be captured using the *coinductive* omni-big-step judgment as follows:

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

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.

Throughout the rest of this section, we present results for total correctness. We write $\{H\}\;t\;\{Q\}$ for the definition of ${}^{\text{Hoare}}\,\{H\}\;t\;\{Q\}$ given above. As we show, these triples inherently satisfy the frame rule, hence yield a Separation Logic. All the corresponding results for partial correctness hold and may be found in our Coq formalization.

5.3 Deriving Reasoning Rules for Hoare Triples

In a foundational program logic, reasoning rules take the form of lemmas proved correct with respect to the definition of triples and with respect to the semantics of the language. Consider for example the case of a let-binding. Let us compare the semantics rule OMNI-BIG-LET with the Hoare-logic rule HOARE-LET, which are shown below. 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. We also present reasoning rules using the horizontal bar, but keep in mind that the statements are not inductive definitions but lemmas.

OMNI-BIG-LET

$$\frac{t_1/s \Downarrow Q_1 \qquad \left(\forall v' s'.\ \ Q_1\, v'\, s' \ \Rightarrow\ ([v'/x]\, t_2)/s' \Downarrow Q\right)}{(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)/s \Downarrow Q}$$

HOARE-LET

$$\frac{\{H\}\ t_1\ \{Q_1\} \qquad \left(\forall v'.\ \ \{Q_1\, v'\}\ ([v'/x]\, t_2)\ \{Q\}\right)}{\{H\}\ (\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)\ \{Q\}}$$

The only difference between OMNI-BIG-LET and HOARE-LET is that the first rule considers one specific state s, whereas the second rule 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 OMNI-BIG-LET. The corresponding Coq proof script witnesses the simplicity of the proof: "`intros. eapply mbig_let; eauto.`"

Likewise, we derive a version of the bind rule, which generalizes the let-binding rule (recall §2.5). For the reasoning rule, shown below, we purposely do not include the premise $\neg$ value t.

OMNI-BIG-BIND

$$\frac{\neg\, \mathsf{value}\, t \qquad t/s \Downarrow Q_1 \qquad \left(\forall v s'.\ \ Q_1\, v\, s' \ \Rightarrow\ E[v]\,/\,s' \Downarrow Q\right)}{E[t]\,/\,s \Downarrow Q}$$

HOARE-BIND

$$\frac{\{H\}\ t\ \{Q_1\} \qquad \left(\forall v.\ \ \{Q_1\, v\}\ E[v]\ \{Q\}\right)}{\{H\}\ E[t]\ \{Q\}}$$

As another example, consider the consequence rule. The Hoare-logic rule is, again, an immediate consequence of the omni-big-step rule.

OMNI-BIG-CONSEQUENCE

$$\frac{t/s \Downarrow Q \qquad Q \subseteq Q'}{t/s \Downarrow Q'}$$

HOARE-CONSEQUENCE

$$\frac{H' \subseteq H \qquad \{H\}\ t\ \{Q\} \qquad Q \subseteq Q'}{\{H'\}\ t\ \{Q'\}}$$

5.4 Deriving The Frame Rule of Separation Logic

We next explain how to derive the frame rule for total-correctness triples. To that end, we first need to state and prove a key lemma capturing the preservation of the omni-big-step judgment $t/s_1 \Downarrow Q$ when the input state s_1 is augmented with a disjoint piece of state s_2. We write $s_1 \perp s_2$ to assert that s_1 and s_2 have disjoint domains.

LEMMA 5.1 (FRAME PROPERTY FOR BIG-STEP OMNISEMANTICS).

$$\frac{t/s_1 \Downarrow Q \qquad s_1 \perp s_2}{t/(s_1 \uplus s_2) \Downarrow (Q \star (\lambda s'.\ s' = s_2))}\ \text{OMNI-BIG-FRAME}$$

PROOF. The proof is carried out by induction on the omnisemantics judgment. There are two interesting cases in the proof: the treatment of an allocation (4 lines of Coq script) and that of a let-binding (3 lines of Coq script). In each case, we assume $s_1 \perp s_2$.

CASE 1: t is $\mathsf{ref}\, v$. The assumption is $(\mathsf{ref}\, v)/s_1 \Downarrow Q$. It is derived by the rule OMNI-BIG-REF, whose premise is $\forall p \notin \mathsf{dom}\, s_1.\ Q\, p\, (s_1[p := v])$. We need to prove $(\mathsf{ref}\, v)/(s_1 \uplus s_2) \Downarrow (Q \star (\lambda s'.\ s' = s_2))$. By OMNI-BIG-REF, we need to justify: $\forall p \notin \mathsf{dom}\, (s_1 \uplus s_2).\ (Q \star (\lambda s'.\ s' = s_2))\, p\, ((s_1 \uplus s_2)[p := v])$. Consider a location p not in $\mathsf{dom}\, s_1$ nor in $\mathsf{dom}\, s_2$. The predicate $(Q \star (\lambda s'.\ s' = s_2))\, p$ is equivalent to $(Q\, p) \star (\lambda s'.\ s' = s_2)$. The state update $(s_1 \uplus s_2)[p := v]$ is equivalent to $(s_1[p := v]) \uplus s_2$. Thus, there remains to prove: $((Q\, p) \star (\lambda s'.\ s' = s_2))\ ((s_1[p := v]) \uplus s_2)$. By definition of separating conjunction and exploiting $(s_1[p := v]) \perp s_2$, it suffices to show $Q\,p\,(s_1[p := v])$. This fact follows from $\forall p \notin \mathrm{dom}\,s_1.\ Q\,p\,(s_1[p := v])$.

Case 2: t is "$\mathsf{let}\,x\ =\ t_1\,\mathsf{in}\,t_2$". The assumption is $t/s_1\ \Downarrow\ Q$. It is derived by the rule omni-big-let, whose premises are $t_1/s_1\ \Downarrow\ Q_1$ and $\forall v's'.\ Q_1\,v'\,s'\ \Rightarrow\ ([v'/x]\,t_2)/s'\ \Downarrow\ Q$. We need to prove $(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)/(s_1 \uplus s_2)\ \Downarrow\ (Q \star (\lambda s'.\ s' = s_2))$. To that end, we invoke omni-big-let. For its first premise, we prove $t_1/(s_1 \uplus s_2)\ \Downarrow\ (Q_1 \star (\lambda s'.\ s' = s_2))$ by exploiting the induction hypothesis applied to $t_1/s_1\ \Downarrow\ Q_1$. For the second premise, we have to prove $\forall v's''.\ (Q_1 \star (\lambda s'.\ s' = s_2))\,v'\,s'' \Rightarrow ([v'/x]\,t_2)/s''\ \Downarrow\ (Q \star (\lambda s'.\ s' = s_2))$. Consider a particular v' and s''. The assumption $(Q_1 \star (\lambda s'.\ s' = s_2))\,v'\,s''$ is equivalent to $((Q_1\,v') \star (\lambda s'.\ s' = s_2))\,s''$. By definition of separating conjunction, we deduce that s'' decomposes as $s_1' \uplus s_2$, with $s_1' \perp s_2$ and $Q_1\,v'\,s_1'$, for some s_1'. There remains to prove $([v'/x]\,t_2)/(s_1' \uplus s_2)\ \Downarrow\ (Q \star (\lambda s'.\ s' = s_2))$. We first exploit $\forall v's'.\ Q_1\,v'\,s'\ \Rightarrow\ ([v'/x]\,t_2)/s'\ \Downarrow\ Q$, on $Q_1\,v'\,s_1'$ to obtain $([v'/x]\,t_2)/s_1'\ \Downarrow\ Q$. We then conclude by applying the induction hypothesis to the latter judgment. □

Lemma 5.2 (Frame rule).

$$\frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q \star H'\}}\ \textsc{frame} \qquad \text{where } Q \star H\ \equiv\ \lambda v.\,(Q\,v \star H)$$

Proof. Assume $\{H\}\ t\ \{Q\}$. Recall from §5.2 that this judgment is defined as $\forall s.\ H\,s \Rightarrow (t/s\ \Downarrow\ Q)$. We have to prove $\{H \star H'\}\ t\ \{Q \star H'\}$, that is, $\forall s.\ (H \star H')\,s\ \Rightarrow\ (t/s\ \Downarrow\ (Q \star H'))$. Consider a particular s such that $(H \star H')\,s$. By definition of separating conjunction, we can deduce that the input state s decomposes as $s_1 \uplus s_2$, with $s_1 \perp s_2$ and $H\,s_1$ and $H'\,s_2$. The goal is to prove: $t/(s_1 \uplus s_2)\ \Downarrow\ (Q \star H')$. By exploiting $\forall s.\ H\,s \Rightarrow (t/s\ \Downarrow\ Q)$ on $H\,s_1$, we derive $t/s_1\ \Downarrow\ Q$. By invoking the lemma omni-big-frame on this judgment and on $s_1\ \perp\ s_2$, we derive $t/(s_1 \uplus s_2)\ \Downarrow\ (Q \star (\lambda s'.\ s' = s_2))$. From there, to obtain the conclusion $t/(s_1 \uplus s_2)\ \Downarrow\ (Q \star H')$, it suffices to exploit the consequence rule omni-big-consequence, and justify that $(\lambda s'.\ s' = s_2)$ entails H'. In other words, we need to show that for any state s' being equal to s_2, this state s' does satisfy H'. Indeed, $H'\,s_2$ holds. (The Coq proof script for this lemma is 4 lines long.) □

In summary, the above proofs establish the frame property for the omni-big-step semantics, and for the triples that we build on top of that semantics. Those results hold for the imperative λ-calculus that we have considered in this paper. We believe that these results could be similarly established for other programming languages for which a Separation Logic can be set up. For example, we proved that the frame property holds for the omni-big-step semantics involved in the case study presented in §6.3.

5.5 Deriving Weakest-Precondition-Style Reasoning Rules

The weakest-precondition operator, written $\mathsf{wp}\,t\,Q$, computes the weakest predicate H for which the triple $\{H\}\ t\ \{Q\}$ holds. Here, "weakest" is interpreted w.r.t. the entailment relation, written $H \vdash H'$ and defined as pointwise predicate implication ($\forall s.\,H\,s \Rightarrow H\,s'$). Weakest reasoning rules are expressed as entailments. See, e.g., the rule for let-bindings and the bind rule shown below.

$$\textsc{wp-let}\quad \frac{}{\mathsf{wp}\,t_1\left(\lambda v'.\,\mathsf{wp}\,([v'/x]\,t_2)\,Q\right)\ \vdash\ \mathsf{wp}\,(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\,Q} \qquad \textsc{wp-bind}\quad \frac{}{\mathsf{wp}\,t\left(\lambda v.\,\mathsf{wp}\,(E[v])\,Q\right)\ \vdash\ \mathsf{wp}\,(E[t])\,Q}$$

Many proof tools simply axiomatize the weakest-precondition rules. In a *foundational* approach, however, one needs to prove the reasoning rules correct with respect to the formal semantics of the source language.

What is very appealing about describing the semantics of the language using an omni-big-step semantics is that it delivers the weakest-precondition-style reasoning rules almost for free. Indeed, the interpretation of the inductive judgment $t/s \Downarrow Q$ matches, up to the order of arguments, the standard interpretation of the weakest-precondition operator.

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

Thus, in a foundational approach, we can formally define wp as $\lambda tQs.\,(t/s \Downarrow Q)$.

There remains to describe how the weakest-precondition-style reasoning rules can be derived from the omni-big-step evaluation rules. Doing so is even easier than for deriving triples. Consider for example the semantics rule and the wp-reasoning rule associated with a let-binding.

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

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

The frame rule in weakest-precondition style follows directly from the OMNI-BIG-FRAME lemma established in the previous section. The rule appears below, together with a very handy corollary named the *ramified frame rule* [Hobor and Villard 2013; Krishnaswami et al. 2010]. In that corollary, the magic wand between postconditions, written $Q_1 \mathrel{-\!\!\star} Q_2$, is defined as $\forall v.\, Q_1\, v \mathrel{-\!\!\star} Q_2\, v$, where $\forall$ and $\star$ are the standard Separation Logic operators (see, e.g., [Charguéraud 2020, §3.2 and §7]).

$$\textsc{wp-frame}\quad \frac{}{(\mathsf{wp}\, t\, Q) \star H \;\vdash\; \mathsf{wp}\, t\, (Q \star H)} \qquad\qquad \textsc{wp-ramified-frame}\quad \frac{}{(\mathsf{wp}\, t\, Q) \;\star\; (Q \mathrel{-\!\!\star} Q') \;\vdash\; (\mathsf{wp}\, t\, Q')}$$

For most other term constructs, the wp rule is nothing but a copy of the omni-big-step rule with arguments reordered. One interesting exception is that of loops. "While" loops have not been discussed so far, but they appear in the language used for the case studies in §6. Typically, standard weakest-precondition rules for while loops are stated using loop invariants. In contrast, an omni-big-step rule 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 by induction, in just a few lines of proof.

In summary, by considering a semantics expressed in omni-big-step style, one can derive practical reasoning rules, both in Hoare-triple style and in weakest-precondition style, in most cases via one-line proofs. The construction of a program logic on top of an omni-big-step semantics 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. Besides, a major benefit of considering an omni-big-step semantics is that, unlike a set of weakest-precondition reasoning rules, it delivers an induction principle for reasoning about program executions. Such induction principles are exploited in the case studies (§6).

6 COMPILER-CORRECTNESS PROOFS FOR TERMINATING PROGRAMS

Omnisemantics also simplify some of the characteristic complexities of behavior-preservation proofs for compilers.

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, an exception, a fatal error, 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. Whether an error such as a division by zero is considered as a terminating behavior or as an undefined behavior is a design decision associated with each programming language. 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 in assembly language [Erbsen et al. 2021]. We leave to future work the application of omnisemantics 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.[7] Such a proof consists of showing that each step from the source program *corresponds to* a number of steps in the compiled program. The correspondence 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 source 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.[8] 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 modeling features of the intermediate language as deterministic even when it is not natural to do so, and even when doing so requires introducing artificial functions for, e.g., computing fresh memory locations in a deterministic manner. Even so, runtime input does not fit the fully deterministic model, leading to the technical definitions of receptiveness and determinacy (roughly, capturing the idea of determinism modulo input) so that lemmas for flipping forward simulations into backwards simulations can be stated and proven. Omnisemantics remove the need for this machinery.

In this section:

- We explain how omnisemantics sidestep the need for backward simulations, by carrying out forward-simulation proofs of compiler correctness, for nondeterministic terminating programs.

[7] We follow CompCert's terminology, using "forward" and "backward" to refer to the direction of compilation, "forward" meaning from source language to target language. We note the conflict with other literature [Lynch and Vaandrager 1995] that uses "forward" and "backward" to refer to the direction of the state transitions.

[8] 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 the idea generalizes to languages including I/O operations and to the case where the source language and 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 an omni-big-step semantics for the source language, whereas it features an omni-small-step semantics for the target language.

6.2 Establishing Correctness via Forward Simulations using Omnisemantics

Consider a compilation function written $C(t)$. For simplicity, we assume that the source and target language are identical, we assume that 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 omni-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:

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

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

$$\text{OMNI-FORWARD-PRESERVATION:} \qquad \forall Q. \qquad t \Downarrow Q \quad\Rightarrow\quad C(t) \Downarrow Q$$

Let us demonstrate the claim. Let us assume that terminates(t) hold. First of all, recall from §2.2 the equivalence named OMNI-BIG-STEP-IFF-TERMINATES-AND-CORRECT that relates the omni-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 OMNI-FORWARD-PRESERVATION assumption reformulates as follows.

$$\begin{array}{ll} \forall Q. & \Big(\mathsf{terminates}(t) \;\wedge\; \Big(\forall v.\ (t \Downarrow v) \;\Rightarrow\; v \in Q\Big)\Big) \\ & \;\Rightarrow\; \Big(\mathsf{terminates}(C(t)) \;\wedge\; \Big(\forall v.\ (C(t) \Downarrow v) \;\Rightarrow\; v \in Q\Big)\Big) \end{array}$$

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 Omnisemantics Simulations for 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 5 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

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

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

Fig. 5. Omni-big-step semantics with traces, selected rules

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 the pointer it returned, to allow for optimizations that reduce the amount of allocation. To our knowledge, this level of flexibility is unique to omnisemantics. 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 values, for the rest of this section. For languages of terms (that return values) rather than commands (that do not return values), we would need a representation relation between source-level and target-level values—we omit one 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

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

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 TraceInclusion as defined below holds.

$$\textsc{TraceInclusion}(c, s_{src}, s_{tgt}, \tau) \ := \ \mathsf{traces}(C(c), s_{tgt}, \tau) \subseteq \mathsf{traces}(c, s_{src}, \tau)$$

Assuming omni-big-step semantics $\Downarrow_{src}$ and $\Downarrow_{tgt}$ for the source and target languages, plus a relation R between source- and target-language states, we define *omnisemantics simulation*, a compiler-correctness criterion designed to be provable by induction on the $\Downarrow_{src}$ judgment, as follows:

$$\begin{aligned} \textsc{OmnisemanticsSimulation}(c) :=\ & \forall s_{src}\, s_{tgt}\, \tau\, Q.\ R(s_{src}, s_{tgt}) \ \wedge\ c/s_{src}/\tau \Downarrow_{src} Q \\ & \Longrightarrow C(c)/s_{tgt}/\tau \Downarrow_{tgt} Q_R \\ \text{where } Q_R(s'_{tgt},\ \tau') :=\ & \exists s'_{src}\,.\ R(s'_{src}, s'_{tgt}) \wedge Q(s'_{src}, \tau') \end{aligned}$$

Our goal in this section is to prove that an omnisemantics simulation implies trace inclusion if the source program terminates, i.e. to show

$$\begin{aligned} \forall c.\quad & \textsc{OmnisemanticsSimulation}(c) \implies \\ & \forall\ s_{src}\, s_{tgt}\, \tau.\ \mathsf{terminates}(c, s_{src}, \tau) \ \wedge\ R(s_{src}, s_{tgt}) \quad \Longrightarrow \quad \textsc{TraceInclusion}(c, s_{src}, s_{tgt}, \tau) \end{aligned}$$

We rely on two properties: First, soundness of omni-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 that terminates safely and whose traditional big-step executions all satisfy a postcondition also has an omnisemantics derivation:

$$\forall c\ s\ \tau\ Q.\ \mathsf{terminates}(c, s, \tau) \wedge (\forall s'\ \tau'.\ c/s/\tau \Downarrow s'/\tau' \implies Q(s', \tau')) \quad \implies \quad c/s/\tau \Downarrow Q \qquad (2)$$

To show trace inclusion, i.e. $\mathsf{traces}(C(c), s_{tgt}, \tau) \subseteq \mathsf{traces}(c, s_{src}, \tau)$, we can assume a target-language execution $C(c)/s_{tgt}/\tau \Downarrow s'_{tgt}/\tau'$ producing trace τ', and we need to show $\tau' \in \mathsf{traces}(c, s_{src}, \tau)$. 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 omnisemantics derivation $c/s_{src}/\tau \Downarrow (\lambda s'_{src}\ \tau'.\ c/s_{src}/\tau \Downarrow s'_{src}/\tau')$. Passing this fact into the omnisemantics 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 fact and the originally assumed target-level execution and obtain an s'_{src} such that $R(s'_{src}, s'_{tgt})$ and $c/s_{src}/\tau \Downarrow s'_{src}/\tau'$, which by definition is exactly what needs to hold to have $\tau' \in \mathsf{traces}(c, s_{src}, \tau)$.

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.

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

We actually consider two variants of this language, differing only in the types 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 functions 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 addition operators.

Omni-big-step semantics. For both languages, the omni-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 6. They are common to both languages—only the set of supported unary and binary operators differs. The semantics of operators are defined by two straightforward auxiliary relations (spelled out in Appendix H), $\mathsf{evalunop}(unop, v_1, v_2)$ asserting that applying *unop* to value v_1 results in v_2, and $\mathsf{evalbinop}(binop, v_1, v_2, v_3)$ asserting that applying *binop* to v_1 and v_2 results in v_3. 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

$$\textsc{eval-unop}\ \frac{(y, v_y) \in \ell \qquad \mathrm{evalunop}(op, v_y, v) \qquad Q(m,\ \ell[x := v],\ \tau)}{(x = op(y))/m/\ell/\tau \Downarrow Q}$$

$$\textsc{eval-store}\ \frac{(x, a) \in \ell \qquad (a+n) \in \operatorname{dom} m \qquad (y, v) \in \ell \qquad Q(m[(a+n) := v],\ \ell,\ \tau)}{(x[n] = y)/m/\ell/\tau \Downarrow Q}$$

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

$$\textsc{eval-alloc}\ \frac{\Big(\forall a\ \bar{v}.\ \mathsf{len}(\bar{v}) = n \ \wedge\ a, \ldots, (a+n-1) \notin \operatorname{dom} m \implies Q\,(m[(a, \ldots (a+n-1)) := \bar{v}],\ \ell[x := a],\ \tau)\Big)}{(x = \mathsf{alloc}(n))/m/\ell/\tau \Downarrow Q}$$

$$\textsc{eval-while-again}\ \frac{(x, 1) \in \ell \qquad c/m/\ell/\tau \Downarrow Q_1 \qquad \Big(\forall m'\ \ell'\ \tau'.\ Q_1(m', \ell', \tau') \implies (\mathsf{while}\ x\ \mathsf{do}\ c)/m'/\ell'/\tau' \Downarrow Q\Big)}{(\mathsf{while}\ x\ \mathsf{do}\ c)/m/\ell/\tau \Downarrow Q}$$

$$\textsc{eval-while-done}\ \frac{(x, 0) \in \ell \qquad Q(m,\ \ell,\ \tau)}{(\mathsf{while}\ x\ \mathsf{do}\ c)/m/\ell/\tau \Downarrow Q}$$

$$\textsc{eval-seq}\ \frac{c_1/m/\ell/\tau \Downarrow Q_1 \qquad (\forall m'\ \ell'\ \tau'.\ Q_1(m', \ell', \tau') \implies c_2/m'/\ell'/\tau' \Downarrow Q)}{(c_1; c_2)/m/\ell/\tau \Downarrow Q}$$

Fig. 6. Nondeterministic omni-big-step semantics for an imperative language (selected rules)

the address contained in local variable x, but only if memory at address $a + n$ has already been allocated.

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, resulting in *external* nondeterminism. 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. This rule encodes *internal* nondeterminism, because this action is not recorded in the event trace. Semantics of while loops are given by sequencing the first iteration with the loop itself as long as the loop test succeeds.

In practice, we found it convenient also to derive a chained version EVAL-SEQ-CHAINED of the omni-big-step rule EVAL-SEQ, just like we did for omni-small-step rules in §3.2.

$$\textsc{eval-seq-chained}: \qquad c_1/m/\ell/\tau \Downarrow \Big(\lambda m'\ell'\tau'.\ (c_2/m'/\ell'/\tau' \Downarrow Q)\Big) \quad \Rightarrow \quad (c_1; c_2)/m/\ell/\tau \Downarrow Q$$

Note that the chained variant cannot be used to define the judgement inductively in Coq due to the *strict positivity* requirement; more details on encoding choices can be found in Appendix A.

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.

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

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 we 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.

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

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

Theorem 6.1 (omnisemantics simulation for the pair-heapification compiler).

$$\begin{array}{c}\forall c\ m_{src}\ \ell_{src}\ m_{tgt}\ \ell_{tgt}\ \tau\ Q.\ \mathsf{tmp} \notin \mathsf{vars}(c)\ \wedge\ R((m_{src}, \ell_{src}), (m_{tgt}, \ell_{tgt}))\ \wedge \\ c/m_{src}/\ell_{src}/\tau \Downarrow_{src} Q \implies \\ C(c)/m_{tgt}/\ell_{tgt}/\tau \Downarrow_{tgt} Q_R\end{array}$$

$$\text{where}\quad Q_R(m'_{tgt}, \ell'_{tgt}, \tau') := \exists m'_{src}\ \ell'_{src}.\ R((m'_{src}, \ell'_{src}), (m'_{tgt}, \ell'_{tgt})) \wedge Q(m'_{src}, \ell'_{src}, \tau')$$

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 6 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:

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

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 an omni-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 smooth outcome 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 omnisemantics made its addition 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 $\mathsf{let}\ x = \mathsf{stackalloc}(n)\ \mathsf{in}\ c$ made available in 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 omni-big-step rule shown below, like in rule OMNI-BIG-REF from §2.

$$\frac{\begin{array}{c}\forall m_{new}\ a.\ (\mathsf{dom}\ m_{new} \cap \mathsf{dom}\ m) = \varnothing \ \wedge\ \mathsf{dom}\ m_{new} = [a, a+n) \implies \\ c/(m \cup m_{new})/\ell[x := a]/\tau \Downarrow \lambda m'\ \ell'\ \tau'.\ P(m' \setminus m_{new}, \ell', \tau')\end{array}}{(\mathsf{let}\ x = \mathsf{stackalloc}(n)\ \mathsf{in}\ c)/m/\ell/\tau \Downarrow P}\ \textsc{omni-big-stackalloc}$$

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.

Compiler-correctness proof. The compiler-correctness proof proceeds by induction on the omnisemantics 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 omnisemantics 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 omnisemantics 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 an omni-forward-preservation proof boils down to instantiating a universal quantifier.

Design decisions around proving absence of out-of-memory. In the verified software-hardware stack described in Erbsen et al. [2021], the main bottleneck in terms of complexity that prevents us from developping bigger applications is the level of proof automation available for verification of mundane aspects of source programs such as address arithmetic. Therefore, we made an effort to avoid adding more proof obligations in the program logic whenever possible. At the same time, for the targeted application it was fine to limit the expressivity of the source language. In particular, we decided that disallowing recursive calls is acceptable. Given that setting, we want to avoid reasoning about out-of-memory conditions in the source language, while still proving that the *compiled* program will not run out of memory, which we can achieve as follows.

In the OMNI-BIG-STACKALLOC rule of our source language, we deliberately use a vacuous universal quantification if we run out of memory, because we prefer to handle out-of-memory conditions outside of the omnisemantics judgment, in an additional external judgment. In particular, this means that if OMNI-BIG-STACKALLOC is applied with a memory m whose domain already contains all (or almost all) addresses (which are 32-bit or 64-bit words), there might be no m_{new} and a such that the left-hand side of the implication above the line in OMNI-BIG-STACKALLOC holds, so we can derive any postcondition P, something that we cautioned against in §2.2.

Effectively, this means that our source-language evaluation rules do not guarantee that the program never runs out of memory. This choice simplifies the program-logic proofs for concrete input programs but requires additional work in the compiler: the compiler performs a simple static-analysis pass over the call graph of the program to determine the maximum amount of stack space that the program needs. Since this analysis rejects recursive calls, the space upper bound is not hard to compute. The compiler-correctness proof contains an additional hypothesis requiring that at least that computed amount of memory is available in the state on which the target-language program begins its execution.

An alternative approach would be to introduce a notion of "amount of used stack space" in the source-language semantics and include an additional precondition in the OMNI-BIG-STACKALLOC rule that requires this amount to be bounded. This approach would put more complexity into the verification of source programs, while simplifying the compiler correctness proof. In order to allow recursive calls and dynamically chosen stack-allocation sizes, reasoning about the amount of stack space in the program logic seems to become unavoidable, in which case this alternative approach would be preferrable.

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

If the semantics of the source language of a compiler phase are most naturally expressed in omni-big-step, but the target language's semantics are best expressed in omni-small-step semantics, it is convenient to prove an omni-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 omni-big-step to omni-small-step during the correctness proof.

Consider one sample omni-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 an omni-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 an omni-single-small-step goal with a given postcondition, which is suitable to discharge using 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 RELATED WORK

This works builds on that of Schäfer et al. [2016], Charguéraud [2020], and Erbsen et al. [2021], all of which are discussed in the introduction (§1). We now will review some additional connections.

Relationship to coinductive 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 $t/s \Uparrow^{co}$, 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 omni-big-step judgment involves a single rule, namely CO-OMNI-BIG-LET, defined as part of the coinductive interpretation of the rules from Fig. 2.

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

In that rule, if Q_1 is instantiated as the empty set, the second premise 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 the rule requires the term $[v_1/x]\,t_2$ to diverge, just like in the rule DIV-LET-2. In summary, CO-OMNI-BIG-LET captures at once the logic of both DIV-LET-1 and DIV-LET-2.

The paper by Leroy and Grall [2009], which focuses on a deterministic semantics, points out that the principle of excluded middle (classical logic) is required for establishing the equivalence between a *coinductive big-step semantics* for divergence and the *standard small-step semantics* for divergence. Interestingly, classical logic is *not* required for establishing the equivalence between a *coinductive omni-big-step* semantics of divergence and the standard small-step semantics for divergence. In the explanations that follow, we omit the state for simplicity, and we write $t \longrightarrow^{\infty}_{\text{co}}$ for the standard small-step divergence judgment, defined as $\forall t'.\ (t \longrightarrow^* t') \Rightarrow \exists t''.\ (t' \longrightarrow t'')$.

The implication that requires classical logic to be established is: $(t \longrightarrow^{\infty}_{\text{co}}) \Rightarrow (t \Uparrow^{\text{co}})$. To see why, consider a term t of the form $\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2$, where t_1 corresponds to a program whose termination is an open mathematical problem, and where t_2 is an infinite loop. Thus, no matter whether t_1 diverges or not, the program $\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2$ diverges. Yet, to establish the judgment $(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2) \Uparrow^{\text{co}}$, one needs to know whether t_1 diverges, in which case the rule DIV-LET-1 applies, or whether t_1 terminates, in which case the rule DIV-LET-2 applies. In the general case, one has to invoke the excluded middle to decide on the termination of an abstract term t_1.

In contrast, the implication $(t \longrightarrow^{\infty}_{\text{co}}) \Rightarrow (t \Downarrow^{\text{co}} \emptyset)$, which targets a coinductive omni-big-step semantics, can be proved without classical logic, as pointed out earlier in §3.4. Intuitively, to prove that the same example program $\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2$ diverges, one can apply the rule CO-OMNI-BIG-LET, regardless of whether t_1 diverges or not. It suffices to instantiate Q_1, which denotes the set of possible results of t_1, as the strongest postcondition of t_1. The strongest postcondition may be expressed in terms of the omni-big-step judgment (recall §2.2), or equivalently in terms of the small-step judgment by instantiating Q_1 as $\{v_1 \mid t_1 \longrightarrow^* v_1\}$. In particular, if t_1 diverges, then the set Q_1 is empty and the second premise of CO-OMNI-BIG-LET becomes vacuous. What matters for the proof of equivalence between the small-step semantics and the coinductive omni-big-step semantics is that we do not need to *decide* whether Q_1 is empty, i.e., whether t_1 diverges or not. We thereby avoid the need for the excluded middle.

Coinductive characterization of safety. Wang et al. [2014] define a safety judgment, written $\mathsf{safe}(t, s)$, to assert that all possible executions of the configuration t/s execute safely, i.e., do not get stuck. To reason in big-step style, and to avoid the cumbersome introduction of error-propagation rules, they consider a coinductive definition. We reproduce below the rule for let-bindings, which reads as follows: to establish that $\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2$ executes safely, prove 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.

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

Our judgment $t/s \Downarrow^{\mathrm{co}} Q$ generalizes the notion of safety, by baking the postcondition directly into the judgment (§2.4). It asserts not only that t/s cannot get stuck but also that any potential final configuration belongs to Q. We formalized in Coq the following equivalence.

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

Our rule omni-big-let extends safe-let not just by adding the postcondition Q to the judgment but also by changing the quantification over v_1/s'. In the rule safe-let, the quantification is constrained by $t_1/s \Downarrow v_1/s'$, whereas in the rule omni-big-let, it is constrained by $(v_1, s') \in Q_1$, where Q_1 denotes the postcondition of t_1/s. The key innovation here is that, thanks to the introduction of postconditions, we no longer need to refer to the standard big-step judgment—the judgment $t/s \Downarrow Q$ gives a stand-alone definition of the semantics of the language.

Semantics of nondeterministic programs. An important aspect of the present paper is the set up of semantics for nondeterministic language constructs. Let us review the key historical papers that have focused on that task. 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 $(p\ \mathsf{par}\ q) = (q\ \mathsf{par}\ p)$ 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 recent work by Tassarotti et al. [2017] heavily relies on the bounded nondeterminism assumption in an extension of Iris [Jung et al. 2018] for developing a logic to justify program refinement. These authors speculate that *transfinite step-indexing* [Schwinghammer et al. 2013; Svendsen et al. 2016] may allow handling unbounded nondeterminism.

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. Despite apparent benefits, this approach seems not to have gained popularity or evaluation in the form of sizable case studies.

Compiler correctness as trace property preservation. Abate et al. [2021] define the notion of *source trace property preservation* (denoted $\mathrm{TP}^{\tilde{\tau}}$) to mean that all properties that hold on traces produced by the source program also hold on traces produced by the target program. They allow different trace formats in the source and target language, relating the source trace s to a target trace t by a relation $s \sim t$ and quantifying over them in the same way as we quantify over the source and target states in the definition of *omnisemantics simulation* (§6.3). If we instantiate the definition of Abate et al. [2021] by traces whose single events stand for emitting final states, we obtain our definition of omnisemantics simulation, and vice versa, if we generalize our definition to also allow different trace formats but omit the state component, we obtain their definition. However, including the state component in our definition makes it directly usable for a forward-style proof by induction on the source-language derivation, even in the presence of target-language nondeterminacy. We speculate that several proofs of example compilers in that paper could be revisited using omnisemantics. Doing so would not only simplify the proofs but also make the results stronger by removing the target-language determinacy assumption, which they need to derive backward simulations from forward simulations.

Semantics of concurrent programs. Concurrency increases the amount of nondeterminism, due to interleavings, and generally increases the sources of undefined behaviors, due in particular to data races. The work on CompCertTSO [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 omni-small-step semantics would help simplify proofs from CompCertTSO.

The Iris framework [Jung et al. 2018, 2015] supports reasoning about concurrent programs in Separation Logic. In Iris, the source language is specified by means of a traditional small-step semantics. The weakest preconditions predicate is then defined using *step-indexing*: one first defines the notion of "a program is well-behaved for n steps" by induction over n; then defines the notion of "a program is well-behaved" as "it is well-behaved for any number of steps". Proofs are then typically carried out by induction over the indices. Yet, the indices involved get in the way of compiler proofs where the number of computation steps may increase or decrease throught a transformation. This observation motivated the introduction of more advanced techniques to tame the issue, such as transfinite step-indexing [Svendsen et al. 2016]. When reasoning about *sequential* programs, the use of step-indexing appears overkill for most applications: by leveraging an inductive definition of the weakest precondition predicate, an omni-big-step semantics provides a direct induction principle that avoids the technicalities and limitations of step-indexing altogether.

Reasoning about termination. Many foundational program logics provide reasoning rules for partial correctness, e.g. [Cao et al. 2018; Chlipala 2013; Jung et al. 2018; Ni and Shao 2006]. More recent frameworks have aimed to support reasoning about total correctness. For example, in the context of the CakeML verified compiler [Kumar et al. 2014], the work by Guéneau et al. [2017], subsequently simplified by Charguéraud [2022], provides a foundational approach to CFML's characteristic formulae [Charguéraud 2011]. In the context of the Iris framework [Jung et al. 2018], Mével et al. [2019] encode in the notion of *time credits* [Charguéraud and Pottier 2019] for establishing upper and lower bounds on the execution cost. The existence of an upper bound on the number of time credits required by a program guarantees the termination of that program. Yet, that bound must be provided *upfront*, thus this approach is not complete. To see why, consider as counterexample a program that picks an unbounded random number in $\mathbb{Z}$, then executes a loop that number of time.[9] This program is terminating according to the operational semantics, yet it does not admit any bound on its execution time. To handle such programs, Spies et al. [2021] introduce

[9]Operational semantics need not provide an actual implementation for the operation of picking a random number in $\mathbb{Z}$.

transfinite time credits, which allow for termination arguments based on dynamic information learned during program execution. In comparison, omnisemantics can handle such programs without requiring sophisticated models of Separation Logic. Indeed, an inductive omnisemantics derivation *inherently* corresponds to a transfinite derivation tree.

Semantics of probabilistic programs. Probabilistic semantics aim to describe not just which executions are possible but also to describe with what probability each execution may happen. A probabilistic small-step execution relation assigns a probability to every transition. One caveat is that probabilities do not suffice to describe all nondeterminism: in particular, memory is allocated at nondeterministically chosen addresses. We refer to Batz et al. [2019] for a solution to this challenge. In the context of program logics, McIver and Morgan [2005] introduce a *weakest preexpectation calculus*. Batz et al. [2019] generalize this notion to set up a *Quantitative Separation Logic*.

Additionally, there is a long line of work aiming at providing denotational models for probabilistic programs—e.g., Staton et al. [2016]; Wang et al. [2019]. Denotational and operational semantics serve different purposes; one important practical benefit of omnisemantics is that it is grounded in inductive definitions, with respect to which proofs by induction can be carried out easily in a proof assistant. An interesting question is whether omnisemantics could be adapted to provide an inductively defined operational semantics that accounts for probabilities, by relating configurations not to sets of outcomes but instead to probability distributions of outcomes.

The problem of termination of probabilistic programs is particularly subtle. One the one hand, one may be interested in capturing that any execution terminates. For example, Staton et al. [2016] define termination as "there exists n, such that termination occurs in n steps." However, this approach does not apply to infinitely branching nondeterminism. On the other hand, one may design rules to establish *almost-sure termination* or *positive-almost-sure termination* [Chakarov and Sankaranarayanan 2013; Ferrer Fioriti and Hermanns 2015; Kaminski et al. 2016; McIver et al. 2017].

Dijkstra monads. Dijkstra monads [Ahman et al. 2017; Maillard et al. 2019] target code written in monadic form and specified using dependent types. The type-checking process essentially applies weakest-precondition rules and results in the production of proof obligations. In practice, specifications are expressed in first-order logic, so that proof obligations can be discharged using SMT solvers. Dijkstra monads encourage metareasoning using object-language dependent types only; they do not appear to have been designed for, or demonstrated capable of, carrying out inductions over program executions. Dijkstra monads can be instantiated in particular with the nondeterminism monad (NDet). In the current presentation [Ahman et al. 2017], the monad models sets of possible outcomes using finite sets, which rules out infinitely branching nondeterminism and does not allow for abstraction in intermediate postconditions (e.g., asserting that a subterm t_1 returns an even integer).

8 CONCLUSION AND FUTURE WORK

This paper provides an in-depth introduction to the definitions, properties, and applications of the omni-big-step and omni-small-step semantics. These applications include mechanized proofs of type soundness, foundational constructions of Separation Logic, and compiler correctness proofs.

It would be interesting future work to investigate whether a mixed inductive-coinductive version of omni-big-step semantics could be defined and provide smooth reasoning for the combined challenge of potentially infinite executions, nondeterminism, and undefined behavior. The key challenge is to find a way to carry out compiler-correctness proofs through a single pass that handles reasoning about both terminating and nonterminating executions.

REFERENCES

Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2021. An Extended Account of Trace-relating Compiler Correctness and Secure Compilation. *ACM Transactions on Programming Languages and Systems* 43, 4 (Nov. 2021), 14:1–14:48. https://doi.org/10.1145/3460860

Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra Monads for Free. *ACM SIGPLAN Notices* 52, 1 (Jan. 2017), 515–529. https://doi.org/10.1145/3093333.3009878

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

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

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative Separation Logic: A Logic for Reasoning about Probabilistic Pointer Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 34 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290347

Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. 2005. Semantics of separation-logic typing and higher-order frame rules. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*. IEEE, 260–269. https://doi.org/10.1109/LICS.2005.47

Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

Qinxiang Cao, Shengyi Wang, Aquinas Hobor, and Andrew W. Appel. 2018. Proof pearl: Magic wand as frame. Unpublished.

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, 511–526. 10.1007/978-3-642-39799-8_34

Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. In *International Conference on Functional Programming (ICFP '11)*. Association for Computing Machinery, 418–430. https://doi.org/10.1145/2034773.2034828

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

Arthur Charguéraud. 2020. Separation Logic for Sequential Programs (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 116 (Aug. 2020), 34 pages. https://doi.org/10.1145/3408998

Arthur Charguéraud. 2022. *A Modern Eye on Separation Logic for Sequential Programs*. Technical Report. 142 pages. https://hal.inria.fr/hal-03864664v1

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning* 62, 3 (March 2019), 331–365. https://doi.org/10.1007/s10817-017-9431-7

Adam Chlipala. 2013. The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Association for Computing Machinery, 391–402. https://doi.org/10.1145/2500365.2500592

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 (POPL '92)*. Association for Computing Machinery, 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. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, 604–619. https://doi.org/10.1145/3453483.3454065

Luis María Ferrer Fioriti and Holger Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. *SIGPLAN Not.* 50, 1 (Jan. 2015), 489–501. https://doi.org/10.1145/2775051.2677001

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.

Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. 2017. Verified Characteristic Formulae for CakeML. In *European Symposium on Programming (ESOP)*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, 584–610. https://doi.org/10.1007/978-3-662-54434-1_22

Aquinas Hobor and Jules Villard. 2013. The Ramifications of Sharing in Data Structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. Association for Computing Machinery, 523–536. https://doi.org/10.1145/2429069.2429131

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018). https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, 637–650. https://doi.org/10.1145/2676726.2676980

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, 364–389. https://doi.org/10.1007/978-3-662-49498-1_15

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

Robbert Krebbers. 2015. *The C standard formalized in Coq*. Ph.D. Dissertation. Radboud University Nijmegen. https://robbertkrebbers.nl/research/thesis.pdf

Neel R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. 2010. Verifying Event-Driven Programs Using Ramified Frame Properties. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '10)*. Association for Computing Machinery, 63–76. https://doi.org/10.1145/1708016.1708025

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*. ACM Press, 179–191. https://doi.org/10.1145/2535838.2535841

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

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).

Nancy Lynch and Frits Vaandrager. 1995. Forward and Backward Simulations Part I: Untimed Systems. *Information and Computation* 121 (1995), 214–233.

Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra Monads for All. *Proceedings of the ACM on Programming Languages* 3, ICFP (July 2019), 104:1–104:29. https://doi.org/10.1145/3341708

Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement And Proof For Probabilistic Systems*. Springer.

Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. 2017. A New Proof Rule for Almost-Sure Termination. *Proc. ACM Program. Lang.* 2, POPL, Article 33 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158121

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

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

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.

Zhaozhong Ni and Zhong Shao. 2006. Certified Assembly Programming with Embedded Code Pointers. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. Association for Computing Machinery, 320–333. https://doi.org/10.1145/1111037.1111066

O'Hearn, Reynolds, and Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL: 15th Workshop on Computer Science Logic*. LNCS, Springer-Verlag. https://doi.org/10.1007/3-540-44802-0_1

Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. https://doi.org/10.1145/3211968 The appendix is linked as supplementary material from the ACM digital library.

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

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

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Annual IEEE Symposium on Logic in Computer Science (LICS)*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

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

Steven Schäfer, Sigurd Schneider, and Gert Smolka. 2016. Axiomatic Semantics for Compiler Verification. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. ACM, St. Petersburg FL USA, 188–196. https://doi.org/10.1145/2854065.2854083

Jan Schwinghammer, Aleš Bizjak, and Lars Birkedal. 2013. Step-indexed relational reasoning for countable nondeterminism. *Logical Methods in Computer Science* 9 (2013). https://doi.org/10.2168/LMCS-9(4:4)2013

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

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, 80–95. https://doi.org/10.1145/3453483.3454031

Sam Staton, Hongseok Yang, Chris Heunen, Ohad Kammar, and Frank Wood. 2016. Semantics for Probabilistic Programming: Higher-Order Functions, Continuous Distributions, and Soft Constraints. *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (July 2016), 525–534. https://doi.org/10.1145/2933575.2935313 arXiv:1601.04943

Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. 2016. Transfinite Step-Indexing: Decoupling Concrete and Logical Steps. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, 727–751. https://doi.org/10.1007/978-3-662-49498-1_28

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

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

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

A ON THE CHALLENGE OF DEFINING WP INDUCTIVELY

The weakest-precondition-style reasoning rule for let-bindings is traditionally stated as follows.

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

Translating it to a big-step omnisemantics rule results in the following rule.

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

The rule omni-big-let-chained can be useful for reasoning when one does not want to specify an explicit postcondition that needs to hold between t_1 and t_2. This *chained* rule can be straightforwardly derived from the omni-big-let rule part of the definition of the omni-big-step semantics, by instantiating Q_1 as $\{(v', s') \mid ([v'/x]\, t_2)/s' \Downarrow Q\}$ in the first premise, then checking the tautology associated with the second premise.

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

One might wonder why we do not use omni-big-let-chained directly in the inductively defined rules. The reason is that Coq's *strict positivity* requirement on the well-formedness of inductive definitions does not allow it.

To elaborate on this point, consider the four candidate Coq rules stated below.

```
Notation "H1 ⊢H2" := (∀ s, H1 s → H2 s). (* notation for entailment *)
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_invalid' : ∀Q1 x t1 t2 Q s, (* non strictly positive occurrence of [wp]. *)
      wp t1 Q1 s →
      Q1 = (fun v1 s2 ⇒ wp (subst x v1 t2) Q s2) →
      wp (trm_let x t1 t2) Q s
  | wp_let_valid : ∀x t1 t2 Q, (* accepted, but with useless induction principle *)
      (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_valid' : ∀x t1 t2 Q1 Q, (* accepted, with useful induction principle *)
      wp t1 Q1 s →
      (∀ v1 s2, Q1 v1 s2 → wp (subst x v1 t2) Q s2) →
      wp (trm_let x t1 t2) Q s.
```

The first rule directly translates wp-let. It is rejected by Coq because it includes a non-strictly-positive occurrence of the predicate wp.

The second rule attempts a reformulation by expanding the definition of entailment and by introducing a variable name Q1 for the intermediate postcondition, together with an equality constraint on Q1. Yet, Coq rejects this rule just like the previous.

The third rule modifies the first rule by introducing an existentially quantified intermediate postcondition Q1, quantifying over the items that belong to it. This rule is accepted by Coq. Yet, in that form, Coq (v8.14) generates a useless induction principle, which provides no induction hypothesis for the nested occurence of `wp`. (This weakness can be corrected by stating and proving an induction principle manually, but we prefer to avoid the extra hassle.)

The fourth rule corresponds to OMNI-BIG-LET. It adapts the previous rule by quantifying `Q1` universally at the level of the constructor. This presentation is properly recognized by the induction-principle generator of Coq.

B UNSPECIFIED EVALUATION ORDER

For a language that uses unspecified but consistent order of evaluation for arguments of, e.g., pairs or applications, we can consider a generalized version of the rule OMNI-BIG-PAIR from the previous section. Essentially, we duplicate the premises to account for the two possible evaluation orders.

OMNI-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 the 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.

Note that we do not attempt to model languages that allow arbitrary interleavings in the evaluation of arguments, as, e.g., arithmetic expressions in the C language [Krebbers 2015]. More generally, concurrent evaluation is out of the scope of the present paper.

C OMNISEMANTICS 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. Moreover, the exception raised should be included in the set of results that the let-binding can produce.

There are two ways to extend the grammar of results with exceptions. The first possibility is to add a constructor to the grammar of values. In this case, the postcondition Q remains a predicate over pairs of values and states. The second possibility is to introduce a type, to capture the sum of the type of values and of the type of exceptions. In that case, the postcondition Q becomes a predicate over pairs of results and states.

For simplicity, let us assume in what follows that the grammar of values includes a constant exception construct, written exn. In that setting, the omni-big-step evaluation rule for a let-binding of the form $(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)$ can be stated as follows. The first premise describes the evaluation of t_1. The second premise handles the case where t_1 produces a normal value. The third premise handles the case where t_1 produces an exception.

OMNI-BIG-LET-WITH-EXCEPTIONS

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

We proved in Coq the equivalence of this treatment of exceptions with the formalization of exceptions expressed both in standard small-step and in standard big-step semantics.

D DEFINITION OF THE TERMINATION JUDGMENT

We introduced the termination judgment to formalize the interpretation of the omni-big-step judgment (§2.2, OMNI-BIG-STEP-IFF-TERMINATES-AND-CORRECT). The predicate $\mathsf{terminates}(t, s)$

asserts that all executions of configuration t/s terminate. In this section, we present two formal definitions of this predicate, one in small-step style and one in big-step style.

The small-step version is inductively defined by the two rules show below.

$$\frac{}{\mathsf{terminates}(v,s)}\ \textsc{small-terminates-here}$$

$$\frac{\left(\exists t's'.\ \ t/s \longrightarrow t'/s'\right) \qquad \left(\forall t's'.\ \ (t/s \longrightarrow t'/s')\ \Rightarrow\ \mathsf{terminates}(t',s')\right)}{\mathsf{terminates}(t,s)}\ \textsc{small-terminates-step}$$

The big-step version is inductively defined using one rule per language construct. We show below the rules for values and for let-bindings. This definition corresponds to an inductive version of the coinductive judgment safe from Wang et al. [2014], described in §8.

$$\frac{}{\mathsf{terminates}(v,s)}\ \textsc{big-terminates-val}$$

$$\frac{\mathsf{terminates}(t_1,s) \qquad \left(\forall v_1 s'.\ (t_1/s \Downarrow v_1/s')\ \Rightarrow\ \mathsf{terminates}(([v_1/x]\,t_2),s')\right)}{\mathsf{terminates}((\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2),s)}\ \textsc{big-terminates-let}$$

E DEFINITION OF THE TYPING JUDGMENT

This section states the typing rules for the state-free language considered in §4.1. The typing rules are given for terms in A-normal form. 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. Finally, $\mathbb{V}$ denotes the set of terms that are either values or variables.

$$\frac{}{\vdash\ tt\ :\ \mathsf{unit}}\ \textsc{vtyp-unit} \qquad \frac{}{\vdash\ b\ :\ \mathsf{bool}}\ \textsc{vtyp-bool} \qquad \frac{}{\vdash\ n\ :\ \mathsf{int}}\ \textsc{vtyp-int}$$

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

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

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

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

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

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

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

F EXTENSION OF THE TYPING JUDGMENT FOR STATE

This section states the typing rules for the imperative language considered in §4.2. There, the typing judgment for terms takes the form $S; E\ \vdash\ t\ :\ T$, and the typing judgment for closed values takes the form $S\ \vdash\ v\ :\ T$, where the store typing S maps locations to types. The rules from the previous appendix are extended simply to thread S throughout the judgment. The new rules include the rule for typing locations and the rules for memory operations. They are shown next.

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

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

G DEFINITION OF THE STANDARD SMALL-STEP JUDGMENT

In §2.4, we gave a characterization of coinductive omni-big-step semantics in terms of the standard small-step semantics, written $t/s \longrightarrow t'/s'$. For reference, we give below the rules that define the standard small-step judgment:

$$\frac{v_1 = (\mu f.\lambda x.t)}{(v_1\, v_2)/s \longrightarrow ([v_2/x]\, [v_1/f]\, t)/s}\ \textsc{small-app} \qquad \frac{}{(\mathsf{if\ true\ then}\ t_1\ \mathsf{else}\ t_2)/s \longrightarrow t_1/s}\ \textsc{small-if-true}$$

$$\frac{}{(\mathsf{if\ false\ then}\ t_1\ \mathsf{else}\ t_2)/s \longrightarrow t_2/s}\ \textsc{small-if-false} \qquad \frac{t_1/s \longrightarrow t_1'/s'}{(\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2)/s \longrightarrow (\mathsf{let}\ x = t_1'\ \mathsf{in}\ t_2)/s'}\ \textsc{small-let-ctx}$$

$$\frac{}{(\mathsf{let}\ x = v_1\ \mathsf{in}\ t_2)/s \longrightarrow ([v_1/x]\, t_2)/s}\ \textsc{small-let-val} \qquad \frac{}{(\mathsf{add}\, n_1\, n_2)/s \longrightarrow (n_1 + n_2)/s}\ \textsc{small-add} \qquad \frac{0 \le m < n}{(\mathsf{rand}\, n)/s \longrightarrow m/s}\ \textsc{small-rand}$$

$$\frac{p \notin \mathrm{dom}\, s}{(\mathsf{ref}\, v)/s \longrightarrow (s[p := v])/s}\ \textsc{small-ref} \qquad \frac{p \in \mathrm{dom}\, s}{(\mathsf{free}\, p)/s \longrightarrow tt/(s \setminus p)}\ \textsc{small-free}$$

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

H EVALUATION OF UNARY AND BINARY OPERATORS

The following definitions complete the semantics described in the case study "compiling immutable pairs to heap-allocated records" (§6.4).

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

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