Semantics of Sets of Programs

JINWOO KIM*, University of California-San Diego, USA
SHAAN NAGY*, University of California-San Diego, USA
THOMAS REPS, University of Wisconsin-Madison, USA
LORIS D'ANTONI, University of California-San Diego, USA

Applications like program synthesis sometimes require proving that a property holds for all of the infinitely many programs described by a grammar—i.e., an inductively defined set of programs. Current verification frameworks overapproximate programs' behavior when sets of programs contain loops, including two Hoare-style logics that fail to be relatively complete when loops are allowed. In this work, we prove that compositionally verifying simple properties for infinite sets of programs requires tracking distinct program behaviors over unboundedly many executions. Tracking this information is both necessary and sufficient for verification. We prove this fact in a general, reusable theory of denotational semantics that can model the expressivity and compositionality of verification techniques over infinite sets of programs. We construct the minimal compositional semantics that captures simple properties of sets of programs and use it to derive the first sound and relatively complete Hoare-style logic for infinite sets of programs. Thus, our methods can be used to design minimally complex, compositional verification techniques for sets of programs.

CCS Concepts: • **Software and its engineering** → **Software verification**; **Semantics**; **Formal software verification**; • **Theory of computation** → *Proof theory*; *Automated reasoning*; *Hoare logic*; **Denotational semantics**; **Program verification**.

Additional Key Words and Phrases: Unrealizability Logic, compositional semantics, infinite sets of programs

ACM Reference Format:
Jinwoo Kim, Shaan Nagy, Thomas Reps, and Loris D'Antoni. 2025. Semantics of Sets of Programs. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 110 (April 2025), 28 pages. https://doi.org/10.1145/3720515

1 Introduction

The central problem of this work is to understand what formal semantics are needed when verifying infinite sets of imperative programs.

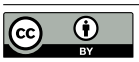
Example 1.1 (Verifying a Set of Imperative Programs). Consider the following grammar, which defines a set $L(W)$ of loops:

$$W ::= \text{while } x < E \text{ do } x := x + 1$$
$$E ::= 0 \mid E + 2$$

Can we prove that, for every loop $w \in L(W)$, running w on the initial value $x = 0$ results in a final state in which x is even?

*Jinwoo Kim and Shaan Nagy contributed equally to this paper.

Authors' Contact Information: Jinwoo Kim, University of California-San Diego, San Diego, USA, pl@ucsd.edu; Shaan Nagy, University of California-San Diego, San Diego, USA, shnagy@ucsd.edu; Thomas Reps, University of Wisconsin-Madison, Madison, USA, reps@cs.wisc.edu; Loris D'Antoni, University of California-San Diego, San Diego, USA, ldantoni@ucsd.edu.



ACM 2475-1421/2025/4-ART110
https://doi.org/10.1145/3720515

The need to verify a property of an infinite set of programs appears in several domains. For example, in language-wide verification, one may wish to prove that every program in a given domain-specific language terminates or satisfies some safety property. Another example is verifying programs that contain constructs like `eval`, which load code as a string/tree at run time and dynamically execute it. In this setting, one may need to reason about infinitely many possible strings/trees that represent programs and are passed to `eval` [23]. A third example, and the main motivation behind our work, is proving unrealizability of program-synthesis problems [18], where the goal is to determine whether all programs in a search space satisfy the negation of a given specification—i.e., no program in the search space matches the desired specification. Unrealizability of program-synthesis problems can be used to speed up program synthesis and also to prove optimality of a program c with respect to a given metric [17]—i.e., by showing that the problem of synthesizing a program that is better than c is unrealizable.

While there are many techniques to prove that a *particular* loop $w \in L(W)$ satisfies the specification given in Example 1.1, no verification technique that we are aware of can prove that the specification holds for *all* $w \in L(W)$.

Our two main goals in this paper are: (*i*) to present a fundamental theoretical obstacle that makes it hard to design effective frameworks for reasoning about extensional properties (i.e., input-output properties) of sets of deterministic imperative programs with loops (e.g. Example 1.1), and (*ii*) in light of this obstruction, to present a provably simplest semantic basis for designing verification techniques. To ground our study, we first consider reasoning about individual imperative programs.

Denotational Semantics for Single Imperative Programs. Formal verification can always be defined with respect to a denotational semantics—i.e., a map that assigns a mathematical object, called a denotation, to each program. These denotations capture the program information one is interested in studying. A familiar example of a denotation is a *state transformer*—i.e., a function (or relation) that captures how a program maps input states to output states.

In this paper, we are interested in understanding how powerful denotations must be to capture extensional properties for sets of programs. For a *single program* c, an *extensional property* is a property of the relation between inputs and outputs captured by the state transformer of c. [1] For example, monotonicity and commutativity are extensional properties—they are direct properties of the mathematical relation that the state transformer assigned to c induces. On the other hand, program text size or running time of the program are *not* extensional properties—they are unrelated to the input-output relation described by c.

To reason about extensional properties of individual deterministic imperative programs, one often uses a semantics that maps every syntactic program to a partial function mapping input program states σ to output program states [31, 33], illustrated in the following example:

$$[\![x := 10]\!] = \lambda\sigma.\sigma[x \mapsto 10] \tag{1}$$

This semantics is the simplest one that expresses single-input extensional properties (i.e., the output state of the program on a given single input state σ). Despite its simplicity, this denotational semantics boasts two key properties that enable effective verification techniques to be built on top of it, the second of which enables the first:

Compositionality. The denotational semantics of a complex program can be expressed precisely in terms of the semantics of its subprograms, as shown by the following example:

$$[\![x := 10; y := 12]\!] = [\![y := 12]\!] \circ [\![x := 10]\!]$$

[1] In this paper, we use c to refer to arbitrary single programs (i.e., program statements or expressions). We use s to refer specifically to program statements (derivable from *Stmt* in the grammar given in Figure 3). Similarly, we use C to refer to sets of programs, and we use S to refer to sets of program statements.

Expression of Multiple Executions. The semantics of a program is capable of describing how the program acts on multiple different input states. For example, to understand how c acts on two states rather than just one, one need not define a new semantics. Instead, one can apply $[\![c]\!]$ to each state:

$$(\sigma_1, \sigma_2) \mapsto ([\![c]\!](\sigma_1), [\![c]\!](\sigma_2)) \tag{2}$$

Both traits are desirable for program analysis and verification. Compositionality enables scalable program analysis: a programcan be analyzed by decomposing the original analysis into simpler ones over smaller subprograms. Expression of multiple executions enables compositionality over constructs that repeatedly execute code (e.g., while loops): to determine the behavior of a while loop "while b do s" on an input state σ compositionally, one must understand how b and s behave on every state that appears in a loop iteration. For example, to see that $[\![\text{while } x < 2 \text{ do } x := x + 1]\!](x \mapsto 0) = (x \mapsto 2)$, one must observe multiple executions of the guard and body:

$$\begin{array}{lcl} [\![x{<}\,2]\!](x \mapsto 0) \;=\; t & \longrightarrow & [\![x := x{+}1]\!](x \mapsto 0) \;=\; (x \mapsto 1) \\ & \swarrow & \\ [\![x{<}\,2]\!](x \mapsto 1) \;=\; t & \longrightarrow & [\![x := x{+}1]\!](x \mapsto 1) \;=\; (x \mapsto 2) \\ & \swarrow & \\ [\![x{<}\,2]\!](x \mapsto 2) \;=\; f & & \end{array}$$

Although one might take the power to express multiple executions for granted in this single-program semantics, it turns out to be surprisingly hard to design a semantics of *sets* of programs that can express multiple executions for multiple programs *at once*. The reason is that "natural" extensions lifting single-program semantics to semantics for sets of programs tend to mix the outputs of different programs together, making it hard to tell when two possible outputs on two different inputs can be produced by the same program. As a result, it is hard to design a compositional semantics for a set of programs, where the programs can contain loops.

Challenges in Designing a Denotational Semantics for Sets of Imperative Programs. Taking inspiration from the properties of denotational semantics of individual programs, this paper addresses the following objective:

*Design a **compositional** semantics for **sets of programs** that can express **extensional properties**.*

The simplest extensional properties are *single-input extensional properties*, which for a single program c, are the facts that can be deduced by computing $[\![c]\!](\sigma)$ on some state σ. To generalize single-input extensional properties to a set of programs C, our first attempt is a

straightforward lifting of the single-program semantics in Equation (1)—which we call $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$, the "program-agnostic" semantics for sets of programs. Whereas $[\![c]\!]$ has the signature State $\to$ State, the signature of $\langle\!\langle C \rangle\!\rangle^{\mathbf{a}}$ must be $2^{\text{State}} \to 2^{\text{State}}$ because C may generate multiple outputs for a single input (i.e., one for each $c \in C$). To maintain compositionality on, e.g., sequential composition $(S_1; S_2)$, $\langle\!\langle C \rangle\!\rangle^{\mathbf{a}}$ must have signature $2^{\text{State}} \to 2^{\text{State}}$—that is, *single-input* extensional properties require reasoning about *sets* of inputs in the case of sets of programs. Then on a given set of input states P, $\langle\!\langle C \rangle\!\rangle^{\mathbf{a}}$ returns the set of all output states that programs in C can generate:

$$\langle\!\langle C \rangle\!\rangle^{\mathbf{a}} = \lambda P.\{[\![c]\!](p) \mid c \in C, p \in P\} \tag{3}$$

(In this paper, we will use $\langle\!\langle \cdot \rangle\!\rangle$—typically with a distinguishing superscript, such as $\langle\!\langle C \rangle\!\rangle^{\mathbf{a}}$—to denote a semantics that operates over a set of programs.) $\langle\!\langle C \rangle\!\rangle^{\mathbf{a}}$, as defined in Equation (3), has been used implicitly in prior work as the basis of relatively incomplete Hoare-style logics for sets of programs without loops [18, 27]. Crucially, Equation (3) is the *least-expressive* semantics (formally, the *coarsest-grained* semantics [see Definition 3.10]) that captures single-input extensional properties of sets of programs. In other words, Equation (3) is the least-expressive semantics that can specify the set of

all possible output states that can be produced by running some program in C on a single given input state σ. Thus, it is reasonable to say that a useful verification technique for sets of programs should capture $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ at a minimum.

As with $[\![\cdot]\!]$, the semantics for sets $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ must also be capable of expressing multiple executions to achieve compositionality for loops. However, for sets of programs, $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ is surprisingly *too weak* to be compositional for loops: it cannot capture the semantics of multiple executions with sufficient precision. The first contribution of this paper is a formal proof that although the semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ is compositional for sets of loop-free programs (Theorem 4.3), $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ is not compositional for sets of programs that are allowed to contain loops (Theorem 4.4). The non-compositionality of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ is due to the fact that, for a set of loops while B do S, it is not always possible to derive $\langle\!\langle \text{while } B \text{ do } S \rangle\!\rangle^{\mathbf{a}}$ from only $\langle\!\langle B \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle S \rangle\!\rangle^{\mathbf{a}}$, because $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ cannot *precisely express multiple executions* of individual programs in B or S.

To see why $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ cannot precisely express multiple executions, suppose that the set $S = \{s_1, s_2\}$ contains two statements, and we are given two input states σ_i and σ_j. We want to combine $\langle\!\langle S \rangle\!\rangle^{\mathbf{a}}(\sigma_i)$ and $\langle\!\langle S \rangle\!\rangle^{\mathbf{a}}(\sigma_j)$ to produce the set $\{([\![s_1]\!](\sigma_i), [\![s_1]\!](\sigma_j)), ([\![s_2]\!](\sigma_i), [\![s_2]\!](\sigma_j))\}$, which tells us what can happen if we run each program in S on both σ_i and σ_j. In Equation (2), our semantics produced a single output state, so we directly combined the results into a tuple of states. Because $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ returns a *set* of output states, we must take the Cartesian product of $\langle\!\langle S \rangle\!\rangle^{\mathbf{a}}(\{\sigma_i\})$ and $\langle\!\langle S \rangle\!\rangle^{\mathbf{a}}(\{\sigma_j\})$ to get a set of tuples of states:

$$
\begin{aligned}
(\sigma_i, \sigma_j) \mapsto & \langle\!\langle S \rangle\!\rangle^{\mathbf{a}}(\{\sigma_i\}) \times \langle\!\langle S \rangle\!\rangle^{\mathbf{a}}(\{\sigma_j\}) \\
&= \{[\![s_1]\!](\sigma_i), [\![s_2]\!](\sigma_i)\} \times \{[\![s_1]\!](\sigma_j), [\![s_2]\!](\sigma_j)\} \\
&= \{([\![s_1]\!](\sigma_i), [\![s_1]\!](\sigma_j)), ([\![s_1]\!](\sigma_i), [\![s_2]\!](\sigma_j)), ([\![s_2]\!](\sigma_i), [\![s_1]\!](\sigma_j)), ([\![s_2]\!](\sigma_i), [\![s_2]\!](\sigma_j))\}
\end{aligned}
$$

The orange-only $([\![s_1]\!](\sigma_i), [\![s_1]\!](\sigma_j))$ and cyan-only $([\![s_2]\!](\sigma_i), [\![s_2]\!](\sigma_j))$ outcomes reflect the programs s_1 and s_2, but the outcomes with two colors—i.e., $([\![s_1]\!](\sigma_i), [\![s_2]\!](\sigma_j))$ and $([\![s_2]\!](\sigma_i), [\![s_1]\!](\sigma_j))$—do not correspond to any single program in S. The semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ does not track which programs produce which output states (i.e., it is program-agnostic), so it cannot tell whether or not a given pair of output states $(\sigma_i', \sigma_j') \in \langle\!\langle S \rangle\!\rangle^{\mathbf{a}}(\{\sigma_i\}) \times \langle\!\langle S \rangle\!\rangle^{\mathbf{a}}(\{\sigma_j\})$ is produced by the same program $s \in S$.

As a result, $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ is not compositional over sets of programs containing loops. In practice, this property means that, given $\langle\!\langle B \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle S \rangle\!\rangle^{\mathbf{a}}$, one cannot distinguish the set of loops while B do S from the singleton set containing the nondeterministic program while $\mathit{choose}(B)$ do $\mathit{choose}(S)$, which nondeterministically chooses an expression from B and program statement from S on every loop iteration. Note that, in the absence of loops, sets of programs can usually be modeled as a single nondeterministic program [16].

Compositional Denotational Semantics for Sets of Imperative Programs. To resolve the noncompositionality of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$, we must design a more expressive semantics that explicitly tracks the effects of programs on multiple inputs. Our next contribution is a "vector-state semantics," denoted by $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$, which meets the above goal by mapping sets of input vectors of states P to sets of output vectors that programs in C can produce when applied to each entry of an input vector:

$$\langle\!\langle C \rangle\!\rangle^{\mathbf{v}} = \lambda P.\{[[\![c]\!](\sigma_1), \cdots, [\![c]\!](\sigma_n)] \mid c \in C, [\sigma_1, \cdots, \sigma_n] \in P\}$$

Unlike the program-agnostic semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$, the vector-state semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ is capable of precisely expressing multiple executions. The key difference is that the vectorized input-states group inputs together, preventing the loss of precision that occurred by taking Cartesian products when we attempted to express multiple executions using $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$. As an illustration, taking the set of

statements $S = \{s_1, s_2\}$ as before, the vectorized semantics yields the following behavior:

$$\begin{aligned}(\sigma_1, \sigma_2) \mapsto &\langle\!\langle S \rangle\!\rangle^{\mathbf{v}}(\{[\sigma_1, \sigma_2]\}) \\ &= \{[[\![s]\!](\sigma_1), [\![s]\!](\sigma_2)] \mid s \in S\} \\ &= \{[[\![s_1]\!](\sigma_1), [\![s_1]\!](\sigma_2)], [[\![s_2]\!](\sigma_1), [\![s_2]\!](\sigma_2)]\}\end{aligned}$$

The main contribution of this paper is to show that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ is compositional, even over sets of loops (Theorem 5.4) and is the coarsest-grained (i.e., least expressive) compositional semantics that is stronger than $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ (Theorem 5.5). Thus, the vector-state semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ is the simplest semantics that can serve as the foundation of a compositional verification framework for sets of programs.

Taken together, our results show that a complete, compositional verification framework for sets of programs that can capture single-input extensional properties must capture at least $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$. This result sets a lower bound on the expressivity of complete, compositional verification frameworks for sets of programs. Whether a framework abandons completeness, abandons compositionality, removes loops from the language under consideration, or adopts the complexity of the vector-state semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ is a design choice to be made on a case-by-case basis.

Relationship to Program Synthesis. Readers familiar with program synthesis may wonder how single-input extensional properties tie in with program-synthesis problems, where specifications are often written over multiple examples, as in programming-by-example (PBE) [14, 32]. Single-input extensional properties correspond to single-example synthesis problems. Later, in §5.2, we discuss how the vector-state semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ can be viewed as an extension of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ that supports reasoning about finitely many inputs simultaneously, allowing the vector-state semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ to model multiple-input PBE problems as well.

Contributions. Although we focus on single-input extensional properties, our work presents a *general framework* for determining the minimum amount of information that must be tracked for a verification technique to be both compositional and relatively complete. In particular, we make the following contributions:

- We introduce formalisms for reasoning about denotational semantics of sets of programs, including a notion of *granularity* to capture the relative expressivity of various semantics (§3).
- We introduce two program-agnostic semantics: $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$. The $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ semantics captures exactly single-input extensional properties as described above, and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$ extends $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ with the ability to capture nontermination (§3).
- We show that, because $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$ cannot express multiple executions, neither semantics is compositional over sets of programs when loops are allowed (§4).
- We define two more-expressive semantics that operate over vector-states, $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$, and prove that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$ are the least-expressive *compositional* semantics that are at least as expressive as $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$, respectively (§5).
- We use the semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$ to design the *first two relatively complete, compositional Hoare-style logics for sets of programs*, a goal not achieved by previous work (§6) [18, 27].

Given a verification technique for sets of programs, our framework lets us analyze its expressivity and compositionality by considering the semantics defined by the set of facts that the technique can prove. In §2, we give two concrete examples, where we expose and resolve the limitations of two different verification techniques by considering their associated semantics.

§7 discusses related work. §8 concludes. The relative granularities of the semantics we introduce are depicted in Figure 1. An extended version of this paper is available on arXiv [20], which includes separate appendices that provides proofs of major theorems and rigorous formalizations

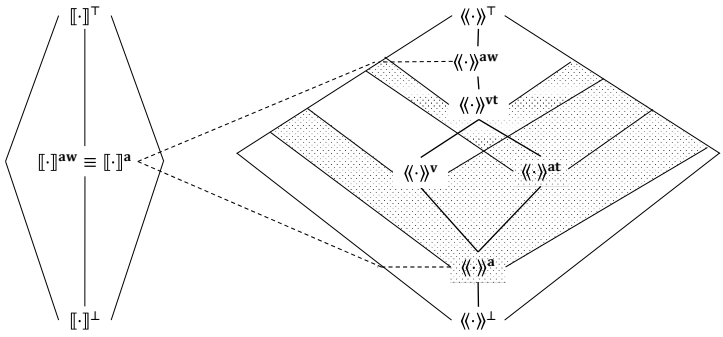


Fig. 1. Partial orders that relate the different semantics discussed in the paper using granularity (finest at the top). The lattice on the left is for semantics of single programs; the lattice on the right is for semantics of sets of programs. Each semantics in each lattice is allowed to have a different type. Semantics are ordered based on the granularity of the equivalence classes their denotations define (see Definition 3.10). The top semantics $[\![\cdot]\!]^{\top}$ and $\langle\!\langle\cdot\rangle\!\rangle^{\top}$ are the respective (syntactic) identity functions—i.e., $[\![\cdot]\!]^{\top} = \lambda c.c$ and $\langle\!\langle\cdot\rangle\!\rangle^{\top} = \lambda C.C$. The bottom semantics $[\![\cdot]\!]^{\bot}$ and $\langle\!\langle\cdot\rangle\!\rangle^{\bot}$ indicate constant semantics that return the same element for all (sets of) programs. The partial order for the semantics of sets of programs has finer structure than the partial order for the semantics of single programs: there exist elements between $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{aw}}$ and $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{a}}$. Moreover, between the divergence-agnostic semantics $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{v}}$ and between the divergence-aware semantics $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{at}}$ and $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{vt}}$, there exist two overlapping crosshatched zones in which all semantics are noncompositional.

of some concepts that are simplified in this paper for presentation. We have clearly marked places where referring to the full version will provide more details.

2 From Verification Frameworks to Semantics and Back

In this section, we demonstrate the utility of our theory by deriving semantics for two verification techniques for sets of programs: program logics [18, 27] for sets of programs (§2.1) and Hoare-style reasoning about an interpreter on sets of programs (§2.2). In particular, our theory can (*i*) prove non-obvious tradeoffs between expressivity and completeness for these verification strategies, and (*ii*) inspire new complete versions of these techniques using the vector-state semantics $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{v}}$.

We note that in this paper we use the word 'program' to refer to not only statements, but also Boolean and integer expressions—e.g., both the statement $x := x + 3$ and the expression $x + 3$ are referred to as programs. Later, when we define semantics for programs (Definition 3.3) and sets of programs (Definition 3.7), we mean that the semantics is not limited to statements.

2.1 A Program Logic for Sets of Programs

A program logic (e.g., Hoare Logic [15]) is a proof system for reasoning about program properties. In this case study, the goal is to design a program logic L for reasoning about single-input extensional properties of sets of programs. Previous attempts at designing such a logic [18, 27] have resulted in incomplete logics, due to implicitly basing the logics on $\langle\!\langle C\rangle\!\rangle^{\mathbf{a}}$ from §1; we illustrate why exactly this is the case, providing an outline here and more formal details in §6.

To design such a logic, one needs to define (*i*) some kind of meaningful judgments ($J(C)$) about sets of programs C, and (*ii*) inference rules that allow the logic to prove new judgments from known judgments or axioms, for example,

$$\frac{\vdash J_1(B) \qquad \vdash J_2(S)}{\vdash J_3(\text{while } B \text{ do } S)} \tag{4}$$

where B is a set of Boolean expressions, S is a set of statements. Critically, the inference rules should be compositional: for example, any valid judgment about the set while B do S should be derivable from valid judgments about B and S, as in Equation (4).

Existing Logics' Judgments are Noncompositional. In the logic proposed by Nagy et al. [27], judgments are triples, denoted by $\{\!|P|\!\}\ C\ \{\!|Q|\!\}$, where C is a set of programs, and P and Q are sets of states. In terms of semantics, the meaning of this kind of triple can be written as follows:

$$\{\!|P|\!\}^{\mathtt{a}}\ C\ \{\!|Q|\!\}^{\mathtt{a}} \triangleq \langle\!\langle C\rangle\!\rangle^{\mathtt{a}}(P) \subseteq Q \tag{5}$$

Other prior work interprets triples slightly differently but relies on the same intuition [18].

Observe that a denotational semantics over a set of programs is just a map that assigns to each set of programs C a mathematical object associated with C (i.e., its denotation). Thus, one can also define a "logic-based" semantics $\langle\!\langle\cdot\rangle\!\rangle^{La}$ that associates with C the set of valid judgments about C:

$$\langle\!\langle C\rangle\!\rangle^{La} \triangleq \{(P,Q) \mid \{\!|P|\!\}^{\mathtt{a}}\ C\ \{\!|Q|\!\}^{\mathtt{a}} \text{ holds}\} \tag{6}$$

Under this definition, the logic-based semantics $\langle\!\langle\cdot\rangle\!\rangle^{La}$ and the program-agnostic semantics $\langle\!\langle\cdot\rangle\!\rangle^{\mathtt{a}}$ carry the same information: $\langle\!\langle C\rangle\!\rangle^{La}$ can be derived from $\langle\!\langle C\rangle\!\rangle^{\mathtt{a}}$ and vice versa.

Previous work has failed to find relatively complete, compositional inference rules for a logic over such triples. Our theory proves that such rules do not exist: because both the program-agnostic semantics $\langle\!\langle\cdot\rangle\!\rangle^{\mathtt{a}}$ and the logic-based semantics $\langle\!\langle\cdot\rangle\!\rangle^{La}$ carry the same information, noncompositionality of the program-agnostic semantics $\langle\!\langle\cdot\rangle\!\rangle^{\mathtt{a}}$ (see Theorem 4.4) implies noncompositionality of the logic-based semantics $\langle\!\langle\cdot\rangle\!\rangle^{La}$. The non-compositionality of $\langle\!\langle\cdot\rangle\!\rangle^{La}$ implies that there is no complete system of compositional inference rules for such triples (see Theorem 6.3)!

More concretely, there are sets of Boolean guards B and loop bodies S for which there exist true triples $\{\!|P|\!\}^{\mathtt{a}}$ while B do S $\{\!|Q|\!\}^{\mathtt{a}}$ that cannot be proven from any true triples about B and S. For example, if $B = \{x == 1, \neg(x == 1)\}$ then we cannot compositionally prove the triple

$$\{\!|x = 0|\!\}^{\mathtt{a}}\ (\text{while } \{x == 1, \neg(x == 1)\} \text{ do } x := x + 1)\ \{\!|x = 0 \vee x = 1|\!\}^{\mathtt{a}} \tag{7}$$

Without the ability to track the effects of programs on multiple inputs, the set of guards B is indistinguishable from the set of guards $B' = \{x == 2, \neg(x == 2)\}$, because both sets can produce true and false on any input state (i.e., $\langle\!\langle B\rangle\!\rangle^{\mathtt{a}} = \langle\!\langle B'\rangle\!\rangle^{\mathtt{a}}$ so $\langle\!\langle B\rangle\!\rangle^{La} = \langle\!\langle B'\rangle\!\rangle^{La}$). In other words, every triple that is true of B is true of B'. However, changing the set of guards from B to B' invalidates the triple in Equation (7) (i.e., $\{\!|x = 0|\!\}^{\mathtt{a}}$ (while B' do $x := x + 1$) $\{\!|x = 0 \vee x = 1|\!\}^{\mathtt{a}}$ is false because $x = 2$ is a possible outcome). A compositional logic that cannot distinguish between B and B' also cannot distinguish between while B do $x := x + 1$ and while B' do $x := x + 1$, so no compositional logic can prove the triple given in Equation (7).

Compositional Triples via Vector-State Semantics. To design a compositional logic, we need to strengthen our triples. Our new triples must yield a semantics that is at least as expressive as the vector-state semantics $\langle\!\langle\cdot\rangle\!\rangle^{\mathtt{v}}$—i.e., the coarsest compositional semantics stronger than the program-agnostic semantics $\langle\!\langle\cdot\rangle\!\rangle^{\mathtt{a}}$—otherwise the resulting semantics will be noncompositional like $\langle\!\langle\cdot\rangle\!\rangle^{La}$ (Theorem 5.5). Following Equation (5), we can take P and Q to be sets of vector states, defining triples by Equation (8) and inducing the vectorized logic-based semantics in Equation (9):

$$\{\!|P|\!\}^{\mathtt{v}}\ C\ \{\!|Q|\!\}^{\mathtt{v}} \triangleq \langle\!\langle C\rangle\!\rangle^{\mathtt{v}}(P) \subseteq Q \tag{8}$$

$$\langle\!\langle C\rangle\!\rangle^{Lv} \triangleq \{(P,Q) \mid \{\!|P|\!\}^{\mathtt{v}}\ C\ \{\!|Q|\!\}^{\mathtt{v}} \text{ holds}\} \tag{9}$$

With this definition, the vectorized logic-based semantics $\langle\!\langle\cdot\rangle\!\rangle^{Lv}$ is equivalent to the vector-state semantics $\langle\!\langle\cdot\rangle\!\rangle^{\mathtt{v}}$, and compositional inference rules follow directly from our constructive proof that $\langle\!\langle\cdot\rangle\!\rangle^{\mathtt{v}}$ is compositional (Theorem 5.4). Moreover, the above definition is the simplest notion of

triples that permits a compositional inference rule for sets of while loops (given in §6.1). To capture hyperproperties, prior work has proposed similar—but different—semantics over vectors, but has only produced incomplete program logics [18, 27].[2]

2.2 Analyzing Sets of Programs Through an Interpreter

Readers might feel that compositionality is an issue only because we are defining our program logic over sets of programs. One way to avoid lifting our reasoning to sets is to instead perform a Hoare-style analysis of an interpreter (e.g., `interpret` in Figure 2).[3] Rather than analyzing C directly as in §2.1, one would carry out an inductive argument about the behavior of `interpret` over C. The hope is that because `interpret` is a single program, an analysis that is compositional over `interpret` would avoid the complexity of an analysis that is compositional over C as in §2.1.

```
1  State interpret (t: Prog, x: State) {
2    match t with
3      ...
4      case While b do s: {
5        State r;
6        State b1 = interpret(b, x);
7        if (b1.bt) {
8          State x1 = interpret(s, x);
9          r = interpret(While b do s, x1);
10       }
11       else {
12         r = x;
13       }
14       return r;
15     }
16     ...
17 }
```

Fig. 2. An interpreter for the language described in Figure 3. The symbol `bt` indicates a reserved variable in `State` that stores the result of evaluating Boolean expressions.

Using the theory presented in this paper, we can *prove* that this approach does not avoid the hardness of compositional reasoning over sets of programs. In particular, we consider a Hoare-style analysis of the interpreter `interpret` where the preconditions are limited to "Cartesian" predicates: predicates of the form $\varphi_1(t) \wedge \varphi_2(x)$, where φ_1 is forbidden from referencing the state x and φ_2 is forbidden from referencing the program t. Maintaining this separation is necessary to prevent one from writing preconditions that reason about interpretation explicitly, which would defeat the purpose of analyzing an interpreter.

In this section, we demonstrate that Hoare-style analysis limited to Cartesian preconditions induces a semantics $\langle\!\langle\cdot\rangle\!\rangle^{\texttt{interpret}_a}$ similar to $\langle\!\langle\cdot\rangle\!\rangle^{La}$. The completeness of Hoare-style analysis limited to Cartesian preconditions would imply compositionality of the induced semantics $\langle\!\langle\cdot\rangle\!\rangle^{\texttt{interpret}_a}$, but $\langle\!\langle\cdot\rangle\!\rangle^{\texttt{interpret}_a}$ is not compositional. Thus, such an analysis over recursive interpreters of the type $\texttt{interpret} : \mathsf{Prog}\times\mathsf{State} \to \mathsf{State}$ is bound to be incomplete. On the other hand, the analysis may be complete if we lift the interpreter to act on vector-states (i.e., $\texttt{interpret} : \mathsf{Prog} \times \mathsf{State}^* \to \mathsf{State}^*$), because the existence of a compositional vector-state semantics as mentioned in §1 implies that a compositional analysis must exist. We sketch the main idea in this section; a rigorous formalization may be found in the extended version of the paper [20].

Noncompositionality of a Simple Interpreter. To analyze the extensional properties of a syntactically defined set of programs C, one can compute $[\![\texttt{interpret}]\!](C \times P)$ given an arbitrary set of inputs P. This approach is equivalent to deriving a Hoare triple of the form:

$$\{t \in C \wedge x \in P\}\ y = \texttt{interpret}(t, x)\ \{y \in Q\}.$$

For example, to understand the possible outputs of the set of loops W from Example 1.1 on the input $(x \mapsto 0)$, one can prove

$$\{t \in W \wedge x = 0\}\ y = \texttt{interpret}(t, x)\ \{y = 0 \pmod 2\}.$$

[2]There is a flaw in the proof of completeness given in Kim et al. [18], which has been discussed with the authors.
[3]Because `interpret` is a recursive function, one technically needs to consider proofs of infinite size or apply recursive Hoare logic [29]. These approaches require additional steps (e.g., introducing an induction hypothesis). We ignore these details to simplify the presentation; the extended version of the paper [20] gives a more thorough treatment.

However, reasoning about such triples can be complicated. Cartesian preconditions of the form $t \in C \wedge x \in P$ can check membership of an input (t, x) in the precondition by checking $t \in C$ and $x \in P$ *independently*. Critically, because C is defined syntactically and P cannot reference t, one cannot relate the syntax of t to its semantics $[\![t]\!]$ inside the precondition. Observe that Cartesian preconditions have a direct correspondence with the program-agnostic semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathtt{a}}$: a triple $\{t \in C \wedge x \in P\}\,\mathtt{interpret}(t, x)\,\{Q\}$ holds iff $\langle\!\langle C \rangle\!\rangle^{\mathtt{a}}(P) \subseteq Q$. Unfortunately, when $\mathtt{t}$ contains a loop, such Cartesian preconditions often cannot be proven without using substantially more complicated "non-Cartesian" preconditions that relate $\mathtt{t}$ and $\mathtt{x}$. In this situation, we no longer have a direct correspondence between Hoare triples and $\langle\!\langle \cdot \rangle\!\rangle^{\mathtt{a}}$, which corresponds to reasoning about complex properties outside the program-agnostic semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathtt{a}}$.

For example, consider proving the triple $\{t \in \mathtt{while}\ B\ \mathtt{do}\ S \wedge x \in P\}\, y = \mathtt{interpret}(t, x)\,\{y \in Q\}$. Analysis is straightforward until Line 9 of Figure 2, emphasized in red, where the semantics of a loop requires that the same $\mathtt{b} \in B$ and $\mathtt{s} \in S$ be *repeated* for each iteration of a loop. Upon reaching Line 9, there is a relation formed with the input state and the program: $\mathtt{x}_1$ is created using a specific combination of $\mathtt{b} \in B$ and $\mathtt{s} \in S$, such that $[\![\mathtt{b}]\!](\mathtt{x}_0) = \mathtt{true}$ and $[\![\mathtt{s}]\!](\mathtt{x}_0) = \mathtt{x}_1$, for some input state $\mathtt{x}_0$. Thus, the precondition over which the recursive call to `interpret` must operate becomes:

$$(\mathtt{t}, \mathtt{x}) \in \{(\mathtt{while\ b\ do\ s}, \mathtt{x}_1) \mid \mathtt{b} \in B \wedge \mathtt{s} \in S \wedge (\exists \mathtt{x}_0 \in P. [\![\mathtt{b}]\!](\mathtt{x}_0) = \mathtt{true} \wedge [\![\mathtt{s}]\!](\mathtt{x}_0) = \mathtt{x}_1)\} \tag{10}$$

Although $t \in \mathtt{while}\ B\ \mathtt{do}\ S \wedge x \in P$ was initially a Cartesian precondition, Equation (10) is not—the additional condition $\exists \mathtt{x}_0 \in P. [\![\mathtt{b}]\!](\mathtt{x}_0) = \mathtt{true} \wedge [\![\mathtt{s}]\!](\mathtt{x}_0) = \mathtt{x}_1$ forces our precondition to reason about both the syntax of our programs (e.g., $s \in S$) as well as their semantic effects (i.e., $[\![\mathtt{s}]\!](\mathtt{x}_0) = \mathtt{x}_1$). This entanglement requires a complex notion of 'interpretation' $[\![\cdot]\!]$ in our precondition that one would expect analysis of an interpreter to avoid.

Reasoning about non-Cartesian preconditions is actually an unavoidable feature of every correct recursive implementation of $\mathtt{interpret} : \mathsf{Prog} \times \mathsf{State} \to \mathsf{State}$. To understand why this observation holds, consider an interpreter-based semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathtt{interpret}_a}$ akin to $\langle\!\langle \cdot \rangle\!\rangle^{La}$ in §2.1:

$$\langle\!\langle C \rangle\!\rangle^{\mathtt{interpret}_a} = \{(P, Q) \mid \{\mathtt{t} \in\ C \wedge \mathtt{x} \in P\}\ y = \mathtt{interpret}(t, x)\ \{Q\} \text{ is true}\} \tag{11}$$

If one could prove every Cartesian triple with a Hoare-logic proof using only Cartesian triples, one would be able to derive a compositional characterization of $\langle\!\langle \cdot \rangle\!\rangle^{\mathtt{interpret}_a}$. However, $\langle\!\langle \cdot \rangle\!\rangle^{\mathtt{interpret}_a}$ is noncompositional because it is equivalent to the noncompositional program-agnostic semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathtt{a}}$, just like the logic-based semantics $\langle\!\langle \cdot \rangle\!\rangle^{La}$. Consequently, reasoning about non-Cartesian preconditions like Equation (10) is unavoidable for *any* recursive implementation of $\mathtt{interpret} : \mathsf{Prog} \times \mathsf{State} \to \mathsf{State}$ (see the extended version of the paper [20] for a formal proof).

Complete Interpreter Analysis. To precisely reason about all facts of the form $[\![\mathtt{interpret}]\!](C \times P)$ with Hoare logic, one can either (*i*) accept the complexity of non-Cartesian preconditions or (*ii*) extend the signature of `interpret` so that $\mathtt{x}$ is a vector-state rather than a single input (i.e., $\mathtt{interpret}_v : \mathsf{Prog} \times \mathsf{State}^* \to \mathsf{State}^*$). In the second case, this new interpreter induces a semantics:

$$\langle\!\langle C \rangle\!\rangle^{\mathtt{interpret}_v} = \{(P, Q) \mid \{\mathtt{t} \in\ C \wedge \mathtt{x} \in P\}\ \mathtt{interpret}_v(t, x)\ \{Q\} \text{ is true}\}, \tag{12}$$

which is equivalent to the vector-state semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathtt{v}}$. Because $\langle\!\langle \cdot \rangle\!\rangle^{\mathtt{v}}$ is compositional (Theorem 5.4), there will also exist a Hoare-style analysis of $\mathtt{interpret}_v$ that stays within the realm of Cartesian preconditions—at the cost of introducing vector-states. Alternatively, one could forego completeness in exchange for a simple and compositional analysis. For example, in Equation (10), one could disallow conditions such as $\exists \mathtt{x}_0 \in P. [\![\mathtt{b}]\!](\mathtt{x}_0) = \mathtt{true} \wedge [\![\mathtt{s}]\!](\mathtt{x}_0) = \mathtt{x}_1$ and only consider preconditions that are Cartesian. This approach would enlarge the possible input states to subsequent calls of `interpret`, resulting in an imprecise, but still sound, analysis.

$$
\begin{array}{lcl}
\mathit{Stmt} & ::= & \mathit{Var} := \mathit{Exp} \mid \mathit{Stmt}; \mathit{Stmt} \mid \\
 & & \quad \text{if } \mathit{BExp} \text{ then } \mathit{Stmt} \text{ else } \mathit{Stmt} \mid \text{while } \mathit{BExp} \text{ do } \mathit{Stmt} \\
\mathit{Exp} & ::= & \mathit{Var} \mid 1 \mid 0 \mid \mathit{Exp} + \mathit{Exp} \mid \mathit{Exp} - \mathit{Exp} \\
\mathit{BExp} & ::= & \mathrm{t} \mid \mathrm{f} \mid \neg \mathit{BExp} \mid \mathit{BExp} \wedge \mathit{BExp} \mid \mathit{Exp} < \mathit{Exp} \mid \mathit{Exp} == \mathit{Exp} \\
\mathit{Var} & ::= & x \mid x_1 \mid x_2 \mid \cdots
\end{array}
$$

Fig. 3. The grammar G_{imp}, which defines the entire set of terms considered in this paper.

3 Semantics of Sets of Programs

Having illustrated the implications of our semantic theory for the hardness of practical verification techniques, we now turn to formalizing the ideas presented in Sections 1 and 2. We start by formally defining the sets of programs we are interested in (§3.1), proceed to formalize what a semantics over (sets of) programs is mathematically (§3.2 and §3.3), then define a notion of *granularity* which allows us to formally compare the expressive power of different semantics (§ 3.4).

3.1 Inductively Defined Sets of Programs

In this paper, we consider a small deterministic imperative language that contains loops, defined via G_{imp} in Figure 3. In particular, we are interested in sets of programs (i.e., subsets of G_{imp}) that are inductively defined by a regular tree grammar.

Definition 3.1 (Regular Tree Grammar). A *(typed) regular tree grammar* (RTG) is a tuple $G = (\mathcal{N}, \mathcal{A}, \mathit{Start}, a, \delta)$, where $\mathcal{N}$ is a finite set of nonterminals; $\mathcal{A}$ is a ranked alphabet; $\mathit{Start} \in \mathcal{N}$ is an initial nonterminal; a is a type assignment that gives types for members of $\mathcal{A} \cup \mathcal{N}$; and δ is a finite set of productions of the form $N_0 \to \alpha^{(i)}(N_1, ..., N_i)$, where for $0 \leq j \leq i$, each $N_j \in \mathcal{N}$ is a nonterminal such that if $a_{\alpha^{(i)}} = (\tau_0, \tau_1, ..., \tau_i)$ then $a_{N_j} = \tau_j$.

Definition 3.2 (Inductively Defined Set of Programs). Given a set of programs C and an RTG G with start nonterminal N, we say that C is *inductively defined by* G if $C = L(N)$.

For this paper, the alphabet of an RTG consists of constructors for each of the constructs of G_{imp}. For simplicity of presentation, we elide the arity of constructors and use infix notation (e.g., $x := 0$, rather than $:=^{(2)} (x^{(0)}, 0^{(0)})$)) and also allow inlining of nonterminals (e.g., $S' ::= x_1 := x_1$ instead of the three productions $S' ::= V_x := E_x$, $E_x ::= V_x$, and $V_x ::= x_1$). We assume grammars use productions that differ from the ones in G_{imp} only due to the nonterminal name (i.e., the constructor $\alpha^{(n)}$ of a production must be from G_{imp} and have the same arity). We call a production $N_0 ::= \alpha^{(i)}(N_1, \cdots, N_i)$ *valid with respect to* G_{imp} if replacing each nonterminal N_j with the nonterminal of the same type in the set $\{\mathit{Stmt}, \mathit{Exp}, \mathit{BExp}\}$ yields a production in G_{imp} (e.g., production $S_1 ::= S_2; S_3$ is valid with respect to G_{imp} because $\mathit{Stmt} ::= \mathit{Stmt}; \mathit{Stmt}$ is a production in G_{imp}).

Given a nonterminal N of a grammar G, we will use N itself as shorthand to refer to $L(N)$ (the set of terms derivable via the grammar G from the nonterminal N). Thus, *Stmt* refers also to the set of statements derivable from *Stmt*. Given a term c, we define $\mathit{vars}(c)$ to be the set of variables appearing in c—e.g., $\mathit{vars}(x_1 := x_2) = \{x_1, x_2\}$. For a set of programs C, we say $\mathit{vars}(C) = \bigcup_{c \in C} \mathit{vars}(c)$.

We use the (bolded) name $\mathbf{Stmt} = \{S \mid S \in 2^{\mathit{Stmt}} \wedge |\mathit{vars}(S)| < \infty\}$ to denote the set of all sets of statements S that each contain finitely many variables. **Exp** and **BExp** are defined analogously, for expressions and Boolean expressions, respectively. This paper only considers sets of programs C where $|\mathit{vars}(C)| < \infty$—sets whose programs reference an infinite number of variables are rare and out of scope for this paper.

3.2 Semantics of Programs

We now formalize the definition of a *semantics* of a *single* program.

Definition 3.3 (Semantics of a Program). Let $A(\tau_{Stmt}, \tau_{Exp}, \tau_{BExp})$ be a 3-tuple with an associated name A where τ_{Stmt}, τ_{Exp}, and τ_{BExp} are the types of the denotations that the semantics assigns to programs of type *Stmt*, *Exp*, and *BExp*, respectively. Then a *semantics over programs* $[\![\cdot]\!]^A$ is a 3-tuple $[\![\cdot]\!]^A = ([\![\cdot]\!]^A_{Stmt}, [\![\cdot]\!]^A_{Exp}, [\![\cdot]\!]^A_{BExp})$, where: $[\![\cdot]\!]^A_{Stmt} : Stmt \to \tau_{Stmt}$, $[\![\cdot]\!]^A_{Exp} : Exp \to \tau_{Exp}$, and $[\![\cdot]\!]^A_{BExp} : BExp \to \tau_{BExp}$. We call $A(\tau_{Stmt}, \tau_{Exp}, \tau_{BExp})$ the *signature* of $[\![\cdot]\!]^A$.

In this paper, we will only be defining semantics where $\tau_{Stmt} = \tau_{Exp} = \tau_{BExp}$, by interpreting integer and Boolean expressions as state transformers as well. Specifically, an expression $e \in Exp$ is considered to update a reserved variable e_t instead of returning an integer value; Boolean expressions update a separate reserved variable b_t in a similar manner. This formalism makes it easier to relate the result of an expression with its input state, which will become convenient later when constructing compositional characterizations for sets of programs, as in Figure 5. When $\tau_{Stmt} = \tau_{Exp} = \tau_{BExp} = \tau$, we abbreviate the signature $A(\tau_{Stmt}, \tau_{Exp}, \tau_{BExp})$ as $A(\tau)$, and we often omit subscripts for *Stmt*, *Exp*, and *BExp* when they are clear from context.

Definition 3.3 allows us to consider multiple different semantics for a program by changing the signature $A(\tau)$ and the definition of $[\![\cdot]\!]^A$. We will first define a collecting big-step semantics for programs in G_{imp}, called $[\![\cdot]\!]^a$, which ignores nontermination and is such that $\tau{=}2^{\text{State}} \to 2^{\text{State}}$. We will then introduce a variant of $[\![\cdot]\!]^a$, denoted by $[\![\cdot]\!]^{at}$, which handles nontermination explicitly by introducing a non-terminating value $\lightning$. We start by defining program states.

*Definition 3.4 (*State *and* DState*).* A *state* σ is defined as a tuple (h, e_t, b_t), where (*i*) h is a map $h : Var \to \mathbb{Z}$ from variables in *Var* to integer values, (*ii*) $e_t \in Z$, and (*iii*) $b_t \in \{\mathrm{t}, \mathrm{f}\}$. We denote the set of all possible states as State. For a state $\sigma = (h, e_t, b_t)$, we write $\sigma[x_j]$ to denote $h(x_j)$, $\sigma[e_t]$ to denote e_t, and $\sigma[b_t]$ to denote b_t. DState denotes the set of all possible states including the divergence behavior $\lightning$, and we define it as DState = State $\cup$ $\{\lightning\}$.

In Definition 3.4, h acts as the 'usual' state that maps variables to values—note that h is defined over *Var*, the entire set of possible variables in G_{imp}, as opposed to a subset of variables that occur in the program; this convention simplifies the definitions of our semantics as well the proofs in §5. When we only care about the values of specific variables, we write states as, for example, $(x_1 \mapsto 4, e_t \mapsto 3)$ to denote any $\sigma \in$ State where $\sigma[x_1] = 4$ and $\sigma[e_t] = 3$. e_t and b_t act as auxiliary variables that "store" the results of evaluating an expression and Boolean expression, respectively.

The value $\lightning$ denotes nontermination (i.e., diverging behavior) and is used in the semantics $[\![\cdot]\!]^{at}$. For example, $[\![\text{while } x = 1 \text{ do } x := x]\!]^{at}(\{[x \mapsto 1]\}) = \{\lightning\}$. To make $[\![c]\!]^{at}$ total, we assume $[\![c]\!]^{at}(\{\lightning\}) = \{\lightning\}$ for every program c.

We are now ready to formally define a big-step collecting semantics for terms in G_{imp}.[4] This semantics is the single-program version of the "program-agnostic" semantics $\langle\!\langle\cdot\rangle\!\rangle^{\mathtt{a}}$ from §1, which map sets of inputs to sets of outputs. We call this semantics *program-agnostic* because it does not attempt to track how the program maps inputs to outputs. This distinction will become clearer when we switch to talking about sets of programs in §3.3.

Definition 3.5 (Program-Agnostic Semantics of a Program). The *program-agnostic* semantics for single programs $[\![\cdot]\!]^a$ over the signature $a(2^{\text{State}} \to 2^{\text{State}})$ is defined as $[\![c]\!]^a(X) = \bigcup_{\sigma \in X} [\![c]\!]^a(\{\sigma\})$, where the rules for computing $[\![c]\!]^a(\{\sigma\})$ are defined in Figure 4. Similarly, the program-agnostic

[4] We consider collecting semantics because the focus on this paper is on the semantics of sets of programs, which do not admit a compositional characterization when limited to single-state input and outputs.

$$[\![0]\!]^a(\sigma) = \sigma[e_t \mapsto 0] \quad [\![1]\!]^a(\sigma) = \sigma[e_t \mapsto 1] \quad [\![x_j]\!]^a(\sigma) = \sigma[e_t \mapsto \sigma[x_j]] \quad [\![\mathrm{t}]\!]^a(\sigma) = \sigma[b_t \mapsto \mathrm{t}]$$

$$[\![\mathrm{f}]\!]^a(\sigma) = \sigma[b_t \mapsto \mathrm{f}] \quad [\![\neg b]\!]^a(\sigma) = \sigma[b_t \mapsto \neg([\![b]\!]^a(\sigma)[b_t])]$$

$$[\![t_1 \odot t_2]\!]^a(\sigma) = \sigma[r_t \mapsto \sigma_1[q_t] \odot \sigma_2[q_t]], \text{where } \sigma_1 = [\![t_1]\!]^a(\sigma) \text{ and } \sigma_2 = [\![t_2]\!]^a(\sigma)$$

$$[\![x := e]\!]^a(\sigma) = \sigma[x \mapsto ([\![e]\!]^a(\sigma)[e_t])] \qquad [\![s_1; s_2]\!]^a(\sigma) = [\![s_2]\!]^a([\![s_1]\!]^a(\sigma))$$

$$[\![\text{if } b \text{ then } s_1 \text{ else } s_2]\!]^a(\sigma) = \text{if } [\![b]\!]^a(\sigma) \text{ then } [\![s_1]\!]^a(\sigma) \text{ else } [\![s_2]\!]^a(\sigma)$$

$$[\![\text{while } b \text{ do } s]\!]^a(\sigma) = \text{if } [\![b]\!]^a(\sigma) \text{ then } [\![\text{while } b \text{ do } s]\!]^a([\![s]\!]^a(\sigma)) \text{ else } \sigma$$

$$[\![\text{while } b \text{ do } s]\!]^{at}(\sigma) = \begin{cases} [\![\text{while } b \text{ do } s]\!]^a(\sigma) & [\![\text{while } b \text{ do } s]\!]^a(\sigma) \neq \emptyset \\ \{\uparrow\} & \text{else} \end{cases}$$

Fig. 4. Definition of $[\![\cdot]\!]^a$ and $[\![\cdot]\!]^{at}$ where $b \in \mathbf{BExp}$, $e, e_1, e_2 \in \mathbf{Exp}$, and $s, s_1, s_2 \in \mathbf{Stmt}$. Recall that e_t and b_t are reserved variables that hold the results of evaluated arithmetic and Boolean expressions, respectively. The symbol $\odot$ denotes a binary operator in G_{imp} (either $+, -, <, ==$, or $\wedge$), t_1 and t_2 refer to expressions of the appropriate types, and r_t and q_t refer to auxiliary variables of the appropriate type. $[\![\cdot]\!]^a$ and $[\![\cdot]\!]^{at}$ are respectively of type $2^{\mathrm{State}} \to 2^{\mathrm{State}}$ and $2^{\mathrm{DState}} \to 2^{\mathrm{DState}}$; this figure only lists definitions for singleton inputs for readability. The definition of $[\![\cdot]\!]^{at}$ is identical to that of $[\![\cdot]\!]^a$ for all constructors other than while. The semantics of $[\![\text{while } b \text{ do } s]\!]^a$ is taken to be the least fixed point of the given equation, as is standard.

semantics for single programs $[\![\cdot]\!]^{at}$ over the signature $at(2^{\mathrm{DState}} \to 2^{\mathrm{DState}})$ is defined as $[\![c]\!]^{at}(X) = \bigcup_{\sigma \in X} [\![c]\!]^{at}(\{\sigma\})$ following the rules defined in Figure 4.

In Figure 4 and other parts of this paper, for a program c and a single state $\sigma \in \mathrm{DState}$, we write $[\![c]\!]^a(\sigma)$ to mean $[\![c]\!]^a(\{\sigma\})$ (the same holds for other kinds of semantics defined in this paper).

The difference between $[\![\cdot]\!]^a$ and $[\![\cdot]\!]^{at}$ lies in their treatment of divergence: $[\![\text{while t do } x := x]\!]^a(\sigma) = \emptyset$ whereas $[\![\text{while t do } x := x]\!]^{at}(\sigma) = \{\uparrow\}$. For single programs, these two semantics are equivalent in their distinguishing power: given two programs c_1 and c_2, $[\![c_1]\!]^a = [\![c_2]\!]^a$ iff $[\![c_1]\!]^{at} = [\![c_2]\!]^{at}$. However, the difference in the treatment of divergence will create a difference when considering semantics for sets of programs, and we will see that explicitly tracking divergence as in $[\![\cdot]\!]^{at}$ is necessary to precisely capture how different programs can diverge.

Next, we define the program-aware semantics $[\![\cdot]\!]^{aw}$, whose denotations are *sets of* functions, as opposed to functions between sets of states (like $[\![\cdot]\!]^a$). This distinction is irrelevant for single programs (because there is only one function) but becomes crucial later for sets of programs.

Definition 3.6 (Program-Aware Semantics ($[\![\cdot]\!]^{aw}$)). The *program-aware semantics* for programs $[\![\cdot]\!]^{aw}$ over the signature $aw(2^{2^{\mathrm{DState}} \to 2^{\mathrm{DState}}})$ is defined so that, for each given program c,

$$[\![c]\!]^{aw} \triangleq \{[\![c]\!]^{at}\}.$$

For example, $[\![x := 5]\!]^{aw} = \{\lambda\sigma.\sigma[x \mapsto 5]\}$ because $[\![x := 5]\!]^{at} = \lambda\sigma.\sigma[x \mapsto 5]$. While $[\![\cdot]\!]^{at}$ and $[\![\cdot]\!]^{aw}$ contain the same information for single programs, they are different mathematical objects. When generalizing these semantics to sets of programs, this difference in representation allows $[\![\cdot]\!]^{aw}$ to keep the semantics of different programs separate, whereas $[\![\cdot]\!]^{at}$ cannot do the same.

3.3 Semantics of Sets of Programs

In this section, we extend Definition 3.3 to define the semantics of a *set of programs*.

Definition 3.7 (Semantics of a Set of Programs). Let $A(\tau_{Stmt}, \tau_{Exp}, \tau_{BExp})$ be a 3-tuple with an associated name A where τ_{Stmt}, τ_{Exp}, and τ_{BExp} are the types of the denotations that the semantics assigns to sets of programs of type **Stmt**, **Exp**, and **BExp**, respectively. Then a *semantics over a set of programs* $\langle\!\langle\cdot\rangle\!\rangle^A$ is defined as the 3-tuple $\langle\!\langle\cdot\rangle\!\rangle^A = (\langle\!\langle\cdot\rangle\!\rangle^A_{Stmt}, \langle\!\langle\cdot\rangle\!\rangle^A_{Exp}, \langle\!\langle\cdot\rangle\!\rangle^A_{BExp})$, where: $\langle\!\langle\cdot\rangle\!\rangle^A_{Stmt} : \mathbf{Stmt} \to \tau_{Stmt}$, $\langle\!\langle\cdot\rangle\!\rangle^A_{Exp} : \mathbf{Exp} \to \tau_{Exp}$, and $\langle\!\langle\cdot\rangle\!\rangle^A_{BExp} : \mathbf{BExp} \to \tau_{BExp}$. We call A the *signature* of $\langle\!\langle\cdot\rangle\!\rangle^A$, and when $\tau_{Stmt} = \tau_{Exp} = \tau_{BExp}$, we write the signature as simply $A(\tau)$.[5] We omit the subscripts *Stmt*, *Exp*, *BExp* if clear from context.

Next, we extend the three semantics over programs ($[\![\cdot]\!]^a$ and $[\![\cdot]\!]^{at}$ from Definition 3.5 and $[\![\cdot]\!]^{aw}$ from Definition 3.6) to semantics over sets of programs ($\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{a}}$, $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{at}}$, and $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{aw}}$, respectively), by taking the union of the semantics of all programs in the sets.

Definition 3.8 (Program-Agnostic and Program-Aware Semantics of a Set of Programs). For a given set of programs C and an input set of states X, each of the semantics $[\![\cdot]\!]^A$ for $A = a, at, aw$ can be lifted to a semantics over sets of programs $\langle\!\langle\cdot\rangle\!\rangle^A$ over the signature $A(2^{\text{State}} \to 2^{\text{State}})$, defined as:

$$\langle\!\langle C\rangle\!\rangle^A(X) = \bigcup_{c \in C} [\![c]\!]^A(X).$$

As discussed in §1, these program-agnostic semantics determine exactly the single-input extensional properties of C (and nontermination in the case of $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{at}}$). The divergence-agnostic semantics are also noticeably less precise than the program-aware semantics of sets of programs, $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{aw}}$, which explicitly gives the semantics of each program $c \in C$, as illustrated in Example 3.9 below.

Example 3.9. Consider two sets of programs, $S_1 = \{x := 1, x := 2\}$, and $S_2 = \{\text{if } x == 0 \text{ then } x := 1 \text{ else } x := 2, \text{if } \neg(x == 0) \text{ then } x := 2 \text{ else } x := 1\}$. Then for an arbitrary input state σ, $\langle\!\langle S_1\rangle\!\rangle^{\mathbf{a}}(\sigma) = \langle\!\langle S_2\rangle\!\rangle^{\mathbf{a}}(\sigma) = \{\sigma[x \mapsto 1], \sigma[x \mapsto 2]\}$. More generally, it can be stated that $\langle\!\langle S_1\rangle\!\rangle^{\mathbf{a}} = \langle\!\langle S_2\rangle\!\rangle^{\mathbf{a}} = \lambda\Sigma. \bigcup_{\sigma \in \Sigma}\{\sigma[x \mapsto 1], \sigma[x \mapsto 2]\}$.

In contrast, $\langle\!\langle S_1\rangle\!\rangle^{\mathbf{aw}} \neq \langle\!\langle S_2\rangle\!\rangle^{\mathbf{aw}}$. Consider $[\![x := 1]\!] = \lambda\sigma.\sigma[x \mapsto 1]$. Then $[\![x := 1]\!] \in \langle\!\langle S_1\rangle\!\rangle^{\mathbf{aw}}$, but $[\![x := 1]\!] \notin \langle\!\langle S_2\rangle\!\rangle^{\mathbf{aw}}$: each function in S_2 assigns 2 to x for some σ. Thus $\langle\!\langle S_1\rangle\!\rangle^{\mathbf{aw}} \neq \langle\!\langle S_2\rangle\!\rangle^{\mathbf{aw}}$.

3.4 Granularity

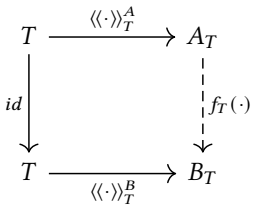
While the two program-agnostic and the program-aware semantics are equi-expressive for single programs, the situation changes for sets of programs. The following definition of *granularity* formalizes the notion of the relative expressivity of a semantics to capture this discrepancy.

Definition 3.10 (Granularity of a Semantics Over Sets of Programs). Let A and B be two signatures. We say that a semantics for sets of programs $\langle\!\langle\cdot\rangle\!\rangle^A$ is *finer* than another semantics $\langle\!\langle\cdot\rangle\!\rangle^B$ (or that $\langle\!\langle\cdot\rangle\!\rangle^B$ is *coarser* than $\langle\!\langle\cdot\rangle\!\rangle^A$), denoted by $\langle\!\langle\cdot\rangle\!\rangle^B \preceq \langle\!\langle\cdot\rangle\!\rangle^A$, iff for each type T (*Stmt*, *Exp*, or *BExp*) there exists a function f_T such that, for all $C \in \boldsymbol{T}$, $f_T(\langle\!\langle C\rangle\!\rangle^A_T) = \langle\!\langle C\rangle\!\rangle^B_T$.

When $\langle\!\langle\cdot\rangle\!\rangle^B \preceq \langle\!\langle\cdot\rangle\!\rangle^A$, we say that $\langle\!\langle\cdot\rangle\!\rangle^A$ is finer than $\langle\!\langle\cdot\rangle\!\rangle^B$. When $\langle\!\langle\cdot\rangle\!\rangle^B \preceq \langle\!\langle\cdot\rangle\!\rangle^A$ but $\langle\!\langle\cdot\rangle\!\rangle^A \not\preceq \langle\!\langle\cdot\rangle\!\rangle^B$, we say that $\langle\!\langle\cdot\rangle\!\rangle^A$ is strictly finer than $\langle\!\langle\cdot\rangle\!\rangle^B$, denoted by $\langle\!\langle\cdot\rangle\!\rangle^B \prec \langle\!\langle\cdot\rangle\!\rangle^A$. This granularity preorder induces a complete lattice, which we prove in the extended version of the paper [20].

Note that f_T is essentially $\langle\!\langle(\langle\!\langle\cdot\rangle\!\rangle^A_T)^{-1}\rangle\!\rangle^{\widehat{B}}_T$, using the convention that, when $\langle\!\langle\cdot\rangle\!\rangle^B_T : T \to \tau_T$, we say $\langle\!\langle\cdot\rangle\!\rangle^{\widehat{B}}_T : 2^T \to 2^{\tau_T}$.

[5] When we consider a semantics over sets of programs (e.g. $\langle\!\langle\cdot\rangle\!\rangle^A_T$), we assume that it carries with it $vars(\cdot)$. Thus, $\langle\!\langle\cdot\rangle\!\rangle^A_T$ really means $\langle\!\langle\cdot\rangle\!\rangle^A_T \times vars(\cdot)$. This assumption is a reasonable one (because *vars* can be computed immediately from an RTG), and the assumption will be needed in later proofs (§ 5).



Definition 3.10 can be understood via the diagram on the left: f_T is such that this diagram commutes. This demands that $(\langle\langle\cdot\rangle\rangle_T^A)^{-1}$ refines $(\langle\langle\cdot\rangle\rangle_T^B)^{-1}$; a semantics $\langle\langle\cdot\rangle\rangle_T^A$ is finer than $\langle\langle\cdot\rangle\rangle_T^B$ if the partition induced by $(\langle\langle\cdot\rangle\rangle_T^A)^{-1}$ is finer than the partition induced by $(\langle\langle\cdot\rangle\rangle_T^B)^{-1}$.

Intuitively, for semantics $\langle\langle\cdot\rangle\rangle^A$ and $\langle\langle\cdot\rangle\rangle^B$, $\langle\langle\cdot\rangle\rangle^B \preceq \langle\langle\cdot\rangle\rangle^A$ if for all C, $\langle\langle C\rangle\rangle^A$ lets us uniquely identify $\langle\langle C\rangle\rangle^B$. Recall that in this paper, a semantics $\langle\langle\cdot\rangle\rangle^A$ is assumed to implicitly carry $vars(\cdot)$, a function that returns the set of variables used by a set of programs. Thus, when considering $\langle\langle C\rangle\rangle^A$, the function f_T has access to $vars(C)$ through its argument as well.

From Definition 3.10, one can show that $\langle\langle\cdot\rangle\rangle^{\mathbf{a}} \prec \langle\langle\cdot\rangle\rangle^{\mathbf{at}}$ and $\langle\langle\cdot\rangle\rangle^{\mathbf{at}} \prec \langle\langle\cdot\rangle\rangle^{\mathbf{aw}}$.

Theorem 3.11 ($\langle\langle\cdot\rangle\rangle^{\mathbf{aw}}$ is strictly finer than $\langle\langle\cdot\rangle\rangle^{\mathbf{at}}$). *thmsinglessthanaware I.e.,* $\langle\langle\cdot\rangle\rangle^{\mathbf{at}} \prec \langle\langle\cdot\rangle\rangle^{\mathbf{aw}}$.

Proof. To see that $\langle\langle\cdot\rangle\rangle^{\mathbf{at}} \preceq \langle\langle\cdot\rangle\rangle^{\mathbf{aw}}$, simply take $f(\langle\langle C\rangle\rangle^{\mathbf{aw}}) = \bigcup_{[\![c]\!]^{at} \in \langle\langle C\rangle\rangle^{\mathbf{aw}}} [\![c]\!]^{at}$.

To see that $\langle\langle\cdot\rangle\rangle^{\mathbf{aw}} \not\preceq \langle\langle\cdot\rangle\rangle^{\mathbf{at}}$, recall the two sets of statements $S_1 = \{x_1 := 2,\ x_1 := 1\}$ and $S_2 = \{\text{if } x_1 == 0 \text{ then } x_1 := 1 \text{ else } x_1 := 2,\ \text{ if } \neg(x_1 == 0) \text{ then } x_1 := 2 \text{ else } x_1 := 1\}$ from Example 3.9. As illustrated in Example 3.9, $\langle\langle S_1\rangle\rangle^{\mathbf{at}} = \langle\langle S_2\rangle\rangle^{\mathbf{at}}$, but because $\langle\langle S_1\rangle\rangle^{\mathbf{aw}} \neq \langle\langle S_2\rangle\rangle^{\mathbf{aw}}$, there cannot exist f_{Stmt} where $f_{Stmt}(\langle\langle S\rangle\rangle^{\mathbf{at}}) = \langle\langle S\rangle\rangle^{\mathbf{aw}}$ for all $S \in \mathbf{Stmt}$. □

Theorem 3.12 ($\langle\langle\cdot\rangle\rangle^{\mathbf{at}}$ is strictly finer than $\langle\langle\cdot\rangle\rangle^{\mathbf{a}}$). *thmyellessthangrn I.e.,* $\langle\langle\cdot\rangle\rangle^{\mathbf{a}} \prec \langle\langle\cdot\rangle\rangle^{\mathbf{at}}$.

Theorem 3.12 can be proven by observing that $\langle\langle\cdot\rangle\rangle^{\mathbf{a}}$ cannot distinguish between an empty set of programs and a set C_{divg} that only contains diverging programs, because $\langle\langle\emptyset\rangle\rangle^{\mathbf{a}} = \langle\langle C_{divg}\rangle\rangle^{\mathbf{a}} = \emptyset$, but $\langle\langle\cdot\rangle\rangle^{\mathbf{at}}$ can. The full proof can be found in the extended version of the paper [20].

4 Compositionality of the Semantics of Sets of Programs

Having established a formal notion of semantics in §3.2, we now turn to the issue of *compositionality*. As argued in §1, the compositionality of a semantics is crucial for supporting practical reasoning methods. In this section, we formally define what it means for a semantics to be compositional, and then prove that $\langle\langle\cdot\rangle\rangle^{\mathbf{aw}}$ *is* compositional, whereas $\langle\langle\cdot\rangle\rangle^{\mathbf{a}}$ and $\langle\langle\cdot\rangle\rangle^{\mathbf{at}}$ *are not* compositional.[6]

The compositionality of a semantics is inspired by *grammar-flow analysis* (GFA) [25]. GFA is a formalism for recursively specifying, and computing, aggregate properties of the nonterminals in a given regular tree grammar. A compositional semantics for sets of programs does something very similar: it computes properties over a given inductive set of programs (which is, in this paper, defined by a regular tree grammar). Thus, the notion that "a semantics for a set of programs $\langle\langle\cdot\rangle\rangle^A$ is compositional" can be taken to mean that $\langle\langle\cdot\rangle\rangle^A$ should be computable in a fashion similar to grammar-flow analysis (i.e., the semantics of a nonterminal should be computed from the semantics of its productions, and the semantics of each production is computed from the semantics of its terms), an analogy that may be drawn for many other analyses over sets of programs as well (as illustrated in §2). We capture this notion of compositionality by requiring that the semantics $\langle\langle\cdot\rangle\rangle^A$ satisfy the following intuitive properties for any input set of programs C, defined by a regular tree grammar G_C:

- For each nonterminal $N \in G_C$, defined by the production right-hand sides (RHSs) $R_1, \cdots, R_n$, the denotation $\langle\langle N\rangle\rangle^A$ is obtained by applying some aggregation function $f_\cup^A$ over the denotations of each production RHS: i.e., $\langle\langle N\rangle\rangle^A = f_\cup^A(\langle\langle R_1\rangle\rangle^A, \cdots, \langle\langle R_2\rangle\rangle^A)$.
- For a constructor α and an RHS R of the form $\alpha(C_1, \cdots, C_n)$, the denotation $\langle\langle R\rangle\rangle^A$ is obtained by applying some function f_α^A to the denotations of the "operand" sets: i.e., $\langle\langle R\rangle\rangle^A = f_\alpha^A(\langle\langle C_1\rangle\rangle^A, \cdots, \langle\langle C_n\rangle\rangle^A)$.

[6] In §3, none of $\langle\langle\cdot\rangle\rangle^{\mathbf{a}}$, $\langle\langle\cdot\rangle\rangle^{\mathbf{at}}$, and $\langle\langle\cdot\rangle\rangle^{\mathbf{aw}}$ are stated in a compositional manner. Theorem 4.2 shows that $\langle\langle\cdot\rangle\rangle^{\mathbf{aw}}$ can be characterized compositionally.

For example, given a set of programs like if $x_1 == 2$ then $x_2 := E$ else S, a compositional semantics is able to compute the semantics of this set using only the semantics of E and S, without individually considering all programs in if $x_1 == 2$ then $x_2 := E$ else S. In contrast, a characterization like $\langle\!\langle \text{if } x_1 == 2 \text{ then } x_2 := E \text{ else } S \rangle\!\rangle^A = f(\{[\![\text{if } x_1 == 2 \text{ then } x_2 := e \text{ else } s]\!]^A \mid e \in E, s \in S\})$ *is noncompositional:* it enumerates all programs in the set. Definition 4.1 formalizes this intuition.

Definition 4.1 (Compositionality). Let $\mathcal{A}$ denote a finite set of constructors, and G be a regular tree grammar that only uses constructors in $\mathcal{A}$. Let N be a nonterminal in G, for which the productions are defined as follows ($N_{1,1}, \cdots, N_{k,n_k}$ are nonterminals of G):

$$N ::= \alpha_1^{(n_1)}(N_{1,1}, \ldots, N_{1,n_1}) \mid \ldots \mid \alpha_k^{(n_k)}(N_{k,1}, \ldots, N_{k,n_k})$$

We say that a semantics for sets of programs $\langle\!\langle \cdot \rangle\!\rangle^A$ *admits a compositional characterization over* $\mathcal{A}$, or simply that $\langle\!\langle \cdot \rangle\!\rangle^A$ is *compositional over* $\mathcal{A}$, if there exists a function $f_{\alpha_i}^A$ for each $\alpha_i \in \mathcal{A}$, and an aggregate function $f_\cup^A$, such that the following holds for any G and N:

$$\langle\!\langle L(N) \rangle\!\rangle^A = f_\cup^A(f_{\alpha_1}^A(\langle\!\langle L(N_{1,1}) \rangle\!\rangle^A, \ldots, \langle\!\langle L(N_{1,n_1}) \rangle\!\rangle^A), \ldots, f_{\alpha_k}^A(\langle\!\langle L(N_{k,1}) \rangle\!\rangle^A, \ldots, \langle\!\langle L(N_{k,n_k}) \rangle\!\rangle^A))$$

We simply say that $\langle\!\langle \cdot \rangle\!\rangle^A$ admits a compositional characterization, or is compositional, if it is compositional over the set of all operators in G_{imp}. We say that $\langle\!\langle \cdot \rangle\!\rangle^A$ is compositional over sets of loop-free programs if it is compositional over the set of all operators in G_{imp} except While.

In the rest of the section, we show that the program-aware semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{aw}}$ admits a compositional characterization, whereas the program-agnostic semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$ admit compositional characterizations only over sets of loop-free programs.

Theorem 4.2 (Compositionality of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{aw}}$). *The program-aware semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{aw}}$ admits a compositional characterization.*

Proof. Given $\alpha^{(n)}$ and $\langle\!\langle C_1 \rangle\!\rangle^{\mathbf{aw}}, \cdots, \langle\!\langle C_n \rangle\!\rangle^{\mathbf{aw}}$, we want to determine $\langle\!\langle \alpha^{(n)}(C_1, \cdots, C_n) \rangle\!\rangle^{\mathbf{aw}}$.

We observe that $\langle\!\langle \alpha^{(n)}(C_1, \cdots, C_n) \rangle\!\rangle^{\mathbf{aw}}$ is exactly $\{\alpha^{(n)}([\![c_1]\!]^{at}, \cdots, [\![c_n]\!]^{at}) \mid \forall j \leq n \,.\, [\![c_j]\!]^{at} \in \langle\!\langle C_j \rangle\!\rangle^{\mathbf{aw}}\}$, where applying $\alpha^{(n)}$ to $[\![c_j]\!]^{at}$ means to use the standard compositional definition of $[\![\cdot]\!]^{at}$ over $\alpha^{(n)}$, shown in Figure 4 for every constructor besides while.

For example, for the sequence constructor, $\langle\!\langle S_1; S_2 \rangle\!\rangle^{\mathbf{aw}} = \{[\![s_2]\!]^{at} \circ [\![s_1]\!]^{at} \mid [\![s_1]\!]^{at} \in \langle\!\langle S_1 \rangle\!\rangle^{\mathbf{aw}} \wedge [\![s_2]\!]^{at} \in \langle\!\langle S_2 \rangle\!\rangle^{\mathbf{aw}}\}$. For the while constructor, taking μ as the usual least fixed-point operator:

$$\langle\!\langle \text{while } B \text{ do } S \rangle\!\rangle^{\mathbf{aw}} = \left\{ \mu f. \begin{cases} f([\![s]\!]^{at}(\sigma)) & \text{if } [\![b]\!]^{at}(\sigma) \\ \sigma & \text{else} \end{cases} \;\middle|\; [\![b]\!]^{at} \in \langle\!\langle B \rangle\!\rangle^{\mathbf{aw}} \wedge [\![s]\!]^{at} \in \langle\!\langle S \rangle\!\rangle^{\mathbf{aw}} \right\}$$

More intuitively (and abusing notation), we can say $\langle\!\langle \text{while } B \text{ do } S \rangle\!\rangle^{\mathbf{aw}} = \{\text{while } [\![b]\!]^{at} \text{ do } [\![s]\!]^{at} \mid [\![b]\!]^{at} \in \langle\!\langle B \rangle\!\rangle^{\mathbf{aw}} \wedge [\![s]\!]^{at} \in \langle\!\langle S \rangle\!\rangle^{\mathbf{aw}}\}$, i.e., we consider all the functions that can be built as loops whose guard computes a function in $\langle\!\langle B \rangle\!\rangle^{\mathbf{aw}}$ and whose body computes a function in $\langle\!\langle S \rangle\!\rangle^{\mathbf{aw}}$. For unions, if $C = C_1, \cdots, C_m$, then $\langle\!\langle C \rangle\!\rangle^{\mathbf{aw}} = \bigcup_{j \leq m} \langle\!\langle C_j \rangle\!\rangle^{\mathbf{aw}}$.

Consequently, $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{aw}}$ is compositional. □

Next, we show that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$ are compositional for loop-free programs.

Theorem 4.3 (Loop-Free Compositionality of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$). *The program-agnostic semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$ admit compositional characterizations over loop-free programs.*

Proof. Figure 5 provides a compositional characterization of both semantics; equivalence to Definition 3.8 can be proven by structural induction. □

However, $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$ are no longer compositional when programs are allowed to have loops.

$$\langle\!\langle 0 \rangle\!\rangle^{\mathbf{a}}(\sigma) = \{\sigma[e_t \mapsto 0]\} \qquad \langle\!\langle 1 \rangle\!\rangle^{\mathbf{a}}(\sigma) = \{\sigma[e_t \mapsto 1]\} \qquad \langle\!\langle x_j \rangle\!\rangle^{\mathbf{a}}(\sigma) = \{\sigma[e_t \mapsto x_j]\}$$

$$\langle\!\langle \mathrm{t} \rangle\!\rangle^{\mathbf{a}}(\sigma) = \{\sigma[b_t \mapsto \mathrm{t}]\} \qquad \langle\!\langle \mathrm{f} \rangle\!\rangle^{\mathbf{a}}(\sigma) = \{\sigma[b_t \mapsto \mathrm{f}]\}$$

$$\langle\!\langle \neg B \rangle\!\rangle^{\mathbf{a}}(\sigma) = \{\sigma[b_t \mapsto \neg\sigma_b[b_t]] \mid \sigma_b \in \langle\!\langle B \rangle\!\rangle^{\mathbf{a}}(\sigma)\} \qquad \langle\!\langle \bigcup_{j<n} C_j \rangle\!\rangle^{\mathbf{a}}(\sigma) = \bigcup_{j<n} \langle\!\langle C_j \rangle\!\rangle^{\mathbf{a}}(\sigma)$$

$$\langle\!\langle A_1 \odot A_2 \rangle\!\rangle^{\mathbf{a}}(\sigma) = \{\sigma[r_t \mapsto \sigma_1[a_t] \odot \sigma_2[a_t]] \mid \sigma_1 \in \langle\!\langle A_1 \rangle\!\rangle^{\mathbf{a}}(\sigma), \sigma_2 \in \langle\!\langle A_2 \rangle\!\rangle^{\mathbf{a}}(\sigma)\}$$

$$\langle\!\langle x := E \rangle\!\rangle^{\mathbf{a}}(\sigma) = \{\sigma[x \mapsto \sigma_e[e_t]] \mid \sigma_e \in \langle\!\langle E \rangle\!\rangle^{\mathbf{a}}(\sigma)\} \qquad \langle\!\langle S_1; S_2 \rangle\!\rangle^{\mathbf{a}}(\sigma) = \langle\!\langle S_2 \rangle\!\rangle^{\mathbf{a}}(\langle\!\langle S_1 \rangle\!\rangle^{\mathbf{a}}(\sigma))$$

$$\langle\!\langle \text{if } B \text{ then } S_1 \text{ else } S_2 \rangle\!\rangle^{\mathbf{a}}(\sigma) = \begin{cases} \langle\!\langle S_1 \rangle\!\rangle^{\mathbf{a}}(\sigma) & \{\sigma_b[b_t] \mid \sigma_b \in \langle\!\langle B \rangle\!\rangle^{\mathbf{a}}(\sigma)\} = \{\mathrm{t}\} \\ \langle\!\langle S_2 \rangle\!\rangle^{\mathbf{a}}(\sigma) & \{\sigma_b[b_t] \mid \sigma_b \in \langle\!\langle B \rangle\!\rangle^{\mathbf{a}}(\sigma)\} = \{\mathrm{f}\} \\ \langle\!\langle S_1 \rangle\!\rangle^{\mathbf{a}}(\sigma) \cup \langle\!\langle S_2 \rangle\!\rangle^{\mathbf{a}}(\sigma) & \text{otherwise} \end{cases}$$

Fig. 5. Compositional definitions for the program-agnostic semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ (Definition 3.8). In the fourth line, $\odot$ denotes a binary operator (either $+$, $-$, $<$, $==$, or $\wedge$), and a_t and r_t are auxiliary variables of the appropriate type. Note that $\langle\!\langle C \rangle\!\rangle^{\mathbf{a}}(X) = \bigcup_{\sigma \in X} \langle\!\langle C \rangle\!\rangle^{\mathbf{a}}(\sigma)$.

Theorem 4.4 (Noncompositionality of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$). *Both $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$ do not admit compositional characterizations when loops are permitted.*

Proof. We prove that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ is not compositional for loops; the proof is identical for $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$.

We will show that there are two sets of loop guards, B_1 and B_2, with the *same* program-agnostic semantics but which can be used to construct otherwise identical sets of loops with *different* program-agnostic semantics. Let

$$B_1 = \{x == 1, \neg(x == 1)\} \qquad B_2 = \{x == 2, \neg(x == 2)\}.$$

Then observe for any input state $\sigma \in \mathrm{State}$ and $B = B_1$ or B_2, there is a guard $\mathrm{guard}_{\mathrm{true}} \in B$ such that $[\![\mathrm{guard}_{\mathrm{true}}]\!]^q(\sigma)[b_t] = \mathrm{t}$ and a guard $\mathrm{guard}_{\mathrm{false}} \in B$ such that $[\![\mathrm{guard}_{\mathrm{false}}]\!]^q(\sigma)[b_t] = \mathrm{f}$. Thus, on every set of input states Σ, $\langle\!\langle B_1 \rangle\!\rangle^{\mathbf{a}}(\Sigma) = \langle\!\langle B_2 \rangle\!\rangle^{\mathbf{a}}(\Sigma) = \Sigma[b_t \mapsto \mathrm{t}] \cup \Sigma[b_t \mapsto \mathrm{f}]$.

Now consider two sets of loops, W_1 and W_2, that share the same body:

$$W_1 = \text{while } B_1 \text{ do } x := x + 1 \qquad W_2 = \text{while } B_2 \text{ do } x := x + 1.$$

Because $\langle\!\langle B_1 \rangle\!\rangle^{\mathbf{a}} = \langle\!\langle B_2 \rangle\!\rangle^{\mathbf{a}}$, if $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ is compositional, it must be the case that $\langle\!\langle W_1 \rangle\!\rangle^{\mathbf{a}} = \langle\!\langle W_2 \rangle\!\rangle^{\mathbf{a}}$. However, a simple check proves they are not:

$$\begin{aligned} &\langle\!\langle W_1 \rangle\!\rangle^{\mathbf{a}}(\{\sigma \in \mathrm{State} \mid \sigma[x] = 0\}) \\ &\qquad = \{\sigma \in \mathrm{State} \mid \sigma[x] = 0\} \cup \{\sigma \in \mathrm{State} \mid \sigma[x] = 1\} \\ &\langle\!\langle W_2 \rangle\!\rangle^{\mathbf{a}}(\{\sigma \in \mathrm{State} \mid \sigma[x] = 0\}) \\ &\qquad = \{\sigma \in \mathrm{State} \mid \sigma[x] = 0\} \cup \{\sigma \in \mathrm{State} \mid \sigma[x] = 2\} \end{aligned}$$

Because $\langle\!\langle W_1 \rangle\!\rangle^{\mathbf{a}} \neq \langle\!\langle W_2 \rangle\!\rangle^{\mathbf{a}}$, we have that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$ are not compositional over sets of programs containing loops. □

Intuitively, Theorem 4.4 means that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ cannot always be determined compositionally when sets of loops are allowed. Thus, compositional verification techniques with semantics equivalent to $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ (see §§ 2.1 and 2.2) will be unable to prove many properties about such sets of programs. For example, no such technique can prove any property that distinguishes W_1 from W_2, as defined above. More generally, properties of sets of loops that involve multiple iterations of a loop are in general unprovable via compositional techniques over $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ (e.g., checking that some loop in W_2 can

map a σ in which $x \mapsto 0$ to $\sigma[x \mapsto 2]$ requires executing the loop while $\neg(x == 2)$ do $x := x + 1$ for two iterations). Put another way, $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ cannot "express multiple executions" as explained in §1, so verification techniques built on $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ cannot check many properties that require repeated execution of the same code.

5 Vector-State Semantics

Having proved that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$ do not admit compositional characterizations, the natural question to ask is: What is the coarsest *compositional* semantics that is at least as fine as $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ (or $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$)? In this section, we answer this question by proposing the *vector-state semantics* $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ (and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$), the coarsest compositional semantics that is at least as fine-grained as $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ (or as $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$).

5.1 Intuition for Vector-States

The fundamental challenge of describing the behavior of sets of programs is "synchronizing" programs with inputs to express multiple executions, as described in §1. To enable this synchronization, we lift our semantics from sets of individual input states to sets of sequences of input states. This approach allows us to apply the same program to each state in an input sequence, so that the corresponding output sequence is the result of executing a single program $c \in C$.

The noncompositionality of the program-agnostic semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$ can be attributed to their inability to express multiple executions. In particular, for a set of guards B and set of statements S, each loop (while b do s) $\in$ (while B do S) may execute b and s arbitrarily many times. Consequently, to answer questions about the program-agnostic semantics of while B do S, we need to be able to answer questions about multiple executions of specific pairs $(b, s) \in B \times S$. For example, to know that $p_n = (x \mapsto 10)$ is a possible final state of $\langle\!\langle$while B do $S\rangle\!\rangle^{\mathbf{a}}(\{(x \mapsto 0)\})$, we would need to know that there is a guard $b \in B$, a loop body $s \in S$, and a sequence of states $[p_1, \cdots, p_n]$ for which b is true on all p_j except p_n, and s maps each p_j to p_{j+1}.

More generally, given a set of programs C and sequences $[p_1, \cdots, p_n]$ and $[q_1, \cdots, q_n]$ of inputs and outputs, one must be able to ask whether there exists a *single* program $c \in C$ such that $[\![c]\!]^a(p_j) = q_j$ for all j. Questions of this sort cannot be answered from $\langle\!\langle C \rangle\!\rangle^{\mathbf{a}}$, yet they are necessary to understand the program-agnostic behavior of sets of loops defined in terms of C.

The goal of this section is to introduce two new, *vector-state semantics*, denoted by $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$, which capture exactly this kind of information about C (with and without the ability to capture nontermination, respectively). These vector-state semantics admit compositional characterizations over while, and they are at least as fine as $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$, respectively. In addition, the vector-state semantics are the *coarsest* such semantics, and thus they represent the minimum amount of information about sets of programs one must track to enable compositional reasoning.

5.2 $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$: Divergence-Agnostic Vector-State Semantics

We begin by developing $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$—a divergence-agnostic vector-state semantics. We first provide some intuition about how $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ should operate on input vector-states, and formalize the definition of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$. Then, we prove that (*i*) $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ has a compositional characterization, and (*ii*) $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ is the coarsest-grained compositional semantics that is finer than $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$. As discussed, we will use vector-states to capture the behavior of a set of programs (e.g., a set of loop bodies) under multiple executions.

Following this intuition, $\langle\!\langle C \rangle\!\rangle^{\mathbf{v}}$ should let each $c \in C$ operate over each entry in the vector to generate an output vector-state. For example, given the set $S = \{x := x + 2, x := x + 10\}$, the vector-state semantics $\langle\!\langle S \rangle\!\rangle^{\mathbf{v}}([(x \mapsto 2), (x \mapsto 4)])$ yields $\{[(x \mapsto 4), (x \mapsto 6)], [(x \mapsto 12), (x \mapsto 14)]\}$.

We define $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ so that, when a program diverges on *any* state in a vector-state, the *entire* output vector is discarded. This is similar to how Hoare logic returns false as the postcondition of a diverging loop. For example, for the singleton set $S = \{$while $x < 2$ do $x := x - 1\}$, the vector-state semantics

$\langle\!\langle S \rangle\!\rangle^{\mathbf{v}}([(x \mapsto 2), (x \mapsto 4)])$ yields $\{[(x \mapsto 2), (x \mapsto 4)]\}$ because the loop does not diverge on any input in the vector, but $\langle\!\langle S \rangle\!\rangle^{\mathbf{v}}([(x \mapsto 2), (x \mapsto 4), (x \mapsto 1)])$ yields $\emptyset$: the *entire output vector-state* is eliminated, due to the existence of a diverging element in the input vector-state.

5.2.1 Formalization of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$. In the following, State^* denotes the set of finite sequences over State.

Definition 5.1 (Divergence-Agnostic Vector-State Semantics of a Program). The *vector-state semantics* for programs $[\![\cdot]\!]^{\mathrm{v}}$ over the signature $\mathrm{v}(2^{\text{State}^*} \to 2^{\text{State}^*})$ is defined as follows. We first define the semantics for singleton vector states. For a program $c \in T$ and finite vector state $[v_1, \cdots, v_n] \in \text{State}^*$, the semantics $[\![c]\!]^{\mathrm{v}}(\{[v_1, \cdots, v_n]\}))$ is defined as follows:

(1) If for every v_i the program c does not diverge—i.e., $[\![c]\!]^{a}(v_i) \neq \emptyset$—then

$$[\![c]\!]^{\mathrm{v}}(\{[v_1, \ldots, v_n]\}) = \{[[\![c]\!]^{a}(v_1), \cdots, [\![c]\!]^{a}(v_n)]\}.$$

(2) If for some v_i the program c does diverge—i.e., $[\![c]\!]^{a}(v_i) = \emptyset$—then

$$[\![c]\!]^{\mathrm{v}}(\{v\}) = \emptyset.$$

Note that entries of the output vectors are technically singleton sets; we unpack them as states. Then for a set of vector states $V \subseteq \text{State}^*$, we define $[\![c]\!]^{\mathrm{v}}(V) = \bigcup_{u \in V} [\![c]\!]^{\mathrm{v}}(\{u\})$.

Intuitively, $[\![c]\!]^{\mathrm{v}}(\{v\})$ (often abbreviated as $[\![c]\!]^{\mathrm{v}}(v)$) gives the output of c on v for each entry of v unless c diverges on some state in v. If c diverges along v, then no vector-state is returned. Thus, $[\![c]\!]^{\mathrm{v}}(V)$ will show no trace of vectors $v \in V$ for which $[\![c]\!]^{\mathrm{v}}(v) = \emptyset$. Generalizing $[\![\cdot]\!]^{\mathrm{v}}$ to sets of programs is done by unioning the semantics of all programs in the set, similar to Definition 3.8.

Definition 5.2 (Divergence-Agnostic Vector-State Semantics of a Set of Programs). The *vector-state semantics* for sets of programs, denoted $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$, over the signature $\mathbf{v}(2^{\text{State}^*} \to 2^{\text{State}^*})$ is defined for an input set $V \subseteq \text{State}^*$ as follows:

$$\langle\!\langle C \rangle\!\rangle^{\mathbf{v}}(V) = \bigcup_{c \in C} [\![c]\!]^{\mathrm{v}}(V).$$

Example 5.3. Consider $S = \{x := x + n \mid n \in \mathbb{N}\}$. Given $v = [(x \mapsto 1), (x \mapsto 2), (x \mapsto 3)]$, we get $\langle\!\langle S \rangle\!\rangle^{\mathbf{v}}(v) = \{[(x \mapsto 1 + n), (x \mapsto 2 + n), (x \mapsto 3 + n)] \mid n \in \mathbb{N}\}$.

5.2.2 Properties of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$. $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ admits a compositional characterization (Theorem 5.4), and is the coarsest compositional semantics that is finer-grained than $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ (Theorem 5.5).

Theorem 5.4 (Compositionality of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$). *thmyelwkcomp $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ admits a compositional characterization.*

Figure 6 gives compositional characterizations of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ over unions and all constructors except while. The full proof is provided in the extended version of the paper [20]; we give a sketch below.

Proof Sketch. We sketch only the most complex cases—if-then-else and while. We restrict our explanation to singleton sets of inputs for simplicity of presentation.

For if B then S_1 else S_2, we can determine the ways that guards in B can split our inputs by evaluating $\langle\!\langle B \rangle\!\rangle^{\mathbf{v}}(v) = \{v^b \mid b \in B\}$. For each such v^b, we pass the part of v where v^b is true to $\langle\!\langle S_1 \rangle\!\rangle^{\mathbf{v}}$ and the false part to $\langle\!\langle S_2 \rangle\!\rangle^{\mathbf{v}}$. We then reassemble the results according to v^b.

For while B do S and vectors $[\sigma]$, we employ a "generate and test" strategy. The execution of a loop can be written as the sequence of states reached after every iteration. To find all possible loop executions, we consider each finite vector t of states beginning with σ and ask (*i*) whether any guard can send all t's states to true except the final state based on $\langle\!\langle B \rangle\!\rangle^{\mathbf{v}}$, and (*ii*) whether any body maps each state in t to its successor based on $\langle\!\langle S \rangle\!\rangle^{\mathbf{v}}$. If both checks pass, the vector t represents a

$$\langle\!\langle 1 \rangle\!\rangle^{\mathbf{v}}(V) = \mathit{reduce}(\bigcup_{v \in V} \{v[e_t \mapsto 1]\})$$

$$\langle\!\langle x_j \rangle\!\rangle^{\mathbf{v}}(V) = \mathit{reduce}(\bigcup_{v \in V} \{v[e_t \mapsto v[x_j]]\}) \quad \langle\!\langle \mathrm{t} \rangle\!\rangle^{\mathbf{v}}(V) = \mathit{reduce}(\bigcup_{v \in V} \{v[b_t \mapsto \mathrm{t}]\})$$

$$\langle\!\langle \mathrm{f} \rangle\!\rangle^{\mathbf{v}}(V) = \mathit{reduce}(\bigcup_{v \in V} \{v[b_t \mapsto \mathrm{f}]\}) \quad \langle\!\langle \neg B \rangle\!\rangle^{\mathbf{v}}(V) = \{v^b[b_t \mapsto \neg v^b[b_t]] \mid v^b \in \langle\!\langle B \rangle\!\rangle^{\mathbf{v}}(V)\}$$

$$\langle\!\langle A_1 \odot A_2 \rangle\!\rangle^{\mathbf{v}}(V) = \{v^2[r_t \mapsto v^1[q_t] \odot v^2[q_t]] \mid v^1 \in \langle\!\langle A_1 \rangle\!\rangle^{\mathbf{v}}(V), v^2 \in \langle\!\langle A_2 \rangle\!\rangle^{\mathbf{v}}(V)\}$$

$$\langle\!\langle x := E \rangle\!\rangle^{\mathbf{v}}(V) = \mathit{reduce}(\bigcup_{v \in V} \{v^e[x \mapsto v^e[e_t]] \mid v^e \in \langle\!\langle E \rangle\!\rangle^{\mathbf{v}}(v)\})$$

$$\langle\!\langle S_1; S_2 \rangle\!\rangle^{\mathbf{v}}(V) = \langle\!\langle S_2 \rangle\!\rangle^{\mathbf{v}}(\langle\!\langle S_1 \rangle\!\rangle^{\mathbf{v}}(V))$$

$$\langle\!\langle \text{if } B \text{ then } S_1 \text{ else } S_2 \rangle\!\rangle^{\mathbf{v}}(V) =$$
$$\left\{ \mathit{interleave}(v^{s_1}, v^{s_2}, v^b) \,\middle|\, v^b \in \langle\!\langle B \rangle\!\rangle^{\mathbf{v}}(V), v^{s_1} \in \langle\!\langle S_1 \rangle\!\rangle^{\mathbf{v}}(\mathit{filter}(V, v^b)), v^{s_2} \in \langle\!\langle S_2 \rangle\!\rangle^{\mathbf{v}}(\mathit{filter}(V, \neg v^b)) \right\}$$

$$\langle\!\langle \text{while } B \text{ do } S \rangle\!\rangle^{\mathbf{v}}(V) = f^{\mathbf{v}}_{\mathit{while}}(\langle\!\langle B \rangle\!\rangle^{\mathbf{v}}, \langle\!\langle S \rangle\!\rangle^{\mathbf{v}}, V)$$

$$\langle\!\langle \bigcup_{j<n} S_j \rangle\!\rangle^{\mathbf{v}}(V) = \mathit{reduce}(\bigcup_{j<n} \langle\!\langle S_j \rangle\!\rangle^{\mathbf{v}}(V))$$

Fig. 6. A compositional characterization for the divergence-agnostic vector-state semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ defined in Definition 5.2. The only differences between the characterizations of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$ (the divergence-aware vector-state semantics introduced in §5.3) are that (*i*) for $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$, one does not apply *reduce* and (*ii*) the functions f_{while} are different. The symbol $\odot$ denotes a binary operator in G_{imp}, and q_t and r_t refer to auxiliary variables of the appropriate type. Substitutions $v[b \mapsto a]$ to vectors are applied elementwise. The functions f_{while}, *interleave*, and *filter* are defined in the proofs of Theorems 5.4 and 5.8, available in the extended version of the paper [20]. Briefly, $\mathit{interleave}(v^{s_1}, v^{s_2}, v^b)$ produces an array using entries of v^{s_1} where v^b is true and v^{s_2} where v^b is false, and $\mathit{filter}(V, v^b)$ deletes entries from vectors in V where v^b is false. On singleton vectors, the function f_{while} produces $\langle\!\langle \text{while } B \text{ do } S \rangle\!\rangle^{\mathbf{v}}([\sigma])$ by finding all realizable traces of loop executions on σ according to $\langle\!\langle B \rangle\!\rangle^{\mathbf{v}}$ and $\langle\!\langle S \rangle\!\rangle^{\mathbf{v}}$.

possible loop execution and its final state is a possible output of while B do S on σ. Generalizing to input vectors of length k requires considering concatenations of k guesses as a single vector.

□

The divergence-agnostic vector-state semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ is the *coarsest* compositional semantics that captures the program-agnostic semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$. In other words, $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ sets a lower bound on the granularity required for a compositional semantics over sets of programs that captures $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$ – i.e., a lower bound on the complexity, of reasoning about sets of programs in a compositional fashion.

Theorem 5.5 ($\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ is Coarsest Compositional for $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$). *thmyelwkmin For every compositional semantics $\langle\!\langle \cdot \rangle\!\rangle^{A}$ such that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}} \preceq \langle\!\langle \cdot \rangle\!\rangle^{A}$, we have that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}} \preceq \langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}} \preceq \langle\!\langle \cdot \rangle\!\rangle^{A}$.*

Proof Sketch. Clearly, $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}} \preceq \langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ as $\langle\!\langle C \rangle\!\rangle^{\mathbf{v}}$ on vectors of length 1 simulates $\langle\!\langle C \rangle\!\rangle^{\mathbf{a}}$.

We must then show that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ is the coarsest compositional semantics at least as fine as $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}}$. To do so, we observe that any compositional semantics $\langle\!\langle \cdot \rangle\!\rangle^{A}$ such that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{a}} \preceq \langle\!\langle \cdot \rangle\!\rangle^{A}$ must determine sets of statements S well enough to reason about the program-agnostic semantics of sets of loops in which S appears in the body. Given a set of statements S, for each pair (v, u) of input and output vector-states, we construct sets of loops $W_{v,u}$ that loop over the indices of our vectors and check the behavior of S on each entry (i.e., while b do $s_1; S; s_2$, where s_1 sets the input state to $v[i]$, s_2 checks if the output state is $u[i]$ and breaks if not, and b iterates through v). The proof concludes by showing the program-agnostic semantics of such sets of loops determines whether $u \in \langle\!\langle S \rangle\!\rangle^{\mathbf{v}}(v)$. Thus, $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}} \preceq \langle\!\langle \cdot \rangle\!\rangle^{A}$ on sets of statements. A similar argument carries over to expressions. □

The full proof of Theorem 5.5 is given in the extended version of the paper [20].

Multiple-Input Extensional Properties. The vector-state semantics $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{v}}$ extends the ability of $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{a}}$ to cover k-input extensional properties for unbounded k, which is useful for, e.g., capturing programming-by-example (PBE) program-synthesis problems [14, 32]. Specifications in PBE are given as a set of input-output examples $\{(in_1, out_1), \cdots, (in_k, out_k)\}$; whether or not such a PBE specification has a solution in a set of programs C can be checked by checking whether $[out_1, \cdots, out_k] \in \langle\!\langle C\rangle\!\rangle^{\mathbf{v}}([in_1, \cdots, in_n])$.

5.3 $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{vt}}$: Divergence-Aware Vector-State Semantics

We now introduce $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{vt}}$, the divergence-aware vector-state semantics, which is the coarsest compositional semantics at least as fine as the divergence-aware program-agnostic semantics $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{at}}$. Whereas $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{v}}$ was a straightforward vectorization of $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{a}}$ (Definition 5.1), nontermination requires more careful treatment: naïvely extending $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{at}}$ to vector-states will result in a semantics that is more fine-grained than necessary. In this section, we provide intuition about why nontermination imposes significant challenges on the construction of $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{vt}}$. The extended version of the paper [20] contains a formal treatment of $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{vt}}$.

5.3.1 The Naïve Approach: $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{vt}_{\text{fine}}}$. The natural approach to handle nontermination is to extend $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{v}}$ to operate over infinite vectors and record divergence on each state in a vector as follows:

Example 5.6 ($\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{vt}_{\text{fine}}}$ Semantics: Too Fine). Consider a semantics for single programs $[\![\cdot]\!]^{\mathbf{vt}_{\text{fine}}}$ that extends $[\![\cdot]\!]^{at}$ towards vector-states, such that $[\![c]\!]^{\mathbf{vt}_{\text{fine}}}(\{[v_1, \cdots, v_n]\}) = \{[[\![c]\!]^{at}(v_1), \cdots, [\![c]\!]^{at}(v_n)]\}$, for a program c and an input vector-state $[v_1, \cdots, v_n]$. Define $\langle\!\langle C\rangle\!\rangle^{\mathbf{vt}_{\text{fine}}}(V) = \bigcup_{c\in C}[\![c]\!]^{\mathbf{vt}_{\text{fine}}}(V)$ for a set of programs C and a set of input vector-states V. Then for a set of programs $S = \{w_1, w_2\}$ where $w_1 = \textsf{while } x = 1 \textsf{ do } x := x$ and $w_2 = \textsf{while } x = 2 \textsf{ do } x := x$,

$$\begin{aligned}
\langle\!\langle S\rangle\!\rangle^{\mathbf{vt}_{\text{fine}}}(\{[(x \mapsto 1), (x \mapsto 2)]\}) &= \{[[\![w_1]\!](x \mapsto 1), [\![w_1]\!](x \mapsto 2)], [[\![w_2]\!](x \mapsto 1), [\![w_2]\!](x \mapsto 2)]\} \\
&= \{[\uparrow, (x \mapsto 2)], [(x \mapsto 1), \uparrow]\} \\
\langle\!\langle S\rangle\!\rangle^{\mathbf{vt}_{\text{fine}}}(\{[(x \mapsto 2), (x \mapsto 3), \cdots]\}) &= \{[[\![w_1]\!](x \mapsto 2), [\![w_1]\!](x \mapsto 3)], [[\![w_2]\!](x \mapsto 2), [\![w_2]\!](x \mapsto 3)]\} \\
&= \{[(x \mapsto 2), (x \mapsto 3), \cdots], [\uparrow, (x \mapsto 3), \cdots]\}
\end{aligned}$$

Observe how $\uparrow$ may appear both before, after, and in between other entries.

Although $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{vt}_{\text{fine}}}$ is compositional and finer than $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{at}}$, it is not the *coarsest* such semantics. An alternative, coarser approach might be $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{vt}_{\text{coarse}}}$, where an input vector containing at least one diverging entry is mapped entirely to $\uparrow$: e.g., $\langle\!\langle\textsf{while } x < 1 \textsf{ do } x := x - 1\rangle\!\rangle^{\mathbf{vt}_{\text{coarse}}}([(x \mapsto 0), (x \mapsto 1)]) = \uparrow$ instead of $[\uparrow, (x \mapsto 1)]$. However, $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{vt}_{\text{coarse}}}$ turns out to be too *coarse*: $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{vt}_{\text{coarse}}}$ is non-compositional, for reasons similar to Theorem 4.4 (see the extended version of the paper [20] for details). The coarsest compositional divergence-aware semantics for vector-states lies in between these two approaches.

5.3.2 The Right Approach: $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{vt}}$. To define a semantics with the appropriate granularity, we adopt the idea of Example 5.6, but simplify the output of $\langle\!\langle S\rangle\!\rangle^{\mathbf{vt}_{\text{fine}}}(P) = X$ to remove unnecessary information about S. Given the output vector-state set X, our approach boils down to first (*i*) truncating all vectors in X at the first occurrence of $\uparrow$, denoted by $truncate(X)$, and second (*ii*) using the following two rules to reduce X (denoted by $reduce(X)$):

(1) If $[a_1, \cdots, a_n, \uparrow] \in X$ and $[a_1, \cdots, a_n, a_{n+1}, \cdots, a_m, \uparrow] \in X$, then $[a_1, \cdots, a_n, a_{n+1}, \cdots, a_m, \uparrow]$ does not appear in the reduction of X.
(2) If $[a_1, \cdots, a_n, \uparrow] \in X$ and an infinite vector $[a_1, \cdots, a_n, a_{n+1}, \cdots] \in X$, then $[a_1, \cdots, a_n, a_{n+1}, \cdots]$ does not appear in the reduction of X.

The coarsest compositional divergent-aware semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$ can then be defined, for a set of programs C and a set of input vector-states V, $\langle\!\langle C \rangle\!\rangle^{\mathbf{vt}}(V) = \mathit{reduce}(\mathit{truncate}(\langle\!\langle C \rangle\!\rangle^{\mathbf{vt}_{\text{fine}}}(V)))$.

Example 5.7 (Correct $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$ Semantics). We illustrate the behavior of $\mathit{truncate}(X)$, $\mathit{reduce}(X)$, and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$, by illustrating their application to the set of programs and inputs from Example 5.6.

$$\begin{aligned}
\langle\!\langle S \rangle\!\rangle^{\mathbf{vt}}(\{[(x \mapsto 1), (x \mapsto 2)]\}) &= \mathit{reduce}(\mathit{truncate}(\langle\!\langle S \rangle\!\rangle^{\mathbf{vt}_{\text{fine}}}(\{[(x \mapsto 1), (x \mapsto 2)]\})))\\
&= \mathit{reduce}(\mathit{truncate}(\{[\uparrow, (x \mapsto 2)], [(x \mapsto 1), \uparrow]\})\\
&= \mathit{reduce}(\{[\uparrow], [(x \mapsto 1), \uparrow]\})\\
&= \{[\uparrow]\}\\
\langle\!\langle S \rangle\!\rangle^{\mathbf{vt}}(\{[(x \mapsto 2), (x \mapsto 3), \cdots]\}) &= \mathit{reduce}(\mathit{truncate}(\langle\!\langle S \rangle\!\rangle^{\mathbf{vt}_{\text{fine}}}(\{[(x \mapsto 2), (x \mapsto 3), \cdots]\})))\\
&= \mathit{reduce}(\mathit{truncate}(\{[(x \mapsto 2), (x \mapsto 3), \cdots], [\uparrow, (x \mapsto 3), \cdots]\}))\\
&= \mathit{reduce}(\{[(x \mapsto 2), (x \mapsto 3), \cdots], [\uparrow]\}))\\
&= \{[\uparrow]\}
\end{aligned}$$

Truncation and reduction correspond to removing all information about S that could not be determined from the divergence-aware program-agnostic semantics of sets of loops W defined in terms of S. This idea is clarified in the extended version of the paper [20]. Removing this unnecessary information makes $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$ the coarsest-grained compositional semantics that is finer than $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$.

Theorem 5.8 (Compositionality of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$). *thmgrnwkcomp The divergence-aware vector-state semantics over sets of programs $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$ admits a compositional characterization.*

The proof of Theorem 5.8 very closely mirrors the proof of Theorem 5.4, with the slight differences that (*i*) as a shortcut, we can determine the semantics of guards in B directly by checking $\langle\!\langle B \rangle\!\rangle^{\mathbf{vt}}(x)$ where x is an enumeration of State $\restriction_{\mathit{vars}(B)}$, and (*ii*) we must handle infinite (and diverging) traces in addition to finite, nondiverging traces.

Theorem 5.9 ($\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$ is Coarsest Compositional for $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}}$). *thmgrnwkmin For every compositional semantics $\langle\!\langle \cdot \rangle\!\rangle^{A}$ such that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}} \preceq \langle\!\langle \cdot \rangle\!\rangle^{A}$, we have that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{at}} \preceq \langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}} \preceq \langle\!\langle \cdot \rangle\!\rangle^{A}$.*

The proof of Theorem 5.9 mimics the proof of Theorem 5.5 with the extra challenge of dealing with divergence. We refer the reader to the extended version of the paper [20] for the full proofs of Theorems 5.8 and 5.9.

Furthermore, we observe that $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$ is strictly finer than $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$. This relation is depicted in the lattice in Figure 1, and proven in the extended version of the paper [20].

6 Compositional Hoare-style Rules

We have argued that when a semantics is compositional, it is easier to design formal systems for reasoning about sets of programs. To validate this claim, we return to the idea of designing a Hoare-style proof system for inductively defined sets of programs. First, we show that in the same way there does not exist a compositional characterization for program-agnostic semantics of sets of programs, there is no sound and relatively complete compositional Hoare-style While-rule for automatically proving properties of sets of programs with program-agnostic triples (Theorem 6.3). Next, we show that for the two vector-state semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ and $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$, which both admit compositional characterizations, there *do* exist sound and relatively complete compositional Hoare-style While-rules (§6.1). Combined with previous work on Unrealizability Logic [18, 27]—a relatively complete, compositional Hoare-style proof system for sets of loop-free programs—these rules provide the first sound, relatively complete, compositional Hoare-style logics for reasoning about sets of programs.

We start by defining what it means for a while-loop rule to be compositional for a Hoare-style proof system for sets of programs under a specific semantics. Given a semantics $\langle\!\langle \cdot \rangle\!\rangle^{A}$ where the

signature $A(\tau)$ has type $\tau = X \to X$ for some X, an *unrealizability triple* $\{\!|P|\!\}^A\ S\ \{\!|Q|\!\}^A$ denotes that $\langle\!\langle S \rangle\!\rangle^A(P) \subseteq Q$ holds. When writing proofs by induction (i.e., for inductively defined sets of programs), one uses a set Γ to hold their inductive hypotheses. When Γ is a set of such triples, we use $\Gamma \vdash \{\!|P|\!\}^A\ S\ \{\!|Q|\!\}^A$ to mean that $\{\!|P|\!\}^A\ S\ \{\!|Q|\!\}^A$ holds, assuming that all triples in Γ hold.

Definition 6.1 (Compositional While-Rule). Given a semantics $\langle\!\langle \cdot \rangle\!\rangle^A$, a While-rule is compositional if it has the following form, where $\varphi(v_1, \cdots, v_n)$ is a formula over $v_1, \cdots, v_n$:

$$\frac{\Gamma \vdash \{\!|P_B|\!\}^A\ B\ \{\!|Q_B|\!\}^A \qquad \Gamma \vdash \{\!|P_S|\!\}^A\ S\ \{\!|Q_S|\!\}^A}{\Gamma \vdash \{\!|P|\!\}^A\ (\mathsf{while}\ B\ \mathsf{do}\ S)\ \{\!|\varphi(\mathit{vars}(B), \mathit{vars}(S), P, P_B, P_S, Q_B, Q_S)|\!\}^A}$$

Definition 6.1 disallows one from referring directly to B and S in the postcondition; only information that can be captured by a compositional semantics (e.g., the input and output sets P_B and Q_B for B) can be referenced. This restriction captures compositionality for inference rules.

Hoare-style logics often also contain "structural" rules that do not correspond to a single constructor (e.g., the weakening rule in standard Hoare logic [15]). An important structural rule we will consider in this paper is the GrmDisj rule from Unrealizability Logic [18, 27], which allows one to partition a set of programs into a finite number of subsets.

Definition 6.2 (The Grammar-Disjunction Rule [18]). Given a semantics $\langle\!\langle \cdot \rangle\!\rangle^A$, the grammar-disjunction rule GrmDisj is defined as follows:

$$\frac{\Gamma \vdash \{\!|P|\!\}^{\mathbf{a}}\ C_1\ \{\!|Q|\!\}^{\mathbf{a}} \quad \Gamma \vdash \{\!|P|\!\}^{\mathbf{a}}\ C_2\ \{\!|Q|\!\}^{\mathbf{a}}}{\Gamma \vdash \{\!|P|\!\}^{\mathbf{a}}\ (C_1 \cup C_2)\ \{\!|Q|\!\}^{\mathbf{a}}}\ \mathsf{GrmDisj}$$

GrmDisj allows a proof system more power than a strictly compositional characterization would: a proof system with GrmDisj allows one to partition sets of loops into smaller sets, while a proof system without GrmDisj is denied this ability. When proving or disproving the existence of a compositional While-rule, we wish to account for this additional power. Thus, we show that there exists no While-rule even if GrmDisj (and the standard weakening rule) are allowed—which in turn shows that Unrealizability Logic over a program-agnostic semantics is fundamentally incomplete.

Theorem 6.3 (No Program-Agnostic While-Rule). *thmnocompoprogagwhile There is no sound and relatively complete compositional While-rule over triples $\{\!|P|\!\}^{\mathbf{a}}\ C\ \{\!|Q|\!\}^{\mathbf{a}}$, even when finitely many applications of* GrmDisj *(and the standard weakening rule) are allowed.*

The proof (given in the extended version of the paper [20]) is similar to the proof of Theorem 4.4, except that it considers an infinite family of sets of guards to account for the GrmDisj rule.

6.1 Hoare While-Rules for Vector States

In contrast with program-agnostic semantics, the compositionality of vector-state semantics makes building compositional While-rules easy—we just encode the compositional characterizations.

We begin with a While-rule for the divergence-agnostic vector-state semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$.

Theorem 6.4 (Divergence-Agnostic Vector-State While-Rule). *thmvsyelwhilerule There is a sound and relatively complete compositional While-rule for Unrealizability Logic for the semantics $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$.*

Proof Sketch. We can adapt $f^{\mathbf{v}}_{\mathit{while}}$—the function that determines $\langle\!\langle \mathsf{while}\ B\ \mathsf{do}\ S \rangle\!\rangle^{\mathbf{v}}$ from $\langle\!\langle B \rangle\!\rangle^{\mathbf{v}}$ and $\langle\!\langle S \rangle\!\rangle^{\mathbf{v}}$ in the proof of Theorem 5.4—to construct an inference rule of the following form:

$$\frac{\Gamma \vdash \{\!|x = z|\!\}^{\mathbf{v}}\ B\ \{\!|Q_B|\!\}^{\mathbf{v}} \quad \Gamma \vdash \{\!|x = z|\!\}^{\mathbf{v}}\ S\ \{\!|Q_S|\!\}^{\mathbf{v}}}{\Gamma \vdash \{\!|P|\!\}^{\mathbf{v}}\ (\mathsf{while}\ B\ \mathsf{do}\ S)\ \{\!|Q|\!\}^{\mathbf{v}}}\ \mathsf{While}_{\mathsf{divergence\text{-}agnostic}}$$

The full proof is available in the extended version of the paper [20]. □

In this way, compositionality is a useful property in defining sound and complete compositional systems of reasoning. A similar rule works for $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$, as stated in Theorem 6.5.

Theorem 6.5 (Divergence-Aware Vector-State While-Rule). *thmvsgrnwhilerule There is a sound and relatively complete compositional While-rule for Unrealizability Logic for the semantics* $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$.

Proof Sketch. The full rule is given in the extended version of the paper [20], which merely applies $f^{\mathbf{vt}}_{while}$ from the proof of Theorem 5.8. The rule differs from the rule $\mathsf{While}_{\text{divergence-agnostic}}$ in that it introduces a variable $i \in \mathbb{N}$ to simulate making countably many queries to $\langle\!\langle S \rangle\!\rangle^{\mathbf{vt}}$:

$$\frac{\Gamma \vdash \{\!|P_B|\!\}^{\mathbf{v}}\ B\ \{\!|Q_B(x)|\!\}^{\mathbf{v}} \quad \Gamma \vdash \{\!|P_S(i)|\!\}^{\mathbf{v}}\ S\ \{\!|Q_S(i)|\!\}^{\mathbf{v}}}{\Gamma \vdash \{\!|P|\!\}^{\mathbf{v}}\ \mathsf{while}\ B\ \mathsf{do}\ S\ \{\!|Q|\!\}^{\mathbf{v}}}\ \mathsf{While}_{\text{divergence-aware}}$$

□

Both rules can be augmented with rules like those in [18] or [27] to yield the first sound and relatively complete proof systems for inductively defined sets of programs in G_{imp}. We observe that, although complex predicates are required in the While rules to achieve completeness, in practice one can often find a proof using simpler predicates as invariants. A similar phenomenon can be found in standard Hoare logic, which requires complex invariants to achieve completeness in theory, but where in practice one can often find a proof using simpler invariants.

7 Related Work

Program Synthesis and Unrealizability. The inputs to the semantics discussed in this paper align well with program-synthesis problems, which often assume a syntactic search space constrained by a regular tree grammar or some other recursively defined language ([1, 11, 19, 34], to single out just a few). As discussed in §5.2, the results on single-input properties discussed in this paper generalize almost immediately to multiple-input properties, which in turn capture PBE specifications used in many synthesizers [14, 19, 30, 32].

In particular, our work provides a formal result about the difficulty of proving *unrealizability* of synthesis problems (i.e., the non-existence of a solution). Two recent publications [18, 27] introduced *Unrealizability Logics*, Hoare-style logics for reasoning about inductively defined sets of programs. Both logics are sound, but not relatively complete when loops are allowed (see Footnote 2). The main semantics for sets of programs used in Nagy et al. [27] is coarser than $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$; Theorem 5.5 proves that a compositional rule for while loops under such semantics does not exist. In Kim et al. [18] (and in the appendices of Nagy et al. [27]), a vector-state semantics finer than $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$ is used. That semantics is compositional, but the inference rules are written with the assumption that a program-agnostic rule like the one disallowed by Theorem 6.3 exists. Vector-states are only added to discuss k-input extensional properties; their role in enabling compositionality is not exploited.

Nagy et al. [27] designed a proof synthesizer for an incomplete Unrealizability Logic for infinite vector-states. Their implementation suggests that reasoning over infinite vector-states is tractable, at least in simple cases, giving hope for tractable verification techniques to be derived from $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{v}}$.

Outcome Logic and Hyper Hoare Logic. An Outcome Logic is a program logic for uninterpreted nondeterministic programs expressed in a Kleene algebra. In an Outcome Logic, program semantics are monadic, and the monad is equipped with the operations of a partial commutative monoid to split monadic objects. Zilberstein et al. [35] give an instantiation of Outcome Logic for the set monad equipped with a set-union monoid operation. Hyper Hoare Logic is a program logic for imperative

nondeterministic programs in which pre- and postconditions are second-order predicates over sets of states, rather than first-order predicates over states [9].

The logic from §6 is not subsumed by existing logics, although Outcome Logic and Hyper Hoare Logic share some similarities. Because both logics reason about sets of states explicitly, and a vector-state is simply an indexed set of states, triples about a single program that are expressible in our divergence-agnostic vector-state logic are expressible in both Outcome and Hyper Hoare Logic. Triples about a single program in divergence-aware logic cannot be expressed in Hyper Hoare Logic because there is no treatment of divergence there; they may be expressible in an Outcome Logic if one can define an *EVS* monad, where *EVS* is the set of vector-states over which $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{vt}}$ operates. Note that neither logic expresses properties of sets of programs.

Because Outcome Logic is expressed over an uninterpreted Kleene algebra, one might wonder whether we can simply interpret elements of the Kleene algebra as sets of programs to obtain a complete Outcome Logic for sets of programs (i.e., can we represent a set of programs as a single nondeterministic program and use an Outcome Logic to reason about its behavior on vector-states?). Unfortunately, this reduction is not possible in general: the traces of infinite sets of while loops, such as the one in our proof of Theorem 6.3, do not always form regular languages and thus are not expressible by Kleene algebras (e.g., by the Myhill-Nerode Theorem [28]).

Incompleteness of Hoare-Style logics. Clarke [8] studies programming languages for which one cannot obtain a relatively complete "Hoare-style logic", which Clarke (and others, most notably Lipton [22]) interprets as a *syntax-directed proof system* involving axiomatic rules that capture the semantics of a program [2, 7]. Clarke proves that such logics cannot exist for certain combinations of language features (e.g., coroutines with parameterless recursive procedures).

Such "Hoare-style logics" are not necessarily compositional as defined in this paper. In particular, such "Hoare-style logics" may contain as part of their proof system a separate procedure that reduces deriving a triple into some other kind of decidable problem (e.g., enumeration in the case of finite interpretations), instead of constructing a proof tree. Clarke observes that these "Hoare-style logics" do not necessarily lead to practical proof systems [7, p11].

Our work defines compositionality as an additional desideratum to support practical reasoning; one might view compositionality as a formal criterion of "natural Hoare-style logics" mentioned by Clarke [7]. We are unaware of any non-compositional logics that see major use.

Variational Analysis. Variational analysis (VA) encompasses a broad set of techniques for tracking the behavior of finite sets of programs, where each program is the result of some combination of 'tags.' A combination of tags encodes the usage of different features, permissions, etc. For example, in a set of differently configured email clients, the tags (`encryption` = `RSA`, `spam` = `t`) would denote the unique configuration that uses RSA encryption and a spam filter. The main aim of VAs is to construct a map from tag combinations (i.e., specific programs) to program properties or behaviors.

For example, the semantics of the choice calculus, used in the analysis of variable software, builds a map from (*i*) combinations of finitely many tags describing choices about program syntax to (*ii*) program behavior [10]. Model checking-based VA approaches, useful in product-line analysis, build more compact maps from (*i*) propositional formulas that represent finite sets of configurations to (*ii*) program behaviors [13]. Faceted execution, used in dynamic enforcement of privacy policies, builds maps from (*i*) users to (*ii*) program states to model different users executing the same code [3].

We emphasize that VA is a family of specific techniques including the above, and does not provide a framework for semantic analysis or derivation of our key results (e.g., Theorem 5.5). One might wonder whether a VA technique (extended to operate over infinite sets) could, at least in theory, efficiently verify extensional properties for sets of programs. Our theory suggests that the answer is no, for two reasons. First, a complete, compositional VA technique capable of verifying extensional

properties will induce a semantics strictly finer than $\langle\!\langle \cdot \rangle\!\rangle^{\mathsf{V}}$ due to the need to track tag combinations, which correspond to syntactic information.[7] Second, expressing maps from tag combinations to semantic behaviors requires non-Cartesian predicates that explicitly relate program syntax and semantics; these predicates are complex and best avoided for infinite sets of programs (see §2.2).

Finally, we remark that Bittner et al. [4] define a notion of denotational semantics for variability languages to compare the expressive power of different VA formalisms. Both our work and theirs use denotational semantics to describe the expressivity of verification objectives, albeit for very different purposes. We believe that this strategy has potential to advance the analysis and design of verification techniques, as both works evidence.

Compilers and Self-Modifying Code. A compiler is the canonical example of a tool that creates a set of programs. Generally, "compiler correctness' is concerned with whether each (single) source program is translated correctly to an appropriate target program. The scheme for establishing compiler correctness is due to Burstall [5] and followed in most of the 14 passes in the CompCert compiler, which translated CLight to PowerPC assembly language [21].

A compiler-correctness problem closer to the questions addressed in this paper would be, e.g., to prove that a compiler can only emit target programs that satisfy a memory-safety property (i.e., the *set* of programs emittable by the compiler's translation function are memory-safe). Myreen [26] addressed a version of this problem: he developed a machine-code Hoare logic to support reasoning about self-modifying x86 machine code, which was applied to the verification of JIT compilers.

Other work on self-modifying code includes Gerth [12] and Cai et al. [6]. The latter is based on an extension of Hoare logic, but to support self-modification, it treats program instructions as any other mutable data structure—an issue that we do not face in our work. Gerth's earlier work also adopts a data-centric view of the world: there is no code, only data. Gerth's proof system was based on Manna and Pnueli [24] and proves properties expressed in linear temporal logic.

While these problems are beyond the scope and capabilities of our work, the semantics and Hoare-style rules studied in our paper provide a new way of reasoning about sets of programs, including languages (domain-specific or produced by compilers) and self-modifying code.

8 Conclusion

This paper set out to understand why existing compositional program logics for sets of programs [18, 27] fail to be relatively complete, and whether such logics can be modified to achieve completeness. By formalizing verification relative to a denotational semantics for sets of programs, we were able to prove that no complete compositional proof system exists for the kind of logical triples proposed in prior work. In light of this result, we define a more expressive kind of triple and give *the first (two) sound and relatively complete Hoare-style logics* for inductively defined sets of programs (even for programs that contain loops), solving an open problem in the literature [18, 27].

Acknowledgments

This work was supported, in part, by a Microsoft Faculty Fellowship, a gift from Rajiv and Ritu Batra, NSF under grants CCF-1918211, CCF-2023222, CCF-2211968, CCF-2212558, CCF-2402833, CCF-2422214, and CCF-2446711, and a grant from the Korea Foundation of Advanced Studies. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-2038238. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

[7] To be precise, to construct a semantics for VA techniques, one must extend G_{imp} to include tags. This extension is necessary because VA techniques construct different maps for differently tagged sets of programs.

Data-Availability Statement

Proofs of the paper's theorems are available in the full version of this paper [20]. Our paper presents only theoretical results, so there is no accompanying artifact.

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[2] Krzysztof R Apt and Ernst-Rüdiger Olderog. 2019. Fifty years of Hoare's logic. *Formal Aspects of Computing* 31, 6 (2019), 751–807. https://doi.org/10.1007/s00165-019-00501-3

[3] Thomas H. Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. *SIGPLAN Not.* 47, 1 (Jan. 2012), 165–178. https://doi.org/10.1145/2103621.2103677

[4] Paul Maximilian Bittner, Alexander Schultheiß, Benjamin Moosherr, Jeffrey M. Young, Leopoldo Teixeira, Eric Walkingshaw, Parisa Ataei, and Thomas Thüm. 2024. On the Expressive Power of Languages for Static Variability. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 307 (Oct. 2024), 33 pages. https://doi.org/10.1145/3689747

[5] Rod M. Burstall. 1969. Proving Properties of Programs by Structural Induction. *Comput. J.* 12, 1 (1969), 41–48. https://doi.org/10.1093/COMJNL/12.1.41

[6] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. 2007. Certified self-modifying code. *SIGPLAN Not.* 42, 6 (June 2007), 66–77. https://doi.org/10.1145/1273442.1250743

[7] Edmund M Clarke, Jr. 1984. The characterization problem for Hoare logics. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* 312, 1522 (1984), 423–440.

[8] Edmund Melson Clarke Jr. 1979. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *Journal of the ACM (JACM)* 26, 1 (1979), 129–147. https://doi.org/10.1145/322108.322121

[9] Thibault Dardinier and Peter Müller. 2023. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties (extended version). arXiv:2301.10037 [cs.LO]

[10] Martin Erwig and Eric Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *ACM Trans. Softw. Eng. Methodol.* 21, 1, Article 6 (Dec. 2011), 27 pages. https://doi.org/10.1145/2063239.2063245

[11] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices* 53, 4 (2018), 420–435. https://doi.org/10.1145/3296979.3192382

[12] R. Gerth. 1991. Formal Verification of Self Modifying Code. In *Int. Conf. for Young Computer Scientists*.

[13] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. 2008. Modeling and Model Checking Software Product Lines. In *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems* (Oslo, Norway) *(FMOODS '08)*. Springer-Verlag, Berlin, Heidelberg, 113–131. https://doi.org/10.1007/978-3-540-68863-1_8

[14] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330. https://doi.org/10.1145/1925844.1926423

[15] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[16] Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. 2019. Proving unrealizability for syntax-guided synthesis. In *International Conference on Computer Aided Verification*. Springer, 335–352. https://doi.org/10.1007/978-3-030-25540-4_18

[17] Qinheping Hu and Loris D'Antoni. 2018. Syntax-Guided Synthesis with Quantitative Syntactic Objectives. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 386–403. https://doi.org/10.1007/978-3-319-96145-3_21

[18] Jinwoo Kim, Loris D'Antoni, and Thomas Reps. 2023. Unrealizability logic. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 659–688. https://doi.org/10.1145/3571216

[19] Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. 2021. Semantics-guided synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434311

[20] Jinwoo Kim, Shaan Nagy, Thomas Reps, and Loris D'Antoni. 2024. Semantics of Sets of Programs. arXiv:2410.16102 [cs.PL] https://arxiv.org/abs/2410.16102

[21] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[22] Richard J Lipton. 1977. A necessary and sufficient condition for the existence of Hoare logics. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE Computer Society, 1–6. https://doi.org/10.1109/SFCS.1977.1

[23] K. Malmkjær. 1993. *Abstract Interpretation of Partial-Evaluation Algorithms*. Ph.D. Dissertation. Dept. of Comp. and Inf. Sci., Kansas State Univ., Manhattan, Kansas. https://doi.org/10.5555/164793

[24] Zohar Manna and Amir Pnueli. 1983. How to Cook a Temporal Proof System for Your Pet Language. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*. 141–154. https://doi.org/10.1145/567067.567082

[25] U. Möncke and R. Wilhelm. 1991. Grammar Flow Analysis. In *Attribute Grammars, Applications and Systems, (Int. Summer School SAGA) (Lec. Notes in Comp. Sci., Vol. 545)*, H. Alblas and B. Melichar (Eds.). Springer-Verlag, Berlin, Heidelberg, 151–186. https://doi.org/10.1007/3-540-54572-7_6

[26] Magnus O Myreen. 2010. Verified just-in-time compiler on x86. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 107–118. https://doi.org/10.1145/1706299.1706313

[27] Shaan Nagy, Jinwoo Kim, Thomas Reps, and Loris D'Antoni. 2024. Automating unrealizability logic: Hoare-style proof synthesis for infinite sets of programs. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 113–139. https://doi.org/10.1145/3689715

[28] A. Nerode. 1958. Linear automaton transformations. *Proc. Amer. Math. Soc.* 9, 4 (1958), 541–544. https://doi.org/10.1090/s0002-9939-1958-0135681-9

[29] Tobias Nipkow. 2002. Hoare logics for recursive procedures and unbounded nondeterminism. In *International Workshop on Computer Science Logic*. Springer, 103–119. https://doi.org/10.1007/3-540-45793-3_8

[30] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 107–126. https://doi.org/10.1145/2814270.2814310

[31] D. Schmidt. 1986. *Denotational Semantics*. Allyn and Bacon, Inc., Boston, MA. https://web.archive.org/web/20050308103715/http://www.cis.ksu.edu/~schmidt/text/densem.html

[32] Armando Solar-Lezama. 2013. Program sketching. *STTT* 15, 5-6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

[33] J.E. Stoy. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The M.I.T. Press, Cambridge, MA. https://doi.org/10.5555/540155

[34] Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 530–541. https://doi.org/10.1145/2594291.2594340

[35] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 93 (apr 2023), 29 pages. https://doi.org/10.1145/3586045

Received 2024-10-16; accepted 2025-02-18