Sound Gradual Verification with Symbolic Execution

CONRAD ZIMMERMAN, Brown University, USA
JENNA DIVINCENZO, Purdue University, USA
JONATHAN ALDRICH, Carnegie Mellon University, USA

Gradual verification, which supports explicitly partial specifications and verifies them with a combination of static and dynamic checks, makes verification more incremental and provides earlier feedback to developers. While an abstract, weakest precondition-based approach to gradual verification was previously proven sound, the approach did not provide sufficient guidance for implementation and optimization of the required run-time checks. More recently, gradual verification was implemented using symbolic execution techniques, but the soundness of the approach (as with related static checkers based on implicit dynamic frames) was an open question. This paper puts practical gradual verification on a sound footing with a formalization of symbolic execution, optimized run-time check generation, and run time execution. We prove our approach is sound; our proof also covers a core subset of the Viper tool, for which we are aware of no previous soundness result. Our formalization enabled us to find a soundness bug in an implemented gradual verification tool and describe the fix necessary to make it sound.

CCS Concepts: • **Theory of computation** → **Logic and verification**; *Separation logic*.

Additional Key Words and Phrases: gradual verification, symbolic execution, static verification, implicit dynamic frames, soundness proof

ACM Reference Format:
Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2024. Sound Gradual Verification with Symbolic Execution. *Proc. ACM Program. Lang.* 8, POPL, Article 85 (January 2024), 30 pages. https://doi.org/10.1145/3632927

1 INTRODUCTION

Static verification technology based on Hoare-logic-styled pre- and postconditions [Hoare 1969] has come a long way in the last few decades. Such tools can now support the modular verification of data structures that manipulate the heap [Reynolds 2002; Smans et al. 2012] and are recursive [Parkinson and Bierman 2005]. However, verification is expensive, requiring many auxiliary specifications such as loop invariants and lemmas, and often costing an order of magnitude more human effort than development alone. In response, Bader et al. [2018] introduced the idea of gradual verification, which supports the incremental specification and verification of code by seamlessly combining static and dynamic verification. A developer can now write partial, *imprecise* specifications—formulas such as $? * x.f == 2$—backed by run-time checking. During static verification, imprecise specifications are strengthened in support of proof goals when it is necessary and non-contradictory to do so. Then, corresponding dynamic checks are inserted to ensure soundness. As a result, gradual verification allows users to specify and verify only the properties and components of their system that they care about, and incrementally increase the scope of verification as necessary.

Authors' addresses: Conrad Zimmerman, conrad_zimmerman@brown.edu, Brown University, Providence, RI, USA; Jenna DiVincenzo, jennad@purdue.edu, Purdue University, West Lafayette, IN, USA; Jonathan Aldrich, jonathan.aldrich@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, PA, USA.

ACM 2475-1421/2024/1-ART85
https://doi.org/10.1145/3632927

Based on this early idea, Wise et al. [2020] and DiVincenzo et al. [2022] extended gradual verification to support recursive heap data structures. Wise et al. [2020] presented the first theory of gradual verification for *implicit dynamic frames* (IDF) [Smans et al. 2012], a variant of *separation logic* [Reynolds 2002], and *abstract predicates* [Parkinson and Bierman 2005]. Their design, corresponding theory, and proofs rely heavily on the backward-reasoning technique called *weakest liberal preconditions* (WLP), and on infinite sets that are not easy to approximate in finite form. Additionally, Wise et al. [2020]'s design checks all proof obligations at run-time, even when some obligations have been discharged statically. Therefore, it remained unclear how to implement gradual verification and whether gradually-verified programs could achieve good performance. Fortunately, in follow-up work, DiVincenzo et al. [2022] implemented and empirically evaluated Gradual C0, the first gradual verifier that can be used on real programs. Gradual C0 is based on symbolic execution, a forward-reasoning technique which is routinely used in static verifiers such as Viper [Müller et al. 2016], and optimizes run-time checks with statically available information to improve run-time performance. DiVincenzo et al. [2022] showed that this improvement over prior work yields significant performance boosts.

Technically, Gradual C0 is built on top of Viper [Müller et al. 2016], which is a static verification infrastructure and tool that facilitates the development of program verifiers supporting IDF and recursive abstract predicates. Viper also uses symbolic execution at its core. Besides Gradual C0, an array of widely-used verifiers have been built on top of Viper, including Prusti [Astrauskas et al. 2022] for Rust, Nagini [Eilers and Müller 2018] for Python, and VerCors [Blom et al. 2017] for Java. However, despite its prominence, Viper has not been proven sound; nor have, to our knowledge, other symbolic execution-based methods for verifying IDF logics. Thanks to the complexities of symbolic execution and Viper's support for practical but advanced verification features, Schwerhoff [2016]'s specification of Viper is full of implementation details that make it difficult to formally state and prove soundness. Since Gradual C0 is built on Viper, this problem carries over to Gradual C0's specification in DiVincenzo et al. [2022] and is made worse by the combination of static and dynamic checking. Thus DiVincenzo et al. [2022] does not contain a proof of soundness for Gradual C0. Furthermore, since Gradual C0 uses symbolic execution instead of WLP and optimizes run-time checks, Wise et al. [2020]'s proof is also not applicable. This is problematic, because the intricate interactions of static and dynamic checking in gradual verification can easily lead to subtle soundness bugs in gradual verifiers like Gradual C0, as we will show in §8.

Therefore, this paper presents a formal statement and proof of soundness for Gradual C0 and its underlying core subset of Viper. We formalize Gradual C0's symbolic execution algorithm in sets of inference rules, rather than the CPS-style specification in DiVincenzo et al. [2022] and Schwerhoff [2016], to enable abstractions that improve the readability of the design and make it easier to state and prove soundness. The level of abstraction we use is far closer to the implementation of Gradual C0 than Wise et al. [2020]'s formal system, but slightly more abstract than DiVincenzo et al. [2022]'s CPS-style specification, which is littered with implementation details. Reaching the right level of abstraction for our goals took some trial and error. We reflect on this process, including our missteps, in this paper as well. Our approach is inspired by the formal system for a basic type checker combined with symbolic execution in Khoo et al. [2010]. However, we separate the rules into several types of judgements to reflect the architecture of DiVincenzo et al. [2022] and Schwerhoff [2016] and deal with the complexities of IDF and gradual verification. Given an initial symbolic state, the rules compute a next possible state (of which many may exist), and a set of run-time checks required for this transition when optimism is relied upon. That is, our rules are non-deterministic, but only in regards to the multiple execution paths explored by symbolic execution at program points like if statements, while loops, and logical conditionals.

Furthermore, we clearly separate the cases required to support imprecise specifications from those dealing with the underlying verification algorithm supporting only complete static specifications. Therefore, our formal system is a conservative extension of a core calculus of Viper; and so, by formalizing Gradual C0 and proving it sound, we have also formalized the core of Viper and proved it sound. To make it easier for readers of this paper to take advantage of our formal statement and proof of soundness for Viper for their own uses, we present first a core language, which we call SVL_{C0}, along with verification rules modeling Gradual C0's underlying static verification algorithm. We then define GVL_{C0}, which extends SVL_{C0} to include gradual specifications and corresponds to the full language used by Gradual C0. We also formally define static verification for GVL_{C0}, modeling the verification algorithm of Gradual C0. We hope this separation provides a solid foundation for future proof endeavors of other static verifiers based on symbolic execution.

In order to fully define the behavior of GVL_{C0} and its subset SVL_{C0}, we specify its dynamic semantics, which combines the semantics of C_0 [Arnold 2010] with the dynamic semantics of GVL_{RP}, the language used to define the theory of gradual verification with recursive predicates in Wise et al. [2020]. The C_0 programming language is a core, safe variant of the C language introduced for education [Arnold 2010] and is also supported by Gradual C0. C_0 allows specification of the pre- and post-conditions of methods, but does not include constructs necessary for static verification using IDF. Thus we add the dynamic semantics from Wise et al. [2020] for IDF specifications, recursive predicates, and imprecise specifications. These semantics assert the validity of every specification at run-time, ensuring both memory safety and functional correctness of programs. Thus these semantics provide a foundation against which we can establish the soundness of Gradual C0's symbolic execution algorithm. That is, we prove that when all run-time checks produced by the symbolic execution algorithm are satisfied, then the program is guaranteed to dynamically execute successfully. A tricky part of this proof is defining a *valuation function* [Khoo et al. 2010], which is a partial function mapping symbolic values from symbolic execution to their concrete values for a specific execution trace from program execution. This function is used to state the correspondence between symbolic and concrete execution states. While we start with Khoo et al. [2010]'s simplistic valuation function, we end up with one that is far more complex as it additionally connects *isorecursive* symbolic predicates from static verification with their *equirecursive* counterparts in dynamic verification and handles global invariants such as separation and access permissions from IDF. This proof technique allows our formal system and reasoning to match the implementation more closely than other techniques such as the evidence calculus used in Garcia et al. [2016]. This enables us to explore future developments using either the implementation or formalization, and easily update the other to ensure we remain both implementable and sound.

Finally, we present and discuss a soundness bug we found in Gradual C0 during our proof work and have since communicated to DiVincenzo et al. [2022]. The bug is a specific interaction caused by reducing run-time checks using statically available information in isorecursive predicates, and then checking the remaining run-time checks using equirecursive predicates. This bug could not have arisen in Wise et al. [2020]'s work as their gradual verification approach checks all proof obligations at run time. We explore several options for addressing this soundness bug, explain our chosen method in detail, and discuss an implementation fix. Despite DiVincenzo et al. [2022]'s thorough empirical evaluation and testing of Gradual C0, this bug was never discovered in their testing. This is likely due to the subtle, intricate interactions between verification technologies in gradual verification that are hard to test. This demonstrates the value of formally proving soundness in the case of gradual verification, and we hope this paper serves as a basis for similar future work. To summarize, this paper makes the following contributions:

- Formalization and proof of soundness for Gradual C0, the first gradual verifier for recursive heap data structures that is based on symbolic execution [DiVincenzo et al. 2022]. The level of abstraction chosen for this proof work improves the readability of Gradual C0's design and makes adapting this work to prove other symbolic execution-based gradual verifiers sound much easier.
- Formalization of a core subset of the Viper static verifier, which is based on symbolic execution and supports IDF. This work provides the first solid foundation for proof work on static verifiers that use symbolic execution and IDF.
- A reflection on the trial and error of picking the right level of abstraction for our proof work in this paper.
- Demonstration of a soundness bug we found in Gradual C0 during our work and have since communicated to DiVincenzo et al. [2022]. We also provide several options for addressing this bug and advise on how to implement one of our solutions.

2 SVL_{C0}

We first introduce SVL_{C0} and a corresponding static verification algorithm. Since it does not include imprecise specifications, SVL_{C0} can be verified by existing static verification tools such as Viper [Müller et al. 2016]. The verification algorithm corresponds to the core algorithm of Viper, which is the foundation for static verification in Gradual C0. We illustrate how our formalism and soundness result can be applied to Viper. In later sections we extend SVL_{C0}'s verification algorithm to support the verification of gradually-specified GVL_{C0} programs.

2.1 Definition

We define an abstract syntax for SVL_{C0} in Figure 1. Its form is similar to the language of Viper, which is intended for use as a generic backend for multiple frontend languages; however, we use the syntax of C_0.

Programs consist of struct, predicate, and method[1] definitions, and an entry statement. Struct definitions contain a list of fields, predicate definitions contain a parameter list and a formula (the *predicate body*), and method definitions contain a parameter list, a return type, a pre-condition (denoted by `requires`), a post-condition (denoted by `ensures`), and a statement (the *method body*). The entry statement represents the body of the `main` method in traditional C programs. Statements in SVL_{C0} follow C conventions, except for `while`, `alloc`, and `return`. All `while` statements specify a formula called a *loop invariant*, which states the properties preserved by the loop during execution. An `alloc` statement allocates new memory on the heap, initializes it with a default value, and updates the variable on the left-hand side to contain a reference to the newly allocated value. This matches C_0 semantics, except C_0 returns a *pointer*, not a *reference*, and thus the type of the variable is written differently. We omit `return` statements; instead, the method body must assign to a special `result` variable, whose value is then returned after executing the method body. This reflects the behavior of Gradual Viper which also does not have a `return` statement. Additionally, we simplify several statements to make formal definitions and proofs easier. For example, assignment only occurs to a variable or a field of a variable; statements such as `x.y.z = 1` are not permitted. We also omit `void` method calls, since these do not differ meaningfully from calls to value-returning methods.

Like Gradual C0 [DiVincenzo et al. 2022], SVL_{C0} does not support arrays. Verifying non-trivial properties of programs that use arrays would require significant extensions to existing gradual

[1]To distinguish them from pure functions (which are used in the specification language of similar verification tools) we use *method* to refer to any potentially impure function.

$x \in \textsc{Var}$ *Variable names*
$f \in \textsc{Field}$ *Field names*
$p \in \textsc{Predicate}$ *Predicate names*
$m \in \textsc{Method}$ *Method names*
$S \in \textsc{Struct}$ *Struct names*
$n \in \mathbb{Z}$ *Integers*

$\Pi ::= \overline{\mathcal{S}}\ \overline{\mathcal{P}}\ \overline{\mathcal{M}}\ s$
$\mathcal{S} ::= \texttt{struct}\ S\ \{\ \overline{T\ f}\ \}$
$\mathcal{P} ::= p(\overline{T\ x}) = \phi$
$\mathcal{M} ::= T\ m(\overline{T\ x})\ \Phi\ \{\ s\ \}$

$\Phi ::= \texttt{requires}\ \phi\ \texttt{ensures}\ \phi$
$T ::= S \mid \texttt{int} \mid \texttt{bool} \mid \texttt{char}$
$s ::= s;\ s \mid \texttt{skip} \mid x = e \mid x = \texttt{alloc}(S) \mid$
$\quad x = m(\overline{e}) \mid \texttt{assert}\ \phi \mid \texttt{fold}\ p(\overline{e}) \mid$
$\quad \texttt{unfold}\ p(\overline{e}) \mid \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s \mid$
$\quad \texttt{while}\ e\ \texttt{invariant}\ \phi\ \texttt{do}\ s$
$e ::= l \mid x \mid e.f \mid e \oplus e \mid e \mid\mid e \mid e\ \texttt{\&\&}\ e \mid \texttt{!}\ e$
$l ::= n \mid \texttt{null} \mid \texttt{true} \mid \texttt{false}$
$\oplus ::= \texttt{+} \mid \texttt{-} \mid \texttt{/} \mid \texttt{*} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{>}$
$\phi ::= \phi * \phi \mid p(\overline{e}) \mid e \mid \texttt{acc}(e.f) \mid$
$\quad \texttt{if}\ e\ \texttt{then}\ \phi\ \texttt{else}\ \phi$

Fig. 1. Abstract syntax for SVL_{C0}

verification theory – for example, quantified formulas. These extensions are left to future work. However, we can verify recursive data structures such as linked lists with abstract predicates. Note that Viper does support quantified formulas and arrays, thus further work is necessary to formally prove soundness of these capabilities.

We make several simplifying assumptions for SVL_{C0} programs. All variables are initialized before they are used, and every execution path for a method body assigns the `result` variable at its end. Every program is well-typed; that is, expressions used in `if` conditions or as boolean operands will evaluate to `bool` values, all arguments passed to method parameters will match the defined parameter type, and the value assign to `result` has type equal to the method's return type. Finally, all specifications (predicate bodies, loop invariants, and method pre- and post-conditions) are *self-framed*, which is a special well-formedness condition from IDF that we define later.

Formulas (specifications) in SVL_{C0} are written in the logic of IDF [Smans et al. 2012] and recursive predicates [Parkinson and Bierman 2005]. Thus formulas may contain expressions as well as abstract predicates and accessibility predicates from IDF; formulas may be joined by the separating conjunction $*$ [Smans et al. 2012]. An accessibility predicate $\texttt{acc}(e.f)$ requires access to the heap location $e.f$. A predicate instance $p(\overline{e})$ applies the boolean predicate p to the arguments $\overline{e}$. An expression e requires that e evaluates to `true`. A separating conjunction, as in $\phi_1 * \phi_2$, acts like a logical AND for ϕ_1 and ϕ_2, but also requires the heap locations specified by predicates and accessibility predicates in ϕ_1 to be disjoint from those specified in ϕ_2, e.g. $\texttt{acc}(x.f) * \texttt{acc}(y.f)$ implies x `!=` y. A conditional formula `if` e `then` ϕ_1 `else` ϕ_2 denotes the validity of ϕ_1 when e evaluates to `true`; otherwise it denotes the validity of ϕ_2.

Formulas in IDF, and thus in SVL_{C0}, must be *self-framed* [Smans et al. 2012], which requires permissions for all heap locations used in a formula to also be in that formula. For example, `x.value == 0` is not self-framed since it references the heap location `x.value`, but does not assert accessibility of the field `x.value`. However, `acc(x.value) * x.value == 0` is self-framed. We specify rules for framing and self-framing in §4.3.

Static verification of predicates is done *isorecursively* [Summers and Drossopoulou 2013], thus predicate instances must be explicitly folded before they can be asserted. Similarly, predicate bodies must be explicitly unfolded before asserting the implications of a predicate. This enables static verification of recursive predicates and simplifies reasoning about the verifier's behavior.

2.2 Representation

In this section, we formally define the data structures used during static verification of SVL_{C0} programs.

- A *symbolic value* $v \in$ SVALUE is an abstract value representing an unknown value, such as an integer or object reference. We leave the concrete type of SVALUE undefined, but assume that an infinite number of distinct new values can be produced by a fresh function.
- A *symbolic expression* $t \in$ SEXPR is a symbolic or literal value, or is composed of other symbolic expressions and operators.

$$t \quad ::= \quad v \quad | \quad l \quad | \quad !\,t \quad | \quad t_1 \,\&\&\, t_2 \quad | \quad t_1 \,||\, t_2 \quad | \quad t_1 \oplus t_2$$

- A *path condition* $g \in$ SEXPR is a symbolic expression composed of conjuncts identifying a particular execution path. Conjuncts are added at every conditional branch during symbolic execution.
- A *field chunk* $\langle f, t, t' \rangle \in$ SFIELD represents, in the symbolic heap, the field f of an object reference t containing a value t'. A heap chunk is roughly approximate to the *points to* construct in separation logic [Reynolds 2002]. A *predicate chunk* $\langle p, \bar{t} \rangle \in$ SPREDICATE represents an isorecursive instance of a predicate p with arguments $\bar{t}$. Together, field chunks and predicate chunks are called *heap chunks*.
- A *symbolic heap* $\mathsf{H} \in \mathcal{P}(\text{SFIELD} \cup \text{SPREDICATE})$ is a finite set of heap chunks. All heap chunks that it contains must represent distinct locations in the heap at run time.
- A *symbolic state* $\sigma \in$ SSTATE is a tuple containing a path condition (referenced by $g(\sigma)$), a symbolic heap (referenced by $\mathsf{H}(\sigma)$), and a symbolic environment (referenced by $\gamma(\sigma)$). A symbolic state stores all values for a particular point during symbolic execution. The symbol σ_{empty} represents an empty symbolic state, i.e. $g(\sigma_{\text{empty}}) = \texttt{true}$ and $\mathsf{H}(\sigma_{\text{empty}}) = \gamma(\sigma_{\text{empty}}) = \emptyset$.
- A *verification state* Σ represents a particular point during static verification. It is either a special symbol or a triple $\langle \sigma, s, \tilde{\phi} \rangle$ consisting of a symbolic state σ, a statement s that remains to be executed, and a formula $\tilde{\phi}$ that must be asserted after executing s. $\sigma(\Sigma)$, $s(\Sigma)$, and $\tilde{\phi}(\Sigma)$ are used to reference a specific component of Σ when Σ is not a symbol.

$$\Sigma \quad ::= \quad \mathsf{init} \quad | \quad \mathsf{final} \quad | \quad \langle \sigma, s, \tilde{\phi} \rangle$$

- A *valuation* V : SVALUE $\rightarrow$ VALUE is a mapping from symbolic values to concrete values (defined in §4.1). Valuations are implicitly extended to be defined for all SEXPR, following the structure of symbolic expressions.
  A symbolic expression t *implies* the symbolic expression t' (written $t \implies t'$) if, for all valuations V, $V(t) = \texttt{true} \implies V(t') = \texttt{true}$. For example, $t_1 \,\&\&\, t_2 \implies t_2$. A symbolic expression t is *satisfiable*, denoted $\text{sat}(t)$, if $V(t) = \texttt{true}$ for some valuation V.

2.3 Evaluating Expressions

Symbolic execution evaluates an expression e to a symbolic value t using the symbolic state σ, and is denoted by the judgement $\sigma \vdash e \Downarrow t \dashv \sigma'$. It also yields a new symbolic state σ' which may contain a more specific path condition if this particular evaluation short-circuits a boolean operator. Selected formal rules for symbolic evaluation are given in Figure 2. Literals are evaluated to themselves and variables are evaluated to the corresponding value in the symbolic store. Some operators, such as negation and arithmetic operators, are directly translated into a symbolic expression using the respective operator. In contrast, boolean operators are short-circuiting: when evaluating $e_1 \,\&\&\, e_2$, if e_1 evaluates to `false`, then e_2 is never evaluated (in this case, $e_1 == \texttt{false}$ is added to the path condition). We define two non-deterministic rules for each binary boolean operator—SEVALANDA represents the short-circuiting case just described, while SEVALANDB represents the non-short-circuiting case where e_1 is true, so e_2 must also be evaluated to determine the result. Finally, field

$$\textsc{SEvalLiteral}\ \frac{}{\sigma \vdash l \Downarrow l \dashv \sigma} \qquad \textsc{SEvalVar}\ \frac{}{\sigma \vdash x \Downarrow \gamma(\sigma)(x) \dashv \sigma} \qquad \textsc{SEvalNeg}\ \frac{\sigma \vdash e \Downarrow t \dashv \sigma'}{\sigma \vdash\ !\, e \Downarrow\ !\, t \dashv \sigma'} \qquad \textsc{SEvalAndA}\ \frac{\sigma \vdash e_1 \Downarrow t_1 \dashv \sigma'}{\sigma \vdash e_1 \,\&\&\, e_2 \Downarrow t_1 \dashv \sigma'[g = g(\sigma')\ \&\&\ !\, t_1]}$$

$$\textsc{SEvalAndB}\ \frac{\sigma \vdash e_1 \Downarrow t_1 \dashv \sigma' \qquad \sigma'[g = g(\sigma')\ \&\&\ t_1] \vdash e_2 \Downarrow t_2 \dashv \sigma''}{\sigma \vdash e_1 \,\&\&\, e_2 \Downarrow t_2 \dashv \sigma''} \qquad \textsc{SEvalField}\ \frac{\sigma \vdash e \Downarrow t_e \dashv \sigma' \qquad g(\sigma') \implies t_e == t'_e \qquad \langle t'_e, f, t\rangle \in \mathrm{H}(\sigma')}{\sigma \vdash e.f \Downarrow t \dashv \sigma'}$$

$$\textsc{SEvalPCAnd}\ \frac{\sigma \vdash e_1 \downarrow t_1 \qquad \sigma \vdash e_2 \downarrow t_2}{\sigma \vdash e_1 \,\&\&\, e_2 \downarrow t_1\ \&\&\ t_2 \dashv \sigma''}$$

Fig. 2. Selected symbolic evaluation rules

references are evaluated to the symbolic value contained in their corresponding field chunk in the symbolic heap. Note, a heap chunk for the field reference must be in the heap, otherwise evaluation fails (and ultimately static verification as well), thus the field reference must be framed by the current state.

We also define a judgment of the form $\sigma \vdash e \downarrow t$ which symbolically evaluates an expression e to a symbolic expression t without short-circuiting. Thus the judgment is deterministic and does not update the path condition. Instead, logical operators such as && are encoded directly in the symbolic expression (compare SEvalPCAnd with SEvalAndA/SEvalAndB in Figure 2). This results in a less specific path condition, but reduces the number of execution paths during symbolic execution. This matches the evaluation method described in DiVincenzo et al. [2022] for evaluation in formulas, while the former style is used for evaluation in imperative code.

2.4 Consuming Formulas

Given a symbolic state σ and formula ϕ, *consuming* a formula ϕ first asserts that ϕ is established by σ, and second removes the heap chunks in σ corresponding to permissions (predicates and accessibility predicates) in ϕ. The judgment $\sigma \vdash \phi \rhd \sigma'$ denotes consumption; i.e., ϕ is consumed from σ, resulting in the new symbolic state σ'. See Figure 3 for selected rules.

Consuming an accessibility predicate such as $\mathtt{acc}(e.f)$ first asserts the predicate has a corresponding field chunk in the heap, and second removes the chunk from the heap (SConsumeAcc). Consuming a predicate similarly looks for and removes the corresponding predicate chunk from the heap (SConsumePredicate). If any of the chunks are missing from the heap, then verification fails. Expressions must evaluate to `true` in the current symbolic execution path. That is, the current path condition must imply the symbolic value of the expression (SConsumeValue). As mentioned previously and seen in the aforementioned rule, expressions in formulas are evaluated with the deterministic evaluation judgment (i.e., not the short-circuiting one), which matches the behavior described in DiVincenzo et al. [2022] and reduces the number of branches generated during symbolic execution. This differs from Viper, which uses a single, short-circuiting `eval` algorithm everywhere, including in `consume`. A separating conjunction, such as $\phi_1 * \phi_2$, is consumed left-to-right, i.e. ϕ_1 is consumed and then ϕ_2 is consumed (SConsumeConjunction). This enforces the separation of permissions between the two conjuncts – heap chunks necessary to satisfy the permissions asserted in ϕ_1 will be removed before consuming ϕ_2, so, if they overlap, consumption of ϕ_2 will fail. Finally, we define consumption of logical conditionals, like $\mathtt{if}\ e\ \mathtt{then}\ \phi_1\ \mathtt{else}\ \phi_2$, in two non-deterministic rules. In SConsumeConditionalA, e is assumed to be true in the path condition and ϕ_1 is consumed. Likewise, in SConsumeConditionalB, e is assumed to be false in the path condition and ϕ_2 is consumed.

Note, as we saw in §2.3, evaluation of a field access in an expression requires the state to contain a heap chunk for the field. But consume removes heap chunks from the state in a left-to-right manner thanks to rules SConsumeAcc and SConsumeConjunction. For example, we may want

$$\textsc{SConsume}\ \frac{\sigma, \sigma \vdash \phi \rhd \sigma'}{\sigma \vdash \phi \rhd \sigma'} \qquad \textsc{SConsumeValue}\ \frac{\sigma_E \vdash e \downarrow t \quad g(\sigma) \implies t}{\sigma, \sigma_E \vdash e \rhd \sigma}$$

$$\textsc{SConsumeAcc}\ \frac{\sigma_E \vdash e \downarrow t_e \quad g(\sigma) \implies t_e == t'_e \quad \mathsf{H}(\sigma) = \{\langle f, t'_e, t\rangle\} \uplus \mathsf{H}'}{\sigma, \sigma_E \vdash \mathsf{acc}(e.f) \rhd \sigma[\mathsf{H} = \mathsf{H}']}$$

$$\textsc{SConsumePredicate}\ \frac{\sigma_E \vdash e \downarrow t \quad g(\sigma) \implies t == t' \quad \mathsf{H}(\sigma) = \{\langle p, \overline{t'}\rangle\} \uplus \mathsf{H}'}{\sigma, \sigma_E \vdash \sigma[\mathsf{H} = \mathsf{H}']}$$

$$\textsc{SConsumeConjunction}\ \frac{\sigma, \sigma_E \vdash \phi_1 \rhd \sigma' \quad \sigma', \sigma_E[g = g(\sigma')] \vdash \phi_2 \rhd \sigma''}{\sigma, \sigma_E \vdash \phi_1 * \phi_2 \rhd \sigma''}$$

$$\textsc{SConsumeConditionalA}\ \frac{\sigma_E \vdash e \downarrow t \quad g' = g(\sigma)\ \&\&\ t \quad \sigma[g = g'], \sigma_E[g = g'] \vdash \phi_1 \rhd \sigma'}{\sigma, \sigma_E \vdash \texttt{if}\ e\ \texttt{then}\ \phi_1\ \texttt{else}\ \phi_2 \rhd \sigma'}$$

$$\textsc{SConsumeConditionalB}\ \frac{\sigma_E \vdash e \downarrow t \quad g' = g(\sigma)\ \&\&\ \neg t \quad \sigma[g = g'], \sigma_E[g = g'] \vdash \phi_2 \rhd \sigma'}{\sigma, \sigma_E \vdash \texttt{if}\ e\ \texttt{then}\ \phi_1\ \texttt{else}\ \phi_2 \rhd \sigma'}$$

Fig. 3. Selected consume rules

to consume the formula $\mathsf{acc}(e.f) * e.f == 0$. First, a heap chunk for $\mathsf{acc}(e.f)$ is found and removed from the heap. Then, the resulting state is used to frame and evaluate $e.f == 0$ in the next consume step. However, the heap chunk for $e.f$ was removed from the state so evaluation fails when it shouldn't since the original state contained the heap chunk. To solve this issue, we define consume using an underlying judgment, denoted $\sigma, \sigma_E \vdash \phi \rhd \sigma'$, which asserts and removes permissions from σ while evaluating expressions with the unchanging reference state σ_E. The state σ_E is the symbolic state before consumption. The rule SConsume defines the top-level consume judgment using this new underlying judgment.

Our consume judgment represents the core functionality of DiVincenzo et al. [2022] and Schwerhoff [2016]'s `consume` algorithms. We, of course, ignore unnecessary implementation details like snapshots, which preserve certain portions of the state that are removed during consume.

2.5 Producing Formulas

Given an initial state σ and formula ϕ, *producing* ϕ adds the information in ϕ into the symbolic state σ, resulting in a new state σ'. The judgment for $\sigma \vdash \phi \lhd \sigma'$ denotes production; i.e., ϕ is produced into the state σ, resulting in σ'. In particular, produce adds heap chunks representing predicates in ϕ to the symbolic heap and symbolic expressions representing constraints from boolean expressions in ϕ to the path condition in a left-to-right manner. Note, each symbolic heap chunk represents a distinct region of memory at run-time, an invariant that we later prove. Thus overlapping heap chunks may only occur in symbolic states which represent an unreachable dynamic state and can safely be ignored. When producing formulas, we use deterministic symbolic evaluation for expressions, but we introduce separate execution paths for conditionals (similar to §2.4).

Formal rules are given in the supplement [Zimmerman et al. 2024]. These rules capture the functionality of the `produce` algorithm specified in DiVincenzo et al. [2022] and Schwerhoff [2016]. As noted in the previous section, Schwerhoff [2016] uses short-circuiting evaluation in all places, while we use deterministic evaluation.

2.6 Executing Statements

Now that we have formally defined symbolic execution of expressions and formulas, we can put the pieces together to define symbolic execution of program statements.

We represent the symbolic execution of program statements as small-step execution rules denoted by the judgment $\sigma \vdash s \longrightarrow s' \dashv \sigma'$, where the initial statement s is symbolically executed with the initial state σ, resulting in the state σ', and then transitions to the next statement s' with the new state σ'. Selected formal rules are shown in Figure 4. Executing a variable assignment updates the symbolic store (SExecAssign); while executing a field assignment first consumes $\mathsf{acc}(x.f)$, and then adds a new heap chunk for $x.f$ to the heap that contains $x.f$'s new symbolic value after

$$\textsc{SExecAssign}\ \frac{\sigma \vdash e \Downarrow t \dashv \sigma' \qquad \gamma' = \gamma(\sigma)[x \mapsto t]}{\sigma \vdash x = e;\ s \to s \dashv \sigma'[\gamma = \gamma']}$$

$$\textsc{SExecAssignField}\ \frac{\sigma \vdash e \Downarrow t \dashv \sigma' \qquad \sigma' \vdash \mathsf{acc}(x.f) \rhd \sigma'' \qquad \mathsf{H}' = \mathsf{H}(\sigma'') \cup \{\langle f, \gamma(\sigma'')(x), t\rangle\}}{\sigma \vdash x.f = e;\ s \to s \dashv \sigma''[\mathsf{H} = \mathsf{H}']}$$

$$\textsc{SExecAlloc}\ \frac{t = \text{fresh} \qquad \overline{T\ f} = \text{struct}(S) \qquad \mathsf{H}' = \mathsf{H}(\sigma) \cup \{\overline{\langle f, t, \text{default}(T)\rangle}\}}{\sigma \vdash x = \mathtt{alloc}(S);\ s \to s \dashv \sigma[\mathsf{H} = \mathsf{H}']}$$

$$\textsc{SExecCall}\ \frac{\sigma \vdash e \Downarrow t \dashv \sigma' \qquad \overline{x} = \text{params}(m) \qquad \sigma'[\gamma = [\overline{x \mapsto t}]] \vdash \text{pre}(m) \rhd \sigma'' \qquad t' = \text{fresh} \qquad \gamma' = \gamma(\sigma')[y \mapsto t'] \qquad \sigma''[\gamma = [\overline{x \mapsto t}, \mathtt{result} \mapsto t']] \vdash \text{post}(m) \lhd \sigma'''}{\sigma \vdash y = m(\overline{e});\ s \to s \dashv \sigma'''[\gamma = \gamma']}$$

$$\textsc{SExecIfA}\ \frac{\sigma \vdash e \Downarrow t \dashv \sigma' \qquad \sigma'' = \sigma'[g = g(\sigma')\ \&\&\ t]}{\sigma \vdash \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2;\ s \to s_1;\ s \dashv \sigma''}$$

$$\textsc{SExecIfB}\ \frac{\sigma \vdash e \Downarrow t \dashv \sigma' \qquad \sigma'' = \sigma'[g = g(\sigma')\ \&\&\ !\ t]}{\sigma \vdash \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2;\ s \to s_2;\ s \dashv \sigma''}$$

$$\textsc{SExecWhile}\ \frac{\sigma \vdash \phi \rhd \sigma' \qquad \overline{x} = \text{modified}(s) \qquad \sigma'[\gamma = \gamma(\sigma')[\overline{x \mapsto \text{fresh}}]] \vdash \phi \lhd \sigma'' \qquad \sigma'' \vdash e \downarrow t}{\sigma \vdash \mathtt{while}\ e\ \mathtt{invariant}\ \phi\ \mathtt{do}\ s;\ s' \to s' \dashv \sigma''[g = g(\sigma'')\ \&\&\ !\ t]}$$

Fig. 4. Selected symbolic execution rules

the write (SExecAssignField). An `alloc`(S) statement adds a heap chunk for each field in S to the symbolic heap. The new object reference is a fresh value but the new field chunks are each initialized with default values, which reflects the behavior of C_0 (SExecAlloc). Execution rules for `if` statements are non-deterministic: given a statement `if` e `then` s_1 `else` s_2, SExecIfA adds e to the path condition and continues execution with s_1, while SExecIfB adds `!` e to the path condition and continues execution with s_2.

Symbolic execution of method calls is modular; i.e., the behavior of the method call is represented by the method's pre- and post-conditions (SExecCall). First, the method's arguments are evaluated to symbolic values. Then the pre-condition is consumed using a special environment containing the argument values. A fresh symbolic value is added to represent the return value of the method, and then the post-condition of the method is produced. The special environment is then replaced by the original environment, with the addition of the result's symbolic value. Loops (i.e `while` statements) are executed similarly: the loop invariant is consumed, variables modified by the loop body are set to fresh values in the symbolic store, the loop invariant is produced, and the negated loop condition is added to the path condition (SExecWhile). Execution of the `fold` and `unfold` statements is also similar to loops and method calls: `fold` consumes the predicate body and adds a representative predicate chunk to the symbolic heap, while `unfold` consumes the predicate instance (thus removing the predicate chunk from the heap) and produces the predicate body.

2.7 Modularly Verifying Programs

We now define verification of entire programs. We start by defining what a program Π is; it is a quadruple $\langle s, M, P, S\rangle$ where s is the entry statement of the program, M is the set of method names, P is the set of predicate names, and S is the set of struct names in the program. Then, we define the judgment $\Pi \vdash \Sigma \to \Sigma'$ that specifies all possible symbolic execution steps that occur during verification of Π. Selected rules are given in Figure 5.

A verification state Σ is *reachable* from program Π if $\Sigma = \mathsf{init}$ or $\Pi \vdash \Sigma_0 \to \Sigma$ for some reachable Σ_0. The latter judgement only holds when Σ_0 is itself reachable.

This judgement includes rules for modular verification. From init, we can begin verification of the entry statement (SVerifyInit) or of any method (SVerifyMethod). When verifying a method, the

$$\textsc{SVerifyInit}\ \frac{}{\langle s, M, P, S\rangle \vdash \mathsf{init} \rightarrow \langle \sigma_{\text{empty}}, s, \mathsf{true}\rangle}$$

$$\textsc{SVerifyMethod}\ \frac{m \in M \qquad \overline{x} = \text{params}(m) \qquad \sigma_{\text{empty}}[\gamma = [\overline{x \mapsto \text{fresh}}]] \vdash \text{pre}(m) \triangleleft \sigma}{\langle s, M, P, S\rangle \vdash \mathsf{init} \rightarrow \langle \sigma, \text{body}(m)\,;\, s, \text{post}(m)\rangle}$$

$$\textsc{SVerifyLoopBody}\ \frac{\Pi \vdash \_ \rightarrow \langle \sigma_0, \mathtt{while}\ e\ \mathtt{invariant}\ \tilde{\phi}\ \mathtt{do}\ s\,;\, s', \tilde{\phi}_0\rangle \quad \langle \bot, \gamma(\sigma_0)[\overline{x \mapsto \text{fresh}}], \emptyset, \emptyset, g(\sigma_0)\rangle \vdash \tilde{\phi} \triangleleft \sigma \quad \overline{x} = \text{modified}(s) \quad \sigma \vdash e \downarrow t \dashv \mathcal{R}}{\Pi \vdash \langle \sigma_0, \mathtt{while}\ e\ \mathtt{invariant}\ \tilde{\phi}\ \mathtt{do}\ s\,;\, s', \tilde{\phi}_0\rangle \rightarrow \langle \sigma[g = g(\sigma)\ \&\&\ t],\, s\,;\, \mathtt{skip},\, \tilde{\phi}\rangle}$$

$$\textsc{SVerifyLoop}\ \frac{\Pi \vdash \_ \rightarrow \langle \sigma_0, \mathtt{while}\ e\ \mathtt{invariant}\ \tilde{\phi}\ \mathtt{do}\ s\,;\, s', \tilde{\phi}_0\rangle \quad \sigma_0 \vdash \tilde{\phi} \triangleright \sigma_0', \_ \quad \sigma_0'[\gamma = \gamma(\sigma_0)[\overline{x \mapsto \text{fresh}}]] \vdash \tilde{\phi} \triangleleft \sigma_0'' \quad \overline{x} = \text{modified}(s)}{\Pi \vdash \langle \sigma_0, \mathtt{while}\ e\ \mathtt{invariant}\ \tilde{\phi}\ \mathtt{do}\ s\,;\, s', \tilde{\phi}_0\rangle \rightarrow \langle \sigma_0, \mathtt{while}\ e\ \mathtt{invariant}\ \tilde{\phi}\ \mathtt{do}\ s\,;\, s', \tilde{\phi}_0\rangle}$$

$$\textsc{SVerifyStep}\ \frac{\Pi \vdash \_ \rightarrow \langle \sigma, s, \phi\rangle \qquad \sigma \vdash s \rightarrow s' \dashv \sigma'}{\Pi \vdash \langle \sigma, s, \phi\rangle \rightarrow \langle \sigma', s', \phi\rangle}$$

$$\textsc{SVerifyFinal}\ \frac{\Pi \vdash \_ \rightarrow \langle \sigma, \mathtt{skip}, \phi\rangle \qquad \sigma \vdash \phi \triangleright \sigma',}{\Pi \vdash \langle \sigma, \mathtt{skip}, \phi\rangle \rightarrow \text{final}}$$

Fig. 5. Selected verification rules

method's post-condition is used as the formula of the verification state. After completely executing the method's body, i.e. having reached `skip`, we consume the formula contained in the verification state (SVerifyFinal), which is the method's post-condition.

We modularly verify loop bodies following a similar pattern. As described in §2.6, symbolic execution steps over loop bodies in the same way it steps over method calls. However, we introduce a verification rule (SVerifyLoopBody) that allows symbolic execution of a loop body, beginning with a new symbolic state. We reuse the symbolic store from the initial symbolic state, except that all variables modified by the loop body are replaced by fresh values. Verification proceeds similar to method verification, except that we use the loop invariant for the formula of the new verification state—we produce the loop invariant, symbolically execute the loop body, and finally consume the loop invariant. Thus symbolic execution, which steps over the loop, ensures that the loop invariant holds for the initial iteration, while this verification rule ensures that the loop invariant is preserved after every iteration.

We also include another verification rule for loops, SVerifyLoop, in order to match the behavior of Gradual C0. This rule and its correspondence with Gradual C0 is described further in §7.2.

Statements are executed by symbolic execution as described in §2.6. Given a reachable verification state $\langle \sigma, s, \phi\rangle$ and the symbolic execution $\sigma \vdash s \rightarrow s' \dashv \sigma'$, the state $\langle \sigma', s', \phi\rangle$ is reachable, i.e. $\Pi \vdash \langle \sigma, s, \phi\rangle \rightarrow \langle \sigma', s', \phi\rangle$.

2.8 Example

We now illustrate verification of the `append` method defined in Figure 6, which appends a given value to the end of a list using recursion. The `append` method is ensured to be memory safe and preserve acyclicity of the list through verification. We begin with an empty state and initialize all parameters with fresh values:

$\sigma_1 = \langle \emptyset, \gamma, \mathsf{true}\rangle \quad \gamma = [\mathtt{l} \mapsto \nu_1, \mathtt{v} \mapsto \nu_2]$

Then the pre-condition `acyclic(l) * l != NULL` is produced:

$\sigma_2 = \langle \{\langle \mathtt{acyclic},\ \nu_1\rangle\}, \gamma,\ \nu_1\ \mathtt{!=}\ \mathtt{null}\rangle$

Unfolding `acyclic(l)` (line 17) consumes the predicate from the state and produces its body. The body of `acyclic(l)` contains a logical conditional resulting in two possible execution paths for produce – one where ν_4 is null and one where ν_4 is not null, where ν_4 is the symbolic value for `l.next`:

```
1  struct List { int value; List next }
2
3  predicate acyclic(List l) =
4    acc(l.value) * acc(l.next) *
5    (if l.next == NULL then true
6     else acyclic(l.next))
7
8  List singleton(int value)
9    requires true
10   ensures (acyclic(result) *
11            result != NULL)
12 {  ···  }
```

```
13 List append(List l, int value)
14   requires acyclic(l) * l != NULL
15   ensures acyclic(result) * result != NULL
16 {
17   unfold acyclic(l);
18   if (l.next == NULL)
19     n = singleton(value);
20   else
21     n = append(l.next, value);
22   l.next = n;
23   fold acyclic(l);
24   result = l;
25 }
```

Fig. 6. Code and supporting declarations for appending to an acyclic linked list

```
17 unfold acyclic(l);
```
$\sigma_{A3} = \langle\{\langle \mathsf{value}, \nu_1, \nu_3\rangle, \langle \mathsf{next}, \nu_1, \nu_4\rangle\}, \gamma, \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ == null}\rangle$

$\sigma_{B3} = \langle\{\langle \mathsf{value}, \nu_1, \nu_3\rangle, \langle \mathsf{next}, \nu_1, \nu_4\rangle, \langle \mathsf{acyclic}, \nu_4\rangle\}, \gamma, \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ != null}\rangle$

We follow both execution paths, using color-coding to distinguish them. Next, when executing the `if` statement (line 18), we first evaluate the condition. Since `l.next` is framed by the state, evaluation of the condition succeeds and execution branches along the `if`. We first consider executing the `then` branch of the `if`, where ν_4 == `null` is added it to the path condition:

```
18 if (l.next == NULL)
```
$\sigma_{A4} = \langle \cdots, \cdots, \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ == null \&\& } \nu_4 \texttt{ == null}\rangle$

$\sigma_{B4} = \langle \cdots, \cdots, \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ != null \&\& } \nu_4 \texttt{ == null}\rangle$

However, the path condition ν_1 `!= null &&` ν_4 `!= null &&` ν_4 `== null` is unsatisfiable, thus we can safely prune this execution path and only continue with the first. We proceed to symbolically execute the call to `singleton` (line 19) by consuming the (empty) pre-condition, and producing the post-condition. The result is represented by a fresh symbolic value ν_5:

```
19     n = singleton(value);
```
$\sigma_{A5} = \langle\{\langle \mathsf{value}, \nu_1, \nu_3\rangle, \langle \mathsf{next}, \nu_1, \nu_4\rangle, \langle \mathsf{acyclic}, \nu_5\rangle\}, \gamma[\mathsf{n} \mapsto \nu_5], \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ == null \&\& } \nu_5 \texttt{ != null}\rangle$

Symbolic execution of this path then jumps to line 22, but to preserve code order we now demonstrate verification of the `else` branch (line 20). To do this, we use states σ_{A3} and σ_{B3}, and add the negation of the condition to verify the `else` body:

```
20 else
```
$\sigma'_{A4} = \langle \cdots, \cdots, \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ == null \&\& } \nu_4 \texttt{ != null}\rangle$

$\sigma'_{B4} = \langle\{\langle \mathsf{value}, \nu_1, \nu_3\rangle, \langle \mathsf{next}, \nu_1, \nu_4\rangle, \langle \mathsf{acyclic}, \nu_4, \}\rangle, \gamma, \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ != null}\rangle$

Here again this results in an unsatisfiable path condition ν_1 `!= null &&` ν_4 `== null &&` ν_4 `!= null`, so we prune that path. We continue with the other path and execute the recursive call to append (line 21), which consumes the pre-condition (removing $\langle$`acyclic`, $\nu_4\rangle$) and produces the post-condition, using the fresh value ν_6 to represent the result (adding $\langle$`acyclic`, $\nu_6\rangle$):

```
21     n = append(l.next, value);
```
$\sigma'_{B5} = \langle\{\langle \mathsf{value}, \nu_1, \nu_3\rangle, \langle \mathsf{next}, \nu_1, \nu_4\rangle, \langle \mathsf{acyclic}, \nu_6\rangle\}, \gamma[\mathsf{n} \mapsto \nu_6], \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ != null \&\& } \nu_6 \texttt{ != null}\rangle$

Now we have completed verifying both branches of the `if` statement. Note that we do not actually join execution at this point; instead, we jump to line 22 immediately after executing the program up to σ_{A5} and σ'_{B5} along both paths. We follow both of these paths for the rest of verification. The

field assignment on line 22 consumes `acc(l.next)` and produces a new corresponding heap chunk with n's value:

```
22 l.next = n;
```

$\sigma_{A6} = \langle\{\langle \texttt{value}, \nu_1, \nu_3\rangle, \langle \texttt{next}, \nu_1, \nu_5\rangle, \langle \texttt{acyclic}, \nu_5\rangle\}, \gamma[\texttt{n} \mapsto \nu_5], \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ == null \&\& } \nu_5 \texttt{ != null}\rangle$

$\sigma'_{B6} = \langle\{\langle \texttt{value}, \nu_1, \nu_3\rangle, \langle \texttt{next}, \nu_1, \nu_6\rangle, \langle \texttt{acyclic}, \nu_6\rangle\}, \gamma[\texttt{n} \mapsto \nu_6], \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ != null \&\& } \nu_6 \texttt{ != null}\rangle$

Folding `acyclic` results in twice the number of execution paths since it consumes `acylic(l)`'s body, which includes an logical conditional. However, again, information from the path conditions in σ_{A6} and σ'_{B6} allow us to prune some of these paths. We elide these pruned paths and only show the taken ones. After consuming `acyclic(l)`'s body, execution produces `acyclic(l)` into the state:

```
23 fold acyclic(l);
```

$\sigma_{A7} = \langle\{\langle \texttt{acyclic}, \nu_1\rangle\}, \gamma[\texttt{n} \mapsto \nu_5], \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ == null \&\& } \nu_5 \texttt{ != null}\rangle$

$\sigma'_{B7} = \langle\{\langle \texttt{acyclic}, \nu_1\rangle\}, \gamma[\texttt{n} \mapsto \nu_6], \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ != null \&\& } \nu_6 \texttt{ != null}\rangle$

```
24 result = l;
```

$\sigma_{A8} = \langle\{\langle \texttt{acyclic}, \nu_1\rangle\}, \gamma[\texttt{n} \mapsto \nu_5, \texttt{result} \mapsto \nu_1], \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ == null \&\& } \nu_5 \texttt{ != null}\rangle$

$\sigma'_{B8} = \langle\{\langle \texttt{acyclic}, \nu_1\rangle\}, \gamma[\texttt{n} \mapsto \nu_6, \texttt{result} \mapsto \nu_1], \nu_1 \texttt{ != null \&\& } \nu_4 \texttt{ != null \&\& } \nu_6 \texttt{ != null}\rangle$

Finally, in both σ_{A8} and σ'_{B8}, we can consume the post-condition `acyclic(result) * result != NULL`. Therefore, we have verified all possible symbolic execution paths of append's body, and thus verified append.

3 GVL_{C0}

SVL_{C0} reflects the core components of Viper—`eval`, `consume`, `produce`, and `exec`. We now formally define GVL_{C0}, an extension of SVL_{C0} which supports gradual specifications. We then define static verification for GVL_{C0} that allows optimistic assumptions to satisfy proof goals and generates checks to be verified at run time to cover these assumptions as in DiVincenzo et al. [2022].

Note, the syntax of GVL_{C0} differs slightly from that of GVC0 (the frontend for Gradual C0), particularly with its omission of C-style pointers. However, due to the restrictions of C_0, all usages of pointers in C_0 can be translated to use object references. This and other translations are done by Gradual C0 during its conversion to an intermediate language Gradual Viper, which is used in the backend verifier. In order to simplify our model, GVL_{C0} is very similar to the language of GVC0, but incorporates elements of the Gradual Viper language when this simplifies the definition of our verification algorithm.

3.1 Gradual Formulas

We first extend the syntax of our language to include *imprecise* formulas—formulas of the form $? * \phi$. An imprecise formula may represent any logically consistent strengthening of the precise portion ϕ [Wise et al. 2020]. For example, the imprecise formula $? * x > 0$ consistently implies $x == 2$, but does not consistently imply $x == 0$. Then, a *gradual formula* $\tilde{\phi}$ may be precise or imprecise, and *gradual programs* are programs that contain gradual formulas. The abstract syntax of GVL_{C0} extends SVL_{C0}'s syntax with gradual formulas:

$$\mathcal{P} ::= p(\overline{T\ x}) = \tilde{\phi} \qquad s ::= \cdots \mid \texttt{while}\ e\ \texttt{invariant}\ \tilde{\phi}\ \texttt{do}\ s$$

$$\Phi ::= \texttt{requires}\ \tilde{\phi}\ \texttt{ensures}\ \tilde{\phi} \qquad \tilde{\phi} ::= \phi \mid ? * \phi$$

Note, imprecise formulas are always considered self-framed, because they can always be strengthened to be self-framing. Therefore we require all method pre- and postconditions, loop invariants, and predicate bodies to be *specifications*—formulas which are either imprecise or self-framed.

Also, note that IDF is particularly well-suited for gradual specifications, in comparison to separation logic [Reynolds 2002], since IDF allows separately specifying access permission and heap values. This allows specification of heap values while leaving more complex accessibility assertions unspecified, as in the formula `? * x.f != null`.

3.2 Representation

In this section we extend the data structures from §2.2 to support *imprecise states*—states in which it is permissible to make optimistic assumptions—and define our representation of run-time checks.

- A symbolic state σ is now a quintuple $\langle \iota, \mathsf{H}, \mathcal{H}, \gamma, g \rangle$ where ι is an *imprecise flag*, H is a *precise heap*, $\mathcal{H}$ is an *optimistic heap*, γ is the *symbolic store*, and g is the *path condition*. As before, we use the notation $\iota(\sigma)$, $\mathsf{H}(\sigma)$, etc. to reference specific components of a symbolic state. γ and g are defined in §2.2 but we redefine the other components.
- An *imprecise flag* $\iota \in \{\top, \bot\}$ is a flag indicating whether a symbolic state is imprecise ($\top$) or precise ($\bot$). $\iota(\sigma)$ denotes that $\iota(\sigma) = \top$ (and thus σ is an imprecise state), while $\neg\iota(\sigma)$ denotes that an $\iota(\sigma) = \bot$ (and thus σ is precise). Imprecise states are produced by consuming or producing an imprecise specification. Once imprecise, a state always remains imprecise.
- A *precise heap* H is a symbolic heap as described in section 2.2. Thus it is a finite set of heap chunks where all heap chunks represent distinct locations in the heap at run time.
- An *optimistic heap* $\mathcal{H}$ is a finite set of field chunks. Field chunks contained in the optimistic heap may represent the same location in the heap at run time, i.e. the optimistic heap does not preserve the separation invariant like the precise heap. The optimistic heap of a well-formed symbolic state must be empty unless it is an imprecise state.

3.3 Run-Time Checks

A *run-time check* $r \in \textsc{SCheck}$ denotes an assertion that validates assumptions made during static verification of imprecise programs. It is a symbolic expression, symbolic permission, pair of symbolic permission sets, or $\bot$:

$$r \quad ::= \quad t \quad | \quad \theta \quad | \quad \mathrm{sep}(\Theta_1, \Theta_2) \quad | \quad \bot$$

A set of run-time checks is denoted $\mathcal{R} \in \mathcal{P}(\textsc{SCheck})$. In a run-time check, a symbolic expression t asserts that the value represented by t at run time is `true`, a symbolic permission θ asserts ownership of a field or a predicate instance, and a pair $\mathrm{sep}(\Theta_1, \Theta_2)$ asserts that the sets of permissions represented by Θ_1 and Θ_2 are disjoint. $\bot$ represents a static verification failure. We represent static verification failure as an unsatisfiable run-time check, instead of failing verification entirely, to accommodate imprecision.

Note that our run-time checks contain symbolic values. This is unlike Gradual C0 [DiVincenzo et al. 2022], where checks produced have their symbolic values replaced by corresponding program variables. This replacement is needed to support the implementation of run-time checks and adds a significant amount of complexity to their algorithms. Fortunately, as we will see later, we can abstract away this connection of symbolic values to program variables (aka. concrete values) using *valuations*; and so we can produce abstracted checks here, avoiding additional complexity in our formalism. Additionally, at each branch point DiVincenzo et al. [2022] check whether *all* possible branches fail and, if so, halt static verification. We do not specify this behavior; however, this is possible by checking for $\bot \in \mathcal{R}$ at each step of symbolic execution.

3.4 Evaluating Expressions

We now extend our previous judgement for symbolic evaluation from §2.3 to allow optimistic symbolic evaluation of expressions. We specify a set $\mathcal{R}$ of run-time checks necessary for a given evaluation, thus our judgement is now $\sigma \vdash e \Downarrow t \dashv \sigma', \mathcal{R}$.

$$\textsc{SEvalFieldOptimistic}\quad \frac{\sigma \vdash e \Downarrow t_e \dashv \sigma', \mathcal{R} \qquad \nexists \langle f, t'_e, t\rangle \in \mathsf{H}(\sigma') : g(\sigma') \implies t'_e == t_e \qquad \langle f, t'_e, t\rangle \in \mathcal{H}(\sigma') \qquad g(\sigma') \implies t'_e == t_e}{\sigma \vdash e.f \Downarrow t \dashv \sigma', \mathcal{R}}$$

$$\textsc{SEvalFieldImprecise}\quad \frac{\iota(\sigma) \qquad \sigma \vdash e \Downarrow t_e \dashv \sigma', \mathcal{R} \qquad \nexists \langle f, t'_e, t\rangle \in \mathsf{H}(\sigma') \cup \mathcal{H}(\sigma') : g(\sigma') \implies t'_e == t_e \qquad t = \text{fresh} \qquad \mathcal{H}' = \mathcal{H}(\sigma) \cup \{\langle f, t_e, t\rangle\}}{\sigma \vdash e.f \Downarrow t \dashv \sigma'[\mathcal{H} = \mathcal{H}'], \mathcal{R} \cup \{\langle t_e, f\rangle\}}$$

$$\textsc{SEvalFieldFailure}\quad \frac{\neg\iota(\sigma) \qquad \sigma \vdash e \Downarrow t_e \dashv \sigma', \mathcal{R} \qquad \nexists \langle f, t'_e, t\rangle \in \mathsf{H}(\sigma') \cup \mathcal{H}(\sigma') : g(\sigma') \implies t'_e == t_e}{\sigma \vdash e.f \Downarrow \text{fresh} \dashv \sigma', \{\bot\}}$$

Fig. 7. Selected rules for evaluation during gradual verification

Field chunks may be referenced in the optimistic heap by SEvalFieldOptimistic in Figure 7. These field chunks have already been validated, thus we do not need additional run-time checks. A field may also be optimistically evaluated by SEvalFieldImprecise, even if it does not exist in H or $\mathcal{H}$. This adds a new field chunk with a fresh value to $\mathcal{H}$. This requires a run-time check which asserts permission to access the field. Finally, SEvalFieldFailure applies in a precise state when a field is referenced but no matching heap chunk exists. This results in a failure of static verification, represented by $\bot$, for that execution branch.

We also modify the existing set of rules described in §2.3 to collect run-time checks from recursive evaluations. Likewise, we modify the deterministic evaluation judgement to add similar rules as those described above, allowing it to also generate run-time checks, thus its form is $\sigma \vdash e \downarrow t \dashv \mathcal{R}$.

3.5 Consuming Formulas

We extend our previous judgment for consuming formulas from §2.4 to handle imprecise formulas and imprecise states. As in §3.4, we add a parameter $\mathcal{R}$ to the consume judgments. Additionally, we collect all permissions for the given formula into a set of symbolic permissions Θ so that separation checks may be added where necessary. Thus the new judgments are of the form $\sigma \vdash \tilde{\phi} \rhd \sigma', \mathcal{R}$ and $\sigma, \sigma_E \vdash \tilde{\phi} \rhd \sigma', \mathcal{R}, \Theta$; these two forms are related by SConsume and correspond to the forms described in §2.4. We list selected rules in Figure 8.

Consuming an imprecise formula empties the precise and optimistic heaps (SConsumeImprecise). This is because the imprecision may represent access to arbitrary fields. For example, a method with an imprecise precondition could modify any field that the callee owns, thus we cannot make any assumptions about field permissions or values after the method returns. Consuming an imprecise formula results in an imprecise state, thus removed field chunks can be referenced optimistically, with the possible addition of a run-time check.

We must also consider the case of consuming an imprecise formula in a precise state. Since optimistic assumptions are not permitted in a precise state, we cannot assume any of the assertions contained in the imprecise formula. However, the imprecise formula may reference fields without a corresponding accessibility predicate. Thus, when consuming an imprecise formula, we use an imprecise state as the symbolic state for evaluation, but use the original (possibly precise) state for assertions.

In an imprecise state we may optimistically consume an expression, even if it is not implied by the current path condition. We then add the value as a run-time check to be asserted at run-time.

Consumption of accessibility predicates must be modified to handle imprecise states, where field chunks in $\mathcal{H}$ may overlap with field chunks in H. We must remove all fields that *may* represent the same heap reference when removing a field chunk from H. To do this, we use the helper functions rem_{fp} and rem_{f}. rem_{fp} is used when removing heap chunks from the precise heap. For precise states, rem_{fp} removes the field chunk that coincides exactly with the heap location being consumed

$$\textsc{SConsume}\ \frac{\sigma,\ \sigma_E \vdash \tilde{\phi} \rhd \sigma',\ \mathcal{R},\ \Theta}{\sigma \vdash \tilde{\phi} \rhd \sigma',\ \mathcal{R}} \qquad \textsc{SConsumeImprecise}\ \frac{\sigma,\ \sigma_E[\iota = \top] \vdash \phi \rhd \sigma',\ \mathcal{R},\ \Theta}{\sigma,\ \sigma_E \vdash ? * \phi \rhd \langle \top, g(\sigma'), \gamma(\sigma'), \emptyset, \emptyset \rangle,\ \mathcal{R},\ \Theta}$$

$$\textsc{SConsumeValueImprecise}\ \frac{\iota(\sigma) \qquad \sigma_E \vdash e \downarrow t \dashv \mathcal{R} \qquad g(\sigma) \Rrightarrow t}{\sigma,\ \sigma_E \vdash e \rhd \sigma[g = g(\sigma)\ \&\&\ t],\ \mathcal{R} \cup \{t\},\ \emptyset} \qquad \textsc{SConsumeValueFailure}\ \frac{\neg\iota(\sigma) \qquad \sigma_E \vdash e \downarrow t \dashv \mathcal{R} \qquad g(\sigma) \Rrightarrow t}{\sigma,\ \sigma_E \vdash e \rhd \sigma,\ \{\bot\},\ \emptyset}$$

$$\textsc{SConsumeAcc}\ \frac{\begin{array}{c}\sigma \vdash e \downarrow t_e \dashv \mathcal{R} \qquad g(\sigma) \implies t'_e == t_e \\ \langle f, t'_e, t\rangle \in \mathsf{H}(\sigma) \\ \mathsf{H}' = \mathrm{rem}_{\mathrm{fp}}(\mathsf{H}(\sigma), \sigma, t_e, f) \\ \mathcal{H}' = \mathrm{rem}_{\mathrm{f}}(\mathcal{H}(\sigma), \sigma, t_e, f) \\ \sigma' = \sigma[\mathsf{H} = \mathsf{H}', \mathcal{H} = \mathcal{H}']\end{array}}{\sigma,\ \sigma_E \vdash \mathsf{acc}(e.f) \rhd \sigma',\ \mathcal{R},\ \{\langle t_e, f\rangle\}}$$

$$\textsc{SConsumeAccOptimistic}\ \frac{\begin{array}{c}\sigma \vdash e \downarrow t_e \dashv \mathcal{R} \qquad g(\sigma) \implies t'_e == t_e \\ \nexists\, \langle f, t'_e, t\rangle \in \mathsf{H}(\sigma) : g(\sigma) \implies t'_e == t_e \\ \langle f, t'_e, t\rangle \in \mathcal{H}(\sigma) \\ \mathsf{H}' = \mathrm{rem}_{\mathrm{f}}(\mathsf{H}(\sigma), \sigma, t_e, f) \\ \mathcal{H}' = \mathrm{rem}_{\mathrm{f}}(\mathcal{H}(\sigma), \sigma, t_e, f) \\ \sigma' = \sigma[\mathsf{H} = \mathsf{H}', \mathcal{H} = \mathcal{H}']\end{array}}{\sigma,\ \sigma_E \vdash \mathsf{acc}(e.f) \rhd \sigma',\ \mathcal{R},\ \{\langle t_e, f\rangle\}}$$

$$\textsc{SConsumeAccImprecise}\ \frac{\begin{array}{c}\iota(\sigma) \qquad \sigma \vdash e \downarrow t_e \dashv \mathcal{R} \\ \nexists\, \langle f, t'_e, t\rangle \in \mathsf{H}(\sigma) \cup \mathcal{H}(\sigma) : g(\sigma) \implies t'_e == t_e \\ \sigma' = \sigma[\mathsf{H} = \mathrm{rem}_{\mathrm{f}}(\mathsf{H}(\sigma), \sigma, t_e, f), \mathcal{H} = \mathrm{rem}_{\mathrm{f}}(\mathcal{H}(\sigma), \sigma, t_e, f)]\end{array}}{\sigma,\ \sigma_E \vdash \mathsf{acc}(e.f) \rhd \sigma',\ \mathcal{R} \cup \{\langle t_e, f\rangle\},\ \{\langle t_e, f\rangle\}}$$

$$\textsc{SConsumeAccFailure}\ \frac{\begin{array}{c}\neg\iota(\sigma) \qquad \sigma \vdash e \downarrow t_e \dashv \mathcal{R} \\ \nexists\, \langle f, t'_e, t\rangle \in \mathsf{H}(\sigma) \cup \mathcal{H}(\sigma) : g(\sigma) \implies t'_e == t_e\end{array}}{\sigma,\ \sigma_E \vdash \mathsf{acc}(e.f) \rhd \sigma,\ \{\bot\},\ \{\langle t_e, f\rangle\}}$$

Fig. 8. Selected rules for consuming gradual formulas

(thus computing the same result as the rules in §2.4). For imprecise states, it also removes all field chunks that could possibly coincide with the specified heap location. $\mathrm{rem}_{\mathrm{f}}$ is used when removing chunks from the optimistic heap and behaves similarly, but also removes all predicate chunks, since predicates occurring in the precise heap could overlap with heap chunks in the imprecise heap. Some optimizations could be made – for instance, if a predicate's unfolding will never reference a field f, we could preserve an instance of this predicate when consuming $\mathsf{acc}(e.f)$. However, we leave such optimizations to future work.

We can also optimistically assume an accessibility predicate in an imprecise state, even if a matching field chunk does not exist in H or $\mathcal{H}$. Since this assumes ownership of the field, we add the corresponding symbolic permission to $\mathcal{R}$. Finally, like accessibility predicates, we allow optimistic consumption of predicate instances. In this case the symbolic permission representing the predicate instance is added as a run-time check.

When consuming any accessibility predicate or predicate instance, the symbolic permission is always added to a set Θ. This allows specification of checks for separation. When consuming $\mathsf{acc}(x.f) * \mathsf{acc}(y.f)$, if $\mathsf{acc}(x.f)$ is optimistically assumed while $\mathsf{acc}(y.f)$ is statically verified, the run-time check for $\mathsf{acc}(x.f)$ does not imply that its permission is disjoint from that of $y.f$. Therefore additional checks for separation are added when consuming a separating conjunction such as $\phi_1 * \phi_2$. If no run-time check for permissions exists, all permissions must have been consumed from H or $\mathcal{H}$ and thus separation may be assumed. However, if a symbolic permission is contained in $\mathcal{R}$ we can no longer assume separation. Thus we add a run-time check $\mathrm{sep}(\Theta_1, \Theta_2)$ where Θ_1 is the set of symbolic permissions collected while consuming ϕ_1 and likewise for Θ_2 and ϕ_2.

3.6 Producing Formulas

Since a formula is only produced when we can assume its validity, producing a gradual formula does not require any optimistic assumptions, thus we do not need to calculate any run-time checks. When producing an imprecise formula, we produce the precise portion and set $\iota = \top$. All other rules from §2.5 are left unchanged.

$$\textsc{SGuardAssign}\ \frac{\sigma \vdash e \Downarrow \_ \dashv \sigma',\ \mathcal{R}}{\langle \sigma,\ x = e;\ s,\ \tilde{\phi} \rangle \rightharpoonup \sigma' \dashv \mathcal{R},\ \emptyset}$$

$$\textsc{SGuardAssignField}\ \frac{\sigma \vdash e \Downarrow \_ \dashv \sigma',\ \mathcal{R} \qquad \sigma' \vdash \mathrm{acc}(x.f) \rhd \sigma'',\ \mathcal{R}'}{\langle \sigma,\ x.f = e;\ s,\ \tilde{\phi} \rangle \rightharpoonup \sigma'' \dashv \mathcal{R} \cup \mathcal{R}',\ \emptyset}$$

$$\textsc{SGuardCall}\ \frac{\sigma \vdash e \Downarrow t \dashv \sigma',\ \mathcal{R} \qquad \overline{x} = \mathrm{params}(m) \qquad \sigma'[\gamma = [\overline{x \mapsto t}]] \vdash \mathrm{pre}(m) \rhd \sigma'',\ \mathcal{R}'}{\langle \sigma,\ y = m(\overline{e});\ s,\ \tilde{\phi} \rangle \rightharpoonup \sigma'' \dashv \mathcal{R} \cup \mathcal{R}', \mathrm{rem}(\sigma'', \mathrm{pre}(m))}$$

$$\mathrm{rem}(\sigma, \tilde{\phi}) := \begin{cases} \emptyset & \text{if } \tilde{\phi} \text{ is completely precise} \\ \{\langle p, \overline{t} \rangle : \langle p, \overline{t} \rangle \in \mathsf{H}(\sigma)\} \cup & \\ \quad \{\langle t, f \rangle : \langle f, t, t' \rangle \in \mathsf{H}(\sigma) \cup \mathcal{H}(\sigma)\} & \text{otherwise} \end{cases}$$

Fig. 9. Selected guard rules

```
1 List append(List l, int value)
2   requires ? * true
3   ensures ? * acyclic(result)
4 {
5   if (l.next == NULL)
6     n = singleton(value);
7   else
8     n = append(l.next, value);
9   l.next = n;
10  result = l;
11 }
```

Fig. 10. $\mathrm{GVL}_{\mathrm{C0}}$ code for appending to an acyclic linked list

3.7 Executing Statements

All rules from §2.6 are left unchanged. While it may seem natural to calculate run-time checks while determining execution transitions (as in the `exec` algorithm of DiVincenzo et al. [2022]), we found that this unnecessarily complicates statements of soundness since symbolic execution steps are not equivalent to dynamic execution steps. For example, a method call occurs in one step during symbolic execution but may never complete during dynamic execution, therefore it may be impossible to determine which symbolic execution step applies. However, assertion of run-time checks must occur before a dynamic execution step may proceed. Therefore we cleanly delineate between symbolic execution transitions, specified by the judgement $\sigma \vdash s \longrightarrow s' \dashv \sigma'$, and the calculation of run-time checks.

3.8 Guarding Execution

As described above, we must the assert run-time checks *before* the corresponding dynamic execution step occurs. Therefore we define *guard* judgements to calculate run-time checks which ensure that execution can safely proceed. A guard for a method call calculates the checks necessary to ensure that the method's pre-condition is satisfied, while a guard for a field assignment calculates the checks necessary to ensure permission to access the assignee and evaluate the value to be stored.

A guard judgement $\Sigma \rightharpoonup \sigma' \dashv \mathcal{R},\ \Theta$ denotes that, at the execution state represented by Σ, when the execution path matches the path condition in σ', the run-time checks $\mathcal{R}$ must be checked. Selected guard rules are defined in Figure 9.

In a guard judgement, Θ determines the exclusion frame—a set of permissions which must not escape the executing method's context. Its necessity and behavior is explained in §8.

3.9 Example

We now illustrate verification of the gradually-specified method in Figure 10. We assume the definition of `List` and `acyclic` from Figure 6. The gradual specification of `append` ensures that all returned lists are acyclic.

Symbolic states are tuples of the form $\langle \iota, \mathsf{H}, \mathcal{H}, \gamma, g \rangle$. As in §2.8, we begin verification of append by assigning fresh symbolic values to all parameters and producing the pre-condition ? ∗ true, which results in an imprecise state:

$\sigma_1 = \langle \top, \emptyset, \emptyset, \gamma, \mathsf{true} \rangle \quad \gamma = [\mathsf{l} \mapsto \nu_1, \mathsf{v} \mapsto \nu_2]$

At each statement we compute the guard to find the necessary run-time checks. The guard for the `if` statement at line 5 evaluates `l.next == NULL`. A heap chunk for `l.next` is optimistically added to $\mathcal{H}$ with a fresh value ν_3. This also results in a run-time check for the symbolic permission $\langle \nu_1, \mathsf{next} \rangle$.

The next state σ_{A2} is computed by symbolic execution. This again evaluates `l.next == NULL` in the state σ_1, which again requires the addition of an optimistic heap chunk. Since ν_3 was not previously used in σ_1 it can be used again as a fresh value for symbolic execution.

$\mathcal{R} = \{\langle \nu_1, \mathsf{next} \rangle\}$

```
5 if (l.next == NULL)
```

$\sigma_{A2} = \langle \top, \emptyset, \{\langle \mathsf{next}, \nu_1, \nu_3 \rangle\}, \gamma, \nu_3 == \mathsf{null} \rangle$

The guard for line 6 consumes the pre-condition of `singleton`, which requires no run-time checks. Symbolic execution consumes the same pre-condition and produces the post-condition; here we use the fresh value ν_4 for the returned value:

$\mathcal{R} = \emptyset$

```
6     n = singleton(value);
```

$\sigma_{A3} = \langle \top, \{\langle \mathsf{acyclic}, \nu_4 \rangle\}, \{\langle \mathsf{next}, \nu_1, \nu_3 \rangle\}, \gamma[\mathsf{n} \mapsto \nu_4], \nu_3 == \mathsf{null} \rangle$

As in §2.8, we follow code order, instead of following each execution path individually, and distinguish separate execution paths with color-coding. The guard at line 5 computes the checks for both branches of `if`, thus the guard is not computed at line 7. We can symbolically execute the `else` branch by adding the negation of the path condition we used previously:

```
7 else
```

$\sigma_{B2} = \langle \top, \emptyset, \{\langle \mathsf{next}, \nu_1, \nu_3 \rangle\}, \gamma, \nu_3 \mathrel{!}= \mathsf{null} \rangle$

The guard for line 8 consumes the pre-condition of append, which is ? ∗ true. Since the body of this imprecise formula is only `true`, no run-time checks are necessary. However, since this is an imprecise formula, we clear the precise and optimistic heaps (SConsumeImprecise in Figure 8). Symbolic execution then produces the post-condition; here we use the fresh value ν_5 for the returned value:

$\mathcal{R}_B = \emptyset$

```
8     n = append(l.next, value);
```

$\sigma_{B3} = \langle \top, \{\langle \mathsf{acyclic}, \nu_5 \rangle\}, \emptyset, \gamma[\mathsf{n} \mapsto \nu_5], \nu_3 \mathrel{!}= \mathsf{null} \;\&\&\; \nu_5 \mathrel{!}= \mathsf{null} \rangle$

We resume symbolic execution of both paths at line 9. Executing `l.next = n` in the state σ_{A3} does not require any run-time checks since it contains the heap chunk representing `l.next`. However, executing the same statement in σ_{B3} requires optimistic assumption of the symbolic permission $\langle \mathsf{next}, \nu_1 \rangle$ which requires a run-time check and removes the predicate instance. DiVincenzo et al. [2022] describe the implementation of such conditional run-time checks, but here we represent it with separate symbolic execution paths:

$\mathcal{R}_A = \emptyset$, $\mathcal{R}_B = \{\langle \mathsf{next}, \nu_1 \rangle\}$

```
9 l.next = n;
```

$\sigma_{A4} = \langle \top, \{\langle \mathsf{acyclic}, \nu_4 \rangle\}, \{\langle \mathsf{next}, \nu_1, \nu_4 \rangle\}, \gamma[\mathsf{n} \mapsto \nu_4], \nu_3 == \mathsf{null} \rangle$

$\sigma_{B4} = \langle \top, \{\langle \mathsf{next}, \nu_1, \nu_5 \rangle\}, \emptyset, \gamma[\mathsf{n} \mapsto \nu_5], \nu_3 \mathrel{!}= \mathsf{null} \;\&\&\; \nu_5 \mathrel{!}= \mathsf{null} \rangle$

$\mathcal{R}_A = \emptyset$, $\mathcal{R}_B = \emptyset$

```
10 result = l;
```

$\sigma_{A5} = \langle \top,\ v_3 \texttt{ == null},\ \{\langle \texttt{acyclic},\ v_4\rangle\},\ \{\langle \texttt{next},\ v_1,\ v_4\rangle\},\ \gamma[\texttt{n} \mapsto v_4, \texttt{result} \mapsto v_1]\rangle$

$\sigma_{B4} = \langle \top,\ v_3 \texttt{ != null \&\& } v_5 \texttt{ != null},\ \{\langle \texttt{next},\ v_1,\ v_5\rangle\},\ \emptyset,\ \gamma[\texttt{n} \mapsto v_5, \texttt{result} \mapsto v_1]\rangle$

Finally, the applicable guard at line 11 consumes the post-condition `acyclic(result)`. Since neither state contains a matching predicate instance, in both paths a run-time check is added for the symbolic permission $\langle \texttt{acyclic},\ v_1\rangle$:

$\mathcal{R}_A = \{\langle \texttt{acyclic},\ v_1\rangle\}$, $\mathcal{R}_B = \{\langle \texttt{acyclic},\ v_1\rangle\}$

```
11 }
```

Now we have verified the method and computed all necessary run-time checks.

4 EXECUTING GVL_{C0}

Since soundness of static verification requires specification of program execution, we define execution semantics for GVL_{C0} programs (including SVL_{C0} programs, which are a subset). This includes dynamic semantics for formulas, and execution semantics which dynamically assert the validity of every specification. Therefore, these semantics define valid execution for GVL_{C0} programs.

As explained in §1, execution semantics are based on those of C_0 [Arnold 2010], while the semantics of formulas are based on those of GVL_{RP} [Wise et al. 2020].

4.1 Representation

In this section, we formally define the data structures used during execution of GVL_{C0} programs:

- A *value* $v \in \textsc{Value}$ is an integer, boolean, or object reference.
- An *object reference* $\ell \in \textsc{Ref}$ is an identifier for a particular object. As with symbolic values, we assume that an infinite number of distinct values can be generated by the fresh function. The type of value represented by fresh is disambiguated by its usage.
- An *environment* ρ is a partial function mapping variable names to values, i.e. $\rho : \textsc{Var} \rightharpoonup \textsc{Value}$.
- A *heap* $H : \textsc{Ref} \times \textsc{Field} \to \textsc{Value}$ is a function mapping object reference and field pairs to values. We assume that the heap function is total, i.e. all reference and field pairs have some corresponding value, but heap access is restricted during execution by a set of *access permissions* $\alpha \in \mathcal{P}(\textsc{Ref} \times \textsc{Field})$. This reflects the semantics of IDF [Smans et al. 2012]. A heap location $\langle \ell,\ f\rangle$ is *owned* if it is contained in the currently-applicable set of access permissions.
- A *stack frame* is a triple $\langle \alpha,\ \rho,\ s\rangle$ containing of a set of owned permissions α, a local environment ρ, and a statement s. A *stack* $\mathcal{S}$ is a list of stack frames – either $\langle \alpha,\ \rho,\ s\rangle \cdot \mathcal{S}$ for some other $\mathcal{S}$, or the empty stack, denoted nil. For a non-empty stack $\mathcal{S}$, $\alpha(\mathcal{S})$, $\rho(\mathcal{S})$, and $s(\mathcal{S})$ refer to their respective components of the head element.
- A *dynamic state* Γ may be a symbol init or final, or pair $\langle H,\ \mathcal{S}\rangle$ containing a heap H and a non-empty stack $\mathcal{S}$. $H(\Gamma)$ and $\mathcal{S}(\Gamma)$ reference individual components of Γ when Γ is not a symbol, while $\alpha(\Gamma)$, $\rho(\Gamma)$, and $s(\Gamma)$ reference a component of the head element of $\mathcal{S}(\Gamma)$. $\alpha(\text{init})$ and $H(\text{init})$ are defined to be $\emptyset$.

4.2 Evaluating Expressions

Given a heap H and environment ρ, the evaluation of expression e to a value v is represented by a judgement of the form $\langle H,\ \rho\rangle \vdash e \Downarrow v$. This follows standard evaluation rules—variables are evaluated to the corresponding value in ρ and field references are evaluated to the corresponding value in H. The boolean operators && and || are short-circuiting—when evaluating e_1 && e_2, e_2 is only evaluated when e_1 is not true.

4.3 Asserting Formulas

A judgement of the form $\langle H,\ \alpha,\ \rho\rangle \vDash \tilde{\phi}$ denotes that a gradual formula $\tilde{\phi}$ is satisfied given a heap H, a set of accessible permissions α, and an environment ρ. Selected rules are shown in Figure 11. Boolean expressions are satisfied when they evaluate to true. An accessibility predicate $\texttt{acc}(e.f)$ is satisfied when the field referenced by $e.f$ is in the set of accessible permissions. A predicate instance $p(\overline{e})$ is satisfied when the predicate body is satisfied using an environment mapping each predicate parameter x to the corresponding argument e. A separating conjunction $\phi_1 * \phi_2$ is satisfied when ϕ_1 is satisfied using a permission set α_1 and ϕ_2 is satisfied using a permission set α_2, where α_1 and α_2 are disjoint subsets of α. Finally, an imprecise formula $? * \phi$ is satisfied exactly when ϕ is satisfied.

A judgement of the form $\langle H,\ \alpha,\ \rho\rangle \vdash_{\text{frm}} e$ denotes that the expression e is *framed* by the given set of permission α. This denotes that all heap locations necessary to evaluate e are included in α. Selected rules are shown in Figure 11.

Note that a predicate instance $p(\overline{e})$ is satisfied iff the predicate body is satisfied. Thus dynamic execution of GVL_{C0} uses *equirecursive* semantics for predicates [Summers and Drossopoulou 2013]. We also define *equirecursive framing* of formulas by the judgement $\langle H,\ \alpha,\ \rho\rangle \vdash_{\text{frmE}} \tilde{\phi}$. A formula is equirecursively framed if its *unrolling*, the recursive expansion of referenced predicate bodies, is framed.

A formula $\tilde{\phi}$ is *self-framed* if $\forall H, \alpha, \rho : \langle H,\ \alpha,\ \rho\rangle \vDash \tilde{\phi} \implies \langle H,\ \alpha,\ \rho\rangle \vdash_{\text{frmE}} \tilde{\phi}$. As specified in §3.1, a specification is a formula which is imprecise or self-framed.

4.4 Footprints

The *footprint* of a formula is the set of permissions necessary to assert a formula [Reynolds 2002]. There are two types of footprints for gradual formulas:

The *exact footprint* of a formula is the minimal set of permissions necessary to assert and frame a formula. Given a heap H and environment ρ, $[\![\tilde{\phi}]\!]_{\langle H,\rho\rangle}$ denotes the exact footprint of a formula $\tilde{\phi}$.

The *maximal footprint* (often abbreviated to *footprint*) of a formula contains the exact footprint and all permissions that are consistently implied by the formula. The maximal footprint of a completely precise formula is its exact footprint, but the maximal footprint of an imprecise formula contains all accessible permissions. Given a heap H, set of owned permissions α, and environment ρ, $\lfloor\tilde{\phi}\rfloor_{\langle H,\alpha,\rho\rangle}$ denotes the maximal footprint of a formula $\tilde{\phi}$.

4.5 Executing Statements

We represent the dynamic execution of program statements as small-step execution semantics denoted by the judgement $\langle H,\ \mathcal{S}\rangle,\ \hat{\alpha} \rightarrow \langle H,\ \mathcal{S}\rangle$, where the statement s is executing with the initial state $\langle H,\ \mathcal{S}\rangle$, and then transitions to the next statement s' with the new state $\langle H,\ \mathcal{S}\rangle$. $\hat{\alpha}$ specifies the *exclusion frame*, which is described below. Selected rules are shown in Figure 11. Execution will be stuck (i.e., no further derivation will apply) when an error is encountered. For example, execution is stuck when a formula is not satisfied or if some expression is not framed.

A method call is executed by evaluating all arguments, asserting the pre-condition, and adding a new stack frame containing the footprint of the pre-condition and the method body. After the method body is completely executed, the post-condition is asserted and the `result` value in the callee's environment is passed to the caller's environment.

A loop is executed similarly to a method, but uses the loop invariant instead of a method contract. When the loop condition is true, an iteration is executed by asserting the invariant and adding a new stack frame for the loop body. When the body is complete, we return to the original loop statement, allowing further iterations as long as the condition remains true. When the condition is

$$\textsc{AssertImprecise}\ \frac{\langle H, \alpha, \rho\rangle \vDash \phi \qquad \langle H, \alpha, \rho\rangle \vdash_{\text{frmE}} \phi}{\langle H, \alpha, \rho\rangle \vDash ? * \phi}$$

$$\textsc{AssertAcc}\ \frac{\langle H, \rho\rangle \vdash e \Downarrow \ell \qquad \langle \ell, f\rangle \in \alpha}{\langle H, \alpha, \rho\rangle \vDash \mathtt{acc}(e.f)}$$

$$\textsc{AssertPredicate}\ \frac{\overline{x} = \text{predicate\_params}(p) \qquad \overline{\langle H, \rho\rangle \vdash e \Downarrow v} \qquad \langle H, \alpha, [\overline{x \mapsto v}]\rangle \vDash \text{predicate}(p)}{\langle H, \alpha, \rho\rangle \vDash p(\overline{e})}$$

$$\textsc{AssertConjunction}\ \frac{\langle H, \alpha_1, \rho\rangle \vDash \phi_1 \qquad \langle H, \alpha_2, \rho\rangle \vDash \phi_2 \qquad \alpha_1 \cap \alpha_2 = \emptyset \qquad \alpha_1 \cup \alpha_2 \subseteq \alpha}{\langle H, \alpha, \rho\rangle \vDash \phi_1 * \phi_2}$$

$$\textsc{ExecAssignField}\ \frac{\langle H, \rho\rangle \vdash x \Downarrow \ell \qquad \langle H, \rho\rangle \vdash e \Downarrow v \qquad \langle H, \alpha, \rho\rangle \vDash \mathtt{acc}(x.f) \qquad \langle H, \alpha, \rho\rangle \vdash_{\text{frm}} e \qquad H' = H[\langle \ell, f\rangle \mapsto v]}{\langle H, \langle \alpha, \rho, x.f = e;\ s\rangle \cdot \mathcal{S}\rangle, \hat{\alpha} \to \langle H', \langle \alpha, \rho, s\rangle \cdot \mathcal{S}\rangle}$$

$$\textsc{ExecCallEnter}\ \frac{\overline{x} = \text{params}(m) \qquad \overline{\langle H, \rho\rangle \vdash e \Downarrow v} \qquad \overline{\langle H, \alpha, \rho\rangle \vdash_{\text{frm}} e} \qquad \rho' = [\overline{x \mapsto v}] \qquad \langle H, \alpha \setminus \hat{\alpha}, \rho'\rangle \vDash \text{pre}(m) \qquad \alpha' = \lfloor \text{pre}(m) \rfloor_{\langle H, \alpha\setminus\hat{\alpha}, \rho\rangle}}{\langle H, \langle \alpha, \rho, y = m(\overline{e});\ s\rangle \cdot \mathcal{S}\rangle, \hat{\alpha} \to \langle H, \langle \alpha', \rho', \text{body}(m);\ \mathtt{skip}\rangle \cdot \langle \alpha \setminus \alpha', \rho, y = m(\overline{e});\ s\rangle \cdot \mathcal{S}\rangle}$$

$$\textsc{ExecCallExit}\ \frac{\langle H, \alpha', \rho'\rangle \vDash \text{post}(m) \qquad \rho'' = \rho[y \mapsto \rho'(\mathtt{result})] \qquad \alpha'' = \alpha \cup \lfloor \text{post}(m) \rfloor_{\langle H, \alpha', \rho'\rangle}}{\langle H, \langle \alpha', \rho', \mathtt{skip}\rangle \cdot \langle \alpha, \rho, y = m(\overline{e});\ s\rangle \cdot \mathcal{S}\rangle, \hat{\alpha} \to \langle H, \langle \alpha, \rho, s\rangle \cdot \mathcal{S}\rangle}$$

Fig. 11. Selected formal rules for dynamic semantics of GVL_{C0}.

false, the invariant is still asserted but execution skips over the statement. These rules are specified in the supplement [Zimmerman et al. 2024].

$\hat{\alpha}$ specifies the *exclusion frame* – a set of permissions which may not be passed to the callee or loop body. It is used only for executing method calls and loops. We later explain why this is necessary for soundness in §8.

`fold` and `unfold` statements are ignored at run-time. Explicit folding and unfolding of predicate instances is not required because the run-time uses equirecursive semantics for predicates.

The entire set of possible execution steps for a program Π is determined by judgements of the form $\Pi \vdash \Gamma,\ \hat{\alpha} \to \Gamma'$, which denote that execution transitions from Γ to Γ', using the exclusion frame $\hat{\alpha}$. From the init state, execution may only step to the entry statement, and then execution follows the rules described above.

5 CORRESPONDENCE

Before formalizing soundness, we must specify the correspondence between verification and dynamic states. We include invariants which depend on concrete values, such as separation, in this correspondence relation. Finally, we specify the behavior of run-time checks in a dynamic state.

5.1 State Correspondence

A dynamic environment ρ *models* a symbolic store γ via a valuation V when $\rho \models_V \gamma$. This denotes that $\forall x \mapsto t \in \gamma : x \mapsto V(t) \in \rho$.

A heap H and set of permissions α model a precise heap H when $\langle H, \alpha\rangle \models_V \mathsf{H}$. This denotes that for all field chunks $\langle f, t, t'\rangle \in \mathsf{H}$, $H(V(t), f) = V(t')$ and $\langle V(t), f\rangle \in \alpha$. Also, for all predicate chunks $\langle p, \overline{t}\rangle$, the corresponding predicate body is true using given arguments $\overline{V(t)}$. Additionally, the footprint represented by each heap chunk must be disjoint.

The footprint of a heap chunk, given valuation V and heap H, is denoted $V(\!|h|\!)_H$. The footprint of a field chunk $\langle f, t, t'\rangle$ is $\{\langle V(t), f\rangle\}$. The footprint of a predicate chunk $\langle p, \overline{t}\rangle$ is the exact footprint of the predicate when applied to the arguments $\overline{V(t)}$.

$$\frac{V(t) = \texttt{true}}{\langle H, \alpha\rangle \vdash_V t} \qquad \frac{\langle V(t), f\rangle \in \alpha}{\langle H, \alpha\rangle \vdash_V \langle t, f\rangle} \qquad \frac{V(\!|\Theta_1|\!)_H \cap V(\!|\Theta_2|\!)_H = \emptyset}{\langle H, \alpha\rangle \vdash_V \text{sep}(\Theta_1, \Theta_2)} \qquad \frac{\begin{array}{c}\langle H, \alpha, [\overline{x \mapsto V(t)}]\rangle \vDash \text{predicate}(p) \\ \overline{x} = \text{predicate\_params}(p)\end{array}}{\langle H, \alpha\rangle \vdash_V \langle p, \overline{t}\rangle}$$

Fig. 12. Rules for run-time check assertions

H and α model an optimistic heap $\mathcal{H}$ when $\langle H, \alpha\rangle \models\!\!\!\models_V \mathcal{H}$. This has the same requirements as that for H, except that heap chunks are allowed to overlap.

H, α, and α model a symbolic state when $\langle H, \alpha, \rho\rangle \models\!\!\!\models_V \sigma$. This denotes that H and α model both $\mathsf{H}(\sigma)$ and $\mathcal{H}(\sigma)$, ρ models γ, and the path condition is true—$V(g(\sigma)) = \texttt{true}$.

We also refer to these relations as correspondence—$\langle H, \alpha, \rho\rangle \models\!\!\!\models_V \sigma$ denotes that the symbolic state σ *corresponds* to H, α, and ρ.

Finally, a verification state Σ corresponds to a dynamic state Γ with valuation V if $\Sigma = \Gamma$ (i.e., Σ and Γ are the same symbol), or $\langle H(H), \alpha(\Gamma), \rho(\Gamma)\rangle \models\!\!\!\models_V \sigma(\Sigma)$ and $s(\Gamma) = s(\Sigma)$. In other words, the heap and head stack frame of Γ model the symbolic state of Σ, and the statement in the head stack frame is syntactically the same statement as that of Σ.

5.2 Run-Time Checks

We also define the semantics of run-time checks using valuations. The judgement $\langle H, \alpha\rangle \vdash_V r$ denotes the assertion of a run-time check r, given a valuation V, a heap H, and a set of owned permissions α. Likewise, $\langle H, \alpha\rangle \vdash_V \mathcal{R}$ denotes the assertion of all run-time checks contained in $\mathcal{R}$. Formal rules are given in Figure 12.

6 SOUNDNESS

We can now state the soundness of our static verifier. We slightly modify a traditional progress/preservation statement of soundness in order to accommodate run-time checks.

6.1 Corresponding Valuations

For most symbolic execution judgements, we define a *corresponding valuation*, inspired by the valuations used in Khoo et al. [2010]. This defines how symbolic values used in the judgement are mapped to concrete values. To calculate the corresponding valuation we require an initial valuation, which defines the valuation for all symbolic values contained by the input symbolic state, and a dynamic heap, which defines the valuation for optimistically-added fields. A corresponding valuation V' must extend the initial valuation V, i.e. $V'(t) = V(t)$ for all $t \in \text{dom}(V)$.

We denote the corresponding valuation for a judgement $\mathcal{J}$, initial valuation V, and heap H by $V[\mathcal{J} \mid H]$. The definition for each judgement type is defined in the supplement [Zimmerman et al. 2024], along with the proofs for that judgement. Each corresponding valuation is defined by induction on the judgement derivation, specifying the corresponding valuation for each derivation rule. The judgment is nondeterministic if only the input state is considered, but knowing the output state resolves this nondeterminism. When the judgement and heap are clear from context, we simply reference the *corresponding valuation extending* V.

6.2 Valid States

A *valid state* is a dynamic state which is completely characterized by verification states. If $\Gamma = \mathsf{init}$ this is trivially true. For a dynamic state $\langle H, \mathcal{S}\rangle$, we require that the head stack frame corresponds

to a reachable verification state. We also require that all other stack frames are *partially validated* by some reachable verification state.

If a stack frame is executing a method call, partial validation is characterized by the stack frame and heap modeling a reachable symbolic state for that program point, with the callee's precondition consumed. For the full definition refer to Zimmerman et al. [2024].

6.3 Progress and Preservation

Our statement of progress is split into two parts. First, theorem 1 states that if Γ is a valid state and Γ satisfies the run-time checks calculated by a guard with a path condition that matches the current dynamic state, then dynamic execution proceeds. Second, theorem 2 states that we can always find the guard necessary to apply theorem 1—a guard whose path condition matches. Thus, theorem 2 represents completeness of symbolic execution with respect to possible dynamic execution paths. Together these theorems show that, in a valid state, the only possible way for execution to be stuck is when the run-time checks cannot be asserted.

Theorem 1 (Progress part 1)**.** For some program Π, let Γ be some dynamic state validated by Σ and V. If $\Sigma \rightharpoonup \sigma \dashv \mathcal{R}, \Theta, V'$ is the corresponding valuation extending V, $V'(g(\sigma)) = \texttt{true}$, and $\langle H, \alpha(\Gamma)\rangle \vdash_{V'} \mathcal{R}$, then $\Pi \vdash \Gamma, V'(\!|\Theta|\!)_{H(\Gamma)} \rightarrow \Gamma'$ for some Γ'.

Theorem 2 (Progress part 2)**.** For some program Π, let Γ be some dynamic state validated by Σ and V. Then $\Sigma \rightharpoonup \sigma \dashv \mathcal{R}, \Theta$ for some σ, $\mathcal{R}$, and Θ such that $V'(g(\sigma)) = \texttt{true}$ where V' is the corresponding valuation extending V.

Finally, our statement of preservation (theorem 3) assumes the antecedent and conclusion of theorem 1—the initial state is valid and satisfies the run-time checks of some matching guard—as well as a dynamic execution step to Γ'. By theorem 2, we know that there is such a guard statement; i.e., we can always find the necessary set of run-time checks. Then preservation states that the resulting dynamic state Γ' is also valid.

Theorem 3 (Preservation)**.** For some program Π, let Γ be some dynamic state validated by Σ and V. If $\Sigma \rightharpoonup \sigma \dashv \mathcal{R}, \Theta, V'$ is the corresponding valuation extending V, $V'(g(\sigma)) = \texttt{true}$, $\langle H, \alpha(\Gamma)\rangle \vdash_{V'} \mathcal{R}$, and $\Pi \vdash \Gamma, V'(\!|\Theta|\!)_{H(\Gamma)} \rightarrow \Gamma'$, then Γ' is a valid state.

Note that our assumptions for preservation require dynamic execution to not only assert the run-time checks represented symbolically by $\mathcal{R}$, but also respect the exclusion frame represented symbolically by Θ. The necessity and implications of this requirement are discussed in §8.

Taken together, these theorems demonstrate that dynamic execution will never be stuck as long as the run-time checks calculated by static verification succeed. Further, it shows that we calculate run-time checks for all possible execution paths. Since the dynamic execution semantics ensures all necessary specifications are satisfied, this implies that the calculated run-time checks are sufficient.

7 CHALLENGES TO FORMALISM OF STATIC VERIFICATION

Our specification of static verification in §2 and §3 is formalised using non-deterministic inference rules. This differs greatly from the specifications of Schwerhoff [2016] and DiVincenzo et al. [2022], which both use a CPS-style definition for algorithms. The latter form is useful when specifying an implementation, but makes it difficult to formulate a syntactic soundness proof. Furthermore, operational semantics allow a higher level of abstraction than pseudo-code definitions. However, we must carefully consider whether our operational semantics represent the system which is implemented.

7.1 Previous Approaches

During development of our soundness proof, we attempted several formulations of soundness. Initially we abstractly defined a *symbolic stack*—a list of symbolic states with the form of a dynamic stack. This approach allowed us to easily state correspondence of the entire dynamic state—each dynamic stack frame models a corresponding symbolic stack frame.

We found it challenging, however, to prove that this correspondence is maintained. When the dynamic stack takes a step, we must verify that there is a corresponding symbolic stack. To address this issue, we defined an execution semantics for symbolic stacks. Unfortunately, this increased the distance between our formalism and implementation, and now we also need to show that all symbolic states are reachable during static verification. Perhaps due to the complexity of this approach, the proof of correspondence remained quite difficult even after defining this execution semantics.

Instead, we defined a valid state primarily by the correspondence of the currently executing dynamic stack frame with some *reachable* symbolic state—a symbolic state which is computed during static verification, with no input from dynamic execution. This resulted in a much simpler definition of *valid state*.

However, in order to completely prove preservation, we also must specify the behavior of intermediate stack frames – frames contained in the dynamic stack below the currently-executing frame. Thus we provide a recursive definition for a valid *partial state*. For intermediate frames containing a method call that is waiting to complete, this requires the frame to model a symbolic state that results from consuming the callee method's pre-condition from a reachable verification state. We use this to prove that the dynamic state after the method returns models the symbolic state after symbolically executing the method call.

7.2 Verification of Loops

Almost all of our symbolic execution rules are finitely non-deterministic. That is, given an input state, there are a finite number of derivations that can apply. This is necessary since all possible states must be computed during static verification.

While this property matches the finite branching of symbolic execution, we make an exception in the case of loops—specifically, the SVerifyLoop rule (Figure 5). It consumes the loop pre-condition, havocs all variables modified by the loop body (i.e., replaces them with fresh values), and produces the loop post-condition. Thus it replaces all symbolic values that could be modified by the loop body with fresh values. The loop is left in place, which means that the rule can be immediately applied again to derive yet another state. However, this is harmless because repeated applications of this rule result in isomorphic symbolic states—states which represent the same state but with different symbolic values. Since the exact symbolic values do not matter, these are equivalent states from the perspective of static verification. Therefore, even though we allow unbounded non-determinism, an implementation such as Gradual C0 can compute all possible states (as determined by our formal model) up to this equivalence. In other words, unbounded non-determinism is an artifact of our formalization that does not affect an implementation.

This exception is motivated by a disconnect between our formal model and the implementation of Gradual C0 [DiVincenzo et al. 2022]. In our formalism, run-time checks are computed as symbolic values and lack a representation in terms of the source. Furthermore, we interpret these run-time checks by means of the valuation function, which we only extend with fresh values as dynamic execution proceeds. Therefore, the references in a run-time check are fixed – for example, the validity of a check does not change when the heap is updated, since the heap reference has already been fully evaluated against the symbolic heap.

```
1 Cell create()
2 requires true ensures ?
3 { result = alloc(Cell); }
4
5 int consume(Cell c)
6 requires acc(c.value) ensures true
7 { ··· }
```

```
8 int main() {
9   x = create();
10  while (true) invariant ? * true {
11    x.value = 1;
12    consume(x);
13    x = create();
14  }
15  result = 0;
16 }
```

Fig. 13. Example illustrating the neccessity of the SVerifyLoop rule.

Consider the example in Figure 13. A new object reference ℓ_1 is allocated by `create` at line 9. Then we consume the loop pre-condition `?*true`, which results in an imprecise state with empty symbolic heaps. Thus we cannot statically assert access to `x.value` in the loop body (line 11). However, we optimistically assume access and produce a run-time check representing `acc(x.value)`. In our formalism, `x` is symbolically evaluated to a symbolic value v_1 and the corresponding valuation contains the mapping $v_1 \mapsto \ell_1$. Thus the symbolic run-time check is $\langle v_1, \texttt{value}\rangle$, which succeeds since the dynamic state owns $\langle \ell_1, \texttt{value}\rangle$. This permission is then lost when `consume` is executed (line 12), but a new reference ℓ_2 is allocated at line 13, and the dynamic environment is updated with $\texttt{x} \mapsto \ell_2$.

During the next iteration of the loop, if we directly applied the same run-time check, this would again require the run-time check $\langle v_1, \texttt{value}\rangle$. However, this would fail since the dynamic state no longer owns $\langle \ell_1, \texttt{value}\rangle$. But the run-time check should reference ℓ_2, since the check is intended to represent `acc(x.value)`, and $\texttt{x} \mapsto \ell_2$ in the dynamic state.

This contrasts with the implementation of run-time checks in Gradual C0, which translates the symbolic checks into source expressions. For the example described, Gradual C0 directly inserts the assertion `acc(x.value)`. The expression `x.value` is then re-evaluated every time this assertion is checked.

SVerifyLoop fixes this mismatch by allowing our formal model to be updated with new symbolic values. With this rule, we can continue execution using a new symbolic state where we havoc `x`, since it is modified by the loop body, and consume the loop invariant `? * true` again. Thus we begin with a symbolic state with empty symbolic heaps and a symbolic store containing $\texttt{x} \mapsto v_2$, where v_2 is a fresh value. We define a new valuation V' where new symbolic values are mapped to the current dynamic state, i.e. $V'(v_2) = \ell_2$ since $\texttt{x} \mapsto \ell_2$ in the dynamic state. The new symbolic state is isomorphic to the state used during the initial symbolic execution, since it also began execution of the body with empty symbolic heaps. We will again optimistically evaluate `x.value`, which produces a new run-time check for the symbolic permission $\langle v_2, \texttt{value}\rangle$, thus we will check access to $\langle \ell_2, \texttt{value}\rangle$, and therefore our run-time checks succeed.

Finally, since this rule introduces *more* symbolic states in our formal model, this means that our soundness theorems are stronger than they would be otherwise; i.e., the soundness result holds for strictly more cases. As our example demonstrates, we want to consider these additional cases since they are already permitted by Gradual C0 due to its source-level run-time checks. Therefore, this additional rule allows us to abstract away the re-evaluation of source-level checks, allowing us to reason with fixed symbolic values.

8 UNSOUNDNESS OF GRADUAL C0

While attempting to prove the soundness of Gradual C0, we discovered that its implementation [DiVincenzo et al. 2022] allows unsound behavior, and have communicated this to the authors.

```
1 struct Cell { int value; }
2 predicate imprecise() = ? * true
3 void set(Cell c, int v)
4   requires imprecise()
5   ensures true
6 {
7   unfold imprecise();
8   c.value = v;
9 }
```

```
10 int test()
11   requires true
12   ensures result == 0
13 {
14   fold imprecise();
15   Cell c = alloc(Cell);
16   c.value = 0;
17   set(c, 1);
18   result = c.value;
19 }
```

Fig. 14. Example exhibiting unsoundness of DiVincenzo et al. [2022].[2]

This unsoundness results from the combination of imprecise specifications, static verification with isorecursive predicates, and run-time checking with equirecursive predicates.

8.1 Example

We show an example which exhibits this behavior in Figure 14. At line 14 `imprecise()` is folded, thus `? * true` is consumed, and the predicate chunk is added to the symbolic heap. At this point $H = \{\langle \texttt{imprecise} \rangle\}$. In lines 15-16 a new `Cell` is allocated and its value is initialized to 0. At this point $H = \{\langle \texttt{imprecise} \rangle, \langle \texttt{value}, t_1, 0 \rangle\}$, where $\texttt{c} \mapsto t_1$.

At line 17, the `set` method is called. Thus the precondition—`imprecise()`—is consumed, resulting in $H = \{\langle \texttt{value}, t_1, 0 \rangle\}$. The postcondition—`true`—is then produced, which does not change the symbolic state. In this symbolic state `c.value` $\mapsto 0$. Then line 18 adds the mapping `result` $\mapsto 0$ to the symbolic store, which allows the postcondition `result == 0` to be consumed successfully. Now `test` is valid and no run-time checks are required in its body. Symbolic execution of the `set` method shows that this method is also valid but requires a check representing `acc(c.value)` at line 16.

Now we consider dynamic execution of the `test` method. We first use no exclusion frame (i.e. using $\emptyset$ for every occurrence of $\hat{\alpha}$ in the rules).

The fold at line 14 is ignored, a new `Cell` is allocated and initialized at lines 15-16, and the `set` method is called at line 17. The formula `imprecise()` is not completely precise, therefore $\lfloor \texttt{imprecise()} \rfloor_{\langle H, \alpha, \rho \rangle} = \alpha$. Thus all of the caller's owned permissions are passed to `set`. The assertion for `imprecise()` succeeds since its equirecursive unrolling is simply `? * true`. Likewise, the assertion for `acc(c.value)` when executing line 8 also succeeds since the required permissions were passed from `test`. After returning from `set`, `c.value` $\mapsto$ 1 in the dynamic state, thus `result` $\mapsto 1$ after executing line 18. However, the postcondition `result == 0` cannot be asserted, therefore execution is stuck.

Since DiVincenzo et al. [2022] does not implement an exclusion frame, execution proceeds as described above, except that only the calculated run-time checks are asserted. Therefore the `test` method returns 1, which contradicts its contract. Wise et al. [2020] follows the dynamic execution behavior described above, but since it checks every assertion at run-time, execution halts and soundness is preserved.

8.2 Diagnosis

As described above, the caller's permissions are passed to `set`, thus the set of permissions owned by `test` is empty during execution of `set`. But we calculated that after consuming pre(`set`) the symbolic heap still contains the field chunk representing `c.value`. Therefore heap chunks which

[2] `void` methods are used for clarity since they can be trivially translated to the formally defined grammar.

are included in the frame of `set` during dynamic execution are not removed by consume during symbolic execution, thus symbolic execution does not accurately represent dynamic execution.

8.3 Possible Solutions

At first this appears to be an error of static verification, and thus we could address this by making static verification more conservative. More specifically, we could require a stronger invariant of the precise heap: the *maximal* footprint represented by predicate chunks cannot overlap. This contrasts with our current definition, where the exact footprint represented by a predicate chunk must be disjoint from that of all other predicate chunks.

For example, we could clear the symbolic heaps when consuming any formula that is not *completely* precise (i.e., the recursive unfolding contains an imprecise formula). When this occurs, we would also need to use an imprecise state, so that the existence of the removed permissions can be optimistically assumed. This would result in empty symbolic heap after line 17 in Figure 14, and a run-time check for the value of `result` would be required before returning from `test`, thus soundness is preserved.

This would allow maximal footprints of predicate chunks to overlap in the symbolic heap, but when consuming a predicate instance, all potentially overlapping predicate chunks would be removed. Thus, after some predicate instance is consumed, its maximal footprint would not overlap with any permission represented by a heap chunk contained in the symbolic heap.

Alternatively, we could achieve soundness by removing any predicate instance that is not completely precise when additional permissions are added to the precise heap. Similar to the previous option, we would also need to use an imprecise state when this occurs. In the example, that would (perhaps unintuitively) *remove* the `imprecise()` predicate when adding permissions for the `alloc` statement. This would ensure that the maximal footprint of heap chunks in the symbolic heap never overlap.

Unfortunately, both of these options reduce the number of assertions that can be statically discharged when verifying gradual programs, thus more run-time checks would be necessary. Furthermore, the run-time checks require checking a predicate instance, which can be quite costly since this traverses the entire unfolding of the predicate.

Furthermore, allowing the predicate instance folded at line 14 to affect permissions allocated afterward, at line 15, seems counter-intuitive. This invalidates the intuitive assumption that the set of permissions represented by a folded predicate instqance will not change while it remains folded. Furthermore this behavior breaks the semantics of `?`, as specified in Wise et al. [2020], since no logically consistent strengthening of the `imprecise` predicate allows it to include permissions allocated after its body is folded.

This indicates that the semantics of dynamic execution should be modified to exclude access permission for `c.value`, which is allocated after `imprecise()` is folded, from being passed to `set`, which is a precise formula that only requires `imprecise()`. Then execution would fail at line 8 in Figure 14. To accomplish this, we have introduced the concept of an *exclusion frame* – a set of permissions which cannot be passed to a callee. This exclusion frame is calculated by symbolic execution, and passed to dynamic execution in much the same way as run-time checks. It is represented by Θ in the guard judgement (§9), which also calculates $\mathcal{R}$, and is translated to dynamic permissions using a valuation.

The guard rules in Figure 9 calculate the exclusion frame by the rem helper function, after consuming the pre-condition of a method. If the pre-condition is completely precise, then $\Theta = \emptyset$, thus execution of an SVL_{C0} program is not affected. Otherwise, Θ contains all permissions currently contained in the symbolic heaps. In Figure 14, since the pre-condition of `set` is not completely precise, $\Theta = \{\langle t_1, \texttt{value}\rangle\}$ when calculating the guard statement at line 17. At run-time this is

translated to $\hat{\alpha} = \{\langle \ell, \texttt{value} \rangle\}$ where $\texttt{c} \mapsto \ell$. Then all permissions *except* $\langle \ell, \texttt{value} \rangle$ are passed to `set`. Thus the run-time check for `acc(c.value)` at line 8 cannot be asserted.

This addresses the intuitive and semantic problems described above. The isorecursive instance of `imprecise` referenced in the pre-condition of `set` *should not* represent access to `c.value` since it was folded before `c` was allocated. Under this interpretation we would expect a failure at line 8, since `set` does not require the necessary permissions. This also matches the semantics of ?, as defined in [Wise et al. 2020], since the predicate instance folded at line 14 cannot consistently imply access to the heap location allocated at line 15.

8.4 Implementation

There are important implementation challenges that must be addressed before this change can be implemented in Gradual C0 [DiVincenzo et al. 2022]. Currently, Gradual C0 constructs sets of permissions at run-time—before calling a method, for example—by recursively unfolding the neccessary specification and collecting all permissions. However, this method cannot be used to create the exclusion frame, since these permissions are not necessarily represented by a specification. But we expect that a translation algorithm can be developed which generates the source code necessary to compute the exclusion frame at run time. This is similar to the existing translation algorithm described by DiVincenzo et al. [2022], which translates symbolic run-time checks into source code that implements the desired assertion.

Also, note that we calculate the exclusion frame using information from symbolic execution of a particular statement. In other words, if method $\texttt{m}$ calls $\texttt{m}'$, we can calculate the exclusion frame necessary for calling $\texttt{m}'$ without considering the exclusion frame used to call $\texttt{m}$. This implies that exclusion frames can be dropped when entering a completely precise method, and then instantiated again when a precise method calls an imprecise method. This is similar to how Gradual C0 does not pass permission sets to precise methods, but reconstructs the permissions when a precise method calls an imprecise methods. Applying this technique to exclusion frames, as described, would ensure that exclusion frames do not affect the run-time performance of methods that are specified with completely precise specifications.

9 FUTURE WORK

There are many possible directions in which this work can be extended. We have not yet proven the gradual guarantees for gradual verification, as formalized in Wise et al. [2020]. These guarantees formalize the notion that, given a valid program, gradual specifications may be used in place of all static specifications without introducing errors (both during verification and at run time). This ensures that any errors do not arise from imprecision, but rather from an invalid program or specification, or (for precise specifications) from incompleteness of verification. Our formalization appears to satisfy this since, as described in §3, we extend the underlying static verification algorithm mainly by adding optimistic capabilities while leaving the bulk of static verification intact. However, we have not completed a formal proof.

Our formalization could also be used to extend gradual verification. Notably, gradual verification has not been implemented for quantified specifications or concurrent programs. Ghost code/parameters (i.e., code only necessary for supporting logical proofs) is also not supported in gradual verification, since the "ghost" code could be necessary for run-time checks. Our high-level definition of the gradual verifier could enable further development to support these techniques. Likewise, our formalization does not capture several important concepts in Viper such as domains, fractional permissions, and joining of symbolic execution paths. Formalizing the usage of these techniques in Viper and proving their soundness would provide further assurance of the correctness of Viper and provide a starting point for integrating these techniques with gradual verification.

Our formalization provides a basis for formally proving properties of verification techniques (in our case, gradual verification) with a model that closely resembles the implementation (in our case, Gradual C0). Thus modifications to our formal model can be more easily implemented and used, while modifications to the implementation can be reflected in the formal model and proven sound.

10 RELATED WORK

As mentioned previously, implementations of verification using symbolic execution, such as Viper [Schwerhoff 2016], Gradual C0 [DiVincenzo et al. 2022], Smallfoot [Berdine et al. 2005], Chalice [Leino et al. 2009], and jStar [Distefano and Parkinson 2008], often lack formal soundness proofs. A notable exception is VeriFast [Jacobs et al. 2011], which implements verification using symbolic execution. The core of its verifier was proven sound in Vogels et al. [2015]. This soundness proof utilizes techniques from abstract interpretation, which may simplify proofs of verifiers using symbolic execution. However, VeriFast uses separation logic instead of IDF.

Several previous verifiers using WLP or verification condition generation (VCG) have been directly proven sound [Herms et al. 2012; Smans et al. 2012; Vogels et al. 2009, 2010]. Several similar verifiers produce a proof during verification which may be checked to the validate soundness of an individual verification result [Filliâtre and Paskevich 2013; Parthasarathy et al. 2021].

Viper [Müller et al. 2016] and Gradual C0 [DiVincenzo et al. 2022] rely on an SMT solver to implement their verification algorithms. While we have proved soundness of our formal model, this soundness is contingent on the soundness of the SMT solver. Other work has extended soundness to include soundness of the entire verification system. Notably, VeriSmall [Keuchel et al. 2022], Diaframe [Mulder et al. 2022], and RefinedC [Sammler et al. 2021] are all encoded in Iris/Coq, making them either foundational or self-verifying.

As described before, soundness of gradual verification based on WLP has been proven in both Wise et al. [2020] and Bader et al. [2018]. However, Wise et al. [2020] depends on dynamically checking all assertions, while Bader et al. [2018] does not handle abstract heap predicates.

11 CONCLUSION

The recent implementation of gradual verification in DiVincenzo et al. [2022] promises a dramatic reduction in the effort required to verify programs. However, this requires confidence in the correctness of their gradual verification system, Gradual C0, as well as its underlying static verification system, Viper. In this work, we formalized symbolic execution in (a subset of) Viper and proved it sound, in addition to formalizing gradual verification in Gradual C0 and proving it sound. During this work we found a soundness bug in Gradual C0, which we communicated to DiVincenzo et al. [2022] along with possible solutions. This illustrates that, while correctness in gradual verifiers can be guaranteed, it should not be assumed without rigorous proof. There are a few interesting directions we could take this work: (1) proving that Gradual C0 adheres to the gradual guarantee as formalized by Wise et al. [2020], which is a very important property of gradual verifiers that should be straightforward to prove with our formal system, and (2) using our formalism to explore new directions in gradual verification like quantification or concurrency, and prove systems utilizing them sound. In general, we hope that this work serves as a strong basis for future proof work in static and gradual verification when using symbolic execution.

ACKNOWLEDGMENTS

We thank Jana Dunfield for her helpful feedback.

This work was supported by the National Science Foundation under Grant No. CCF-1901033 (https://www.nsf.gov/awardsearch/showAward?AWD_ID=1901033) and a Google PhD Fellowship.

REFERENCES

Rob Arnold. 2010. *C0, an Imperative Programming Language for Novice Computer Scientists.* Master's thesis. Department of Computer Science, Carnegie Mellon University. http://reports-archive.adm.cs.cmu.edu/anon/anon/usr/ftp/home/ftp/2010/CMU-CS-10-145.pdf

Vytautas Astrauskas, Aurel Bílý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13260)*, Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer, 88–108. https://doi.org/10.1007/978-3-031-06773-0_5

Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual Program Verification. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10747)*, Isil Dillig and Jens Palsberg (Eds.). Springer, 25–46. https://doi.org/10.1007/978-3-319-73721-8_2

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Lecture Notes in Computer Science, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 115–137. https://doi.org/10.1007/11804192_6

Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 2017. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10510)*, Nadia Polikarpova and Steve A. Schneider (Eds.). Springer, 102–110. https://doi.org/10.1007/978-3-319-66845-1_7

Dino Distefano and Matthew J. Parkinson. 2008. JStar: Towards Practical Verification for Java. *SIGPLAN Not.* 43, 10 (oct 2008), 213–226. https://doi.org/10.1145/1449955.1449782

Jenna DiVincenzo, Ian McCormack, Hemant Gouni, Jacob Gorenburg, Mona Zhang, Conrad Zimmerman, Joshua Sunshine, Éric Tanter, and Jonathan Aldrich. 2022. Gradual C0: Symbolic Execution for Efficient Gradual Verification. arXiv:2210.02428 [cs.LO]

Marco Eilers and Peter Müller. 2018. Nagini: A Static Verifier for Python. 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, 596–603. https://doi.org/10.1007/978-3-319-96145-3_33

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. (2016), 429–442. https://doi.org/10.1145/2837614.2837670

Paolo Herms, Claude Marché, and Benjamin Monate. 2012. A Certified Multi-prover Verification Condition Generator. In *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7152)*, Rajeev Joshi, Peter Müller, and Andreas Podelski (Eds.). Springer, 2–17. https://doi.org/10.1007/978-3-642-27705-4_2

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. https://doi.org/10.1145/363235.363259

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. 2022. Verified Symbolic Execution with Kripke Specification Monads (and No Meta-Programming). *Proc. ACM Program. Lang.* 6, ICFP, Article 97 (aug 2022), 31 pages. https://doi.org/10.1145/3547628

Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S. Foster. 2010. Mixing Type Checking and Symbolic Execution. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 436–447. https://doi.org/10.1145/1806596.1806645

K. Rustan M. Leino, Peter Müller, and Jan Smans. 2009. Verification of Concurrent Programs with Chalice. In *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures (Lecture Notes in Computer Science, Vol. 5705)*, Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri (Eds.). Springer, 195–222. https://doi.org/10.1007/978-3-642-

03829-7_7

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 809–824. https://doi.org/10.1145/3519939.3523432

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Matthew Parkinson and Gavin Bierman. 2005. Separation Logic and Abstraction. *SIGPLAN Not.* 40, 1, 247–258. https://doi.org/10.1145/1047659.1040326

Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. 2021. Formally Validating a Practical Verification Condition Generator. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 704–727. https://doi.org/10.1007/978-3-030-81688-9_33

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. https://doi.org/10.1145/3453483.3454036

Malte Schwerhoff. 2016. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. Ph. D. Dissertation. ETH Zurich, Zürich, Switzerland. https://doi.org/10.3929/ETHZ-A-010835519

Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit Dynamic Frames. *ACM Trans. Program. Lang. Syst.* 34, 1, Article 2 (may 2012), 58 pages. https://doi.org/10.1145/2160910.2160911

Alexander J. Summers and Sophia Drossopoulou. 2013. A Formal Semantics for Isorecursive and Equirecursive State Abstractions. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7920)*, Giuseppe Castagna (Ed.). Springer, 129–153. https://doi.org/10.1007/978-3-642-39038-8_6

Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2009. A Machine Checked Soundness Proof for an Intermediate Verification Language. In *SOFSEM 2009: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 24-30, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5404)*, Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia (Eds.). Springer, 570–581. https://doi.org/10.1007/978-3-540-95891-8_51

Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2010. A Machine-Checked Soundness Proof for an Efficient Verification Condition Generator. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (Sierre, Switzerland) *(SAC '10)*. Association for Computing Machinery, New York, NY, USA, 2517–2522. https://doi.org/10.1145/1774088.1774610

Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2015. Featherweight VeriFast. *Log. Methods Comput. Sci.* 11, 3 (2015). https://doi.org/10.2168/LMCS-11(3:19)2015

Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual Verification of Recursive Heap Data Structures. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 228 (nov 2020), 28 pages. https://doi.org/10.1145/3428296

Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2024. Sound Gradual Verification with Symbolic Execution. (2024). arXiv:2311.07559 [cs.PL]

Received 2023-07-11; accepted 2023-11-07