Gradual C0: Symbolic Execution for Gradual Verification

JENNA DIVINCENZO, Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana, USA
IAN MCCORMACK, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
CONRAD ZIMMERMAN, Northeastern University, Boston, Massachusetts, USA
HEMANT GOUNI, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
JACOB GORENBURG, Haverford College, Haverford, Pennsylvania, USA
JAN-PAUL RAMOS-DÁVILA, Cornell University, Ithaca, New York, USA
MONA ZHANG, Columbia University, New York, New York, USA
JOSHUA SUNSHINE, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
ÉRIC TANTER, Department of Computer Science, University of Chile, Santiago, Chile
JONATHAN ALDRICH, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA

Current static verification techniques such as separation logic support a wide range of programs. However, such techniques only support complete and detailed specifications, which places an undue burden on users. To solve this problem, prior work proposed gradual verification, which handles complete, partial, or missing specifications by soundly combining static and dynamic checking. Gradual verification has also been extended to programs that manipulate recursive, mutable data structures on the heap. Unfortunately, this extension does not reward users with decreased dynamic checking as more specifications are written and more static guarantees are made. In fact, all properties are checked dynamically regardless of any static guarantees. Additionally, no full-fledged implementation of gradual verification exists so far, which prevents studying its performance and applicability in practice.

We present Gradual C0, the first practicable gradual verifier for recursive heap data structures, which targets C0, a safe subset of C designed for education. Static verifiers supporting separation logic or implicit dynamic frames use symbolic execution for reasoning; so Gradual C0, which extends one such verifier, adopts symbolic

This material is based upon work supported by a Google PhD Fellowship award and the National Science Foundation under Grant Nos. CCF-1901033, DGE1745016, and DGE2140739. É. Tanter is partially funded by the ANID FONDECYT Regular Project 1190058 and the Millennium Science Initiative Program: code ICN17_002. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, Google, ANID, or the Millennium Science Initiative.
Authors' Contact Information: Jenna DiVincenzo (corresponding author), Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana, USA; e-mail: jennad@purdue.edu; Ian McCormack, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; e-mail: icmccorm@cs.cmu.edu; Conrad Zimmerman, Northeastern University, Boston, Massachusetts, USA; e-mail: zimmerman.co@northeastern.edu; Hemant Gouni, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; e-mail: hsgouni@cs.cmu.edu; Jacob Gorenburg, Haverford College, Haverford, Pennsylvania, USA; e-mail: jsgorenburg@gmail.com; Jan-Paul Ramos-Dávila, Cornell University, Ithaca, New York, USA; e-mail: jvr34@cornell.edu; Mona Zhang, Columbia University, New York, New York, USA; e-mail: mz2781@columbia.edu; Joshua Sunshine, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; e-mail: sunshine@cs.cmu.edu; Éric Tanter, Department of Computer Science, University of Chile, Santiago, Chile; e-mail: etanter@dcc.uchile.cl; Jonathan Aldrich, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; e-mail: jonathan.aldrich@cs.cmu.edu.

ACM 1558-4593/2025/1-ART14
https://doi.org/10.1145/3704808

execution at its core instead of the weakest liberal precondition approach used in prior work. Our approach addresses technical challenges related to symbolic execution with imprecise specifications, heap ownership, and branching in both program statements and specification formulas. We also deal with challenges related to minimizing insertion of dynamic checks and extensibility to other programming languages beyond C0. Finally, we provide the first empirical performance evaluation of a gradual verifier, and found that on average, Gradual C0 decreases run-time overhead between 7.1 and 40.2% compared to the fully dynamic approach used in prior work (for context, the worst cases for the approach by Wise et al. [2020] range from 0.1 to 4.5 seconds depending on the benchmark). Further, the worst-case scenarios for performance are predictable and avoidable. This work paves the way towards evaluating gradual verification at scale.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Automated reasoning**; **Hoare logic**; **Separation logic**;

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

ACM Reference format:
Jenna DiVincenzo, Ian McCormack, Conrad Zimmerman, Hemant Gouni, Jacob Gorenburg, Jan-Paul Ramos-Dávila, Mona Zhang, Joshua Sunshine, Éric Tanter, and Jonathan Aldrich. 2025. Gradual C0: Symbolic Execution for Gradual Verification. *ACM Trans. Program. Lang. Syst.* 46, 4, Article 14 (January 2025), 57 pages.
https://doi.org/10.1145/3704808

1 Introduction

Separation logic [Reynolds, 2002] supports the modular static verification of heap-manipulating programs, including ones that contain recursion. Its variant **Implicit Dynamic Frames (IDF)** [Smans et al., 2009] and extension with *recursive abstract predicates* [Parkinson and Bierman, 2005; Smans et al., 2009] further support verifying recursive heap data structures, such as trees, lists, and graphs. While these techniques allow users to specify and verify more code than ever before, tools implementing them (e.g., Viper [Müller et al., 2016], VeriFast [Jacobs et al., 2011], Chalice [Leino et al., 2009], JStar [Distefano and Parkinson, 2008], and SmallFoot [Berdine et al., 2006]) are still largely unused due to the burden they place on their users. Such tools poorly support partial specifications, and thus require users to provide a number of auxiliary specifications (such as folds, unfolds, loop invariants, and inductive lemmas) in an all or nothing fashion to support inductive proofs of correctness. The tools also require many of these auxiliary specifications to be written before they can provide feedback on the correctness of specifications for important functional properties. For example, to prove that a simple insertion function preserves list acyclicity, static verifiers need 1.5 times as many lines of auxiliary specifications to program code (Section 2). They also need a significant number of these auxiliary specifications to uncover problems with specifications of the acyclic property (Section 2).

Inspired by gradual typing [Siek and Taha, 2006], Bader et al. [2018] proposed *gradual verification* to support the incremental specification and verification of software. Users can write *imprecise* (i.e., partial) specifications backed by run-time checking where necessary. An imprecise formula can be fully unknown, written ?, or combine a *static part* with the unknown, as in $? * x.f == 2$. Wise et al. [2020] extend the initial system by Bader et al. [2018] by designing and formalizing the first gradual verifier for recursive heap data structures. It supports imprecise specifications with accessibility predicates from IDF and abstract predicates; and thus, also (in theory) the run-time verification of these constructs. During static verification, an imprecise specification can be optimistically strengthened (in noncontradictory ways) by the verifier to support proof goals. Wherever such strengthenings occur, dynamic checks are inserted to preserve soundness. Gradual verification smoothly supports the spectrum between static and dynamic verification. This is captured by

properties adapted from gradual typing [Siek et al., 2015], namely the *gradual guarantee*, stating that the verifier will not flag static or dynamic errors for specifications that are correct but imprecise, and the fact that gradual verification *conservatively extends* static verification, i.e., they coincide on fully precise programs.

While promising, the gradual verifier by Wise et al. [2020] has neither been implemented nor validated in practice. Furthermore, their design relies on *weakest liberal preconditions* [Dijkstra, 1975] for static reasoning rather than *symbolic execution* [King, 1976], which is the ideal reasoning technique for tools based on separation logic or IDF. Indeed, Viper [Müller et al., 2016], VeriFast [Jacobs et al., 2011], JStar [Distefano and Parkinson, 2008], and SmallFoot [Berdine et al., 2006] all support these permission logics with symbolic execution, not weakest liberal preconditions. Finally, the gradual verifier by Wise et al. [2020] is not efficient in the sense that it checks all heap ownership and functional properties dynamically regardless of the precision of specifications.

This article presents the design, implementation, and validation of Gradual C0[1]—the first gradual verifier for imperative programs manipulating recursive heap data structures that is based on symbolic execution. Gradual C0 targets C0, a safe subset of C designed for education, with appropriate support (and pedagogical material) for dynamic verification. Technically, Gradual C0 is built on top of the Viper static verification infrastructure [Müller et al., 2016], which facilitates the development of program verifiers supporting IDF and recursive abstract predicates. Gradual C0's back-end leverages this infrastructure to simplify the implementation of gradual verifiers for other programming languages, and Gradual C0's front-end demonstrates how this is done for C0. Further, Gradual C0 minimizes the insertion of dynamic checks using statically available information and optimizes the checks' overhead at runtime.

Overall, we address new technical challenges in gradual verification related to symbolic execution, extensibility to multiple programming languages, and minimizing run-time checks and their overhead:

- —Gradual C0's symbolic execution algorithm is responsible for statically verifying programs with imprecise specifications and producing minimized run-time checks. In particular, Gradual C0 tracks the branch conditions created by program statements and specifications to produce run-time checks for corresponding execution paths. At run time, branch conditions are assigned to variables at the branch point that introduced them, which are then used to coordinate the successive checks as required. Further, Gradual C0 creates run-time checks by translating symbolic expressions into specifications—reversing the symbolic execution process.
- —The run-time checks produced by Gradual C0 contain branch conditions, simple logical expressions, accessibility predicates, separating conjunctions, and predicates. Each of these constructs is specially translated into source code that can be executed at run-time for dynamic verification. Logical expressions are turned into assertions. Accessibility predicates and separating conjunctions are checked by tracking and updating a set of owned heap locations. Finally, predicates are translated into recursive Boolean functions. By encoding run-time checks into C0 source code, we avoid complexities from augmenting the C0 compiler to support dynamic verification. We also design these encodings to be performance friendly, e.g., owned heap locations are tracked in a dynamic hash table.

Work on gradual typing performance has shown that minimizing the insertion of dynamic checks does not trivially correlate with overall execution performance; the nature of the inserted checks (such as higher-order function wrappers) as well as their location in the overall execution flow of a

[1]Gradual C0 is hosted on Github: https://github.com/gradual-verification/gvc0

program can have drastic and hard-to-predict consequences [Campora et al., 2018; Feltey et al., 2018; Takikawa et al., 2016]. Therefore, our validation of Gradual C0 aims to empirically evaluate the relationship between minimizing check insertion and observed run-time performance in gradual verification. We evaluate the performance of Gradual C0 by adapting the performance lattice method of Takikawa et al. [2016] to gradual verification, exploring the performance characteristics for partial specifications of four common data structures. This method models the gradual verification process as a series of steps of partial specifications from an unspecified program (containing all ?s) to a statically verifiable specification (not containing any ?s) where, at each step, an atomic conjunct is added to the current, partial specification. Statically, we observe that as more specifications are added, more verification conditions can be statically discharged. Though imprecision introduces unavoidable run-time checks, gradual verification decreases run-time overhead by an average of 7.1–40.2% compared to dynamic verification (and thus the approach by Wise et al. [2020]; for context, the worst cases for the approach by Wise et al. [2020] range from 0.1 to 4.5 seconds depending on the benchmark). Sources of run-time overhead correspond to the predictions made in prior work, and our study shows that the gradual guarantee holds empirically for our tool across thousands of sampled imprecise specifications. A reproducibility package for this evaluation is provided as supplementary material.

Our work is complimentary to recent work by Zimmerman et al. [2024], and distinct in contributions. Our work includes the original development of symbolic execution based gradual verification and describes the first implementation of the same, as well as related empirical results. The paper by Zimmerman et al. [2024] proves that the approach in this article is sound by formalizing our design at a higher level of abstraction, which is more useful for proofs.

2 Gradual C0 Improves the Static Specification Process

```
1  struct Node { int val; struct Node* next; };
2  typedef struct Node Node;
3
4  Node* insertLast(Node* list, int val)
5  {
6      Node* y = list;
7      while (y->next != NULL)
8        { y = y->next; }
9      y->next = alloc(struct Node);
10     y->next->val = val;
11     y->next->next = NULL;
12     return list;
13 }
```

Fig. 1. Non-empty linked list insertion in C0.

Static verification tools, like Viper [Müller et al., 2016], VeriFast [Jacobs et al., 2011], Chalice [Leino et al., 2009], JStar [Distefano and Parkinson, 2008], and SmallFoot [Berdine et al., 2006], require a number of user-provided auxiliary specifications, such as folds, unfolds, lemmas, and loop invariants, to prove properties about recursive heap data structures. Worse, they also require users to write many of these auxiliary specifications before the tools can provide useful feedback on the correctness of other specifications, including ones containing important functional properties. Therefore, users are burdened by writing many detailed and extraneous specifications with inadequate static feedback through the process. In this section, we illustrate this burden with a simple list insertion example (inspired by a similar introductory example and discussion from Wise et al. [2020]) and output from Viper. Then, we show in Section 3 how Gradual C0 overcomes this burden by smoothly supporting the spectrum between static and dynamic checking. Users can avoid writing auxiliary specifications and still get sound verification of their code with increased run-time checking. Users can also receive run-time feedback on the correctness of their specifications very early in the specification process, and the resulting error messages closely align with inherent problems in the specifications or in the program, making debugging them easier.

Figure 1 implements a linked list and function that inserts a new node at the end of a given list, called `insertLast`, in C0 [Arnold, 2010]. The `insertLast` function traverses the list to its end with

```
1  /*@ predicate acyclicSeg(Node* s, Node* e) =
2        (s == e) ? true : acc(s->val) && acc(s->next) && acyclicSeg(s->next,e); @*/
3  //@ predicate acyclic(Node* n) = acyclicSeg(n, NULL);

4  Node* insertLast(Node* list, int val)
5    //@ requires acyclic(list) && list != NULL;
6    /*@ ensures acyclic(\result) &&
7                        \result != NULL; @*/
8  {
9      //@ unfold acyclic(list);
10     //@ unfold acyclicSeg(list, NULL);
11     Node* y = list;
12     //@ fold acyclicSeg(list, y);
13     while (y->next != NULL)
14     /*@ loop_invariant acyclicSeg(list, y) &&
15           acc(y->next) && acc(y->val) &&
16           acyclicSeg(y->next, NULL); @*/
17     {
18       Node* tmp = y;
19       y = y->next;
20       //@ unfold acyclicSeg(y, NULL);
21       //@ fold acyclicSeg(tmp->next, y);
22       //@ fold acyclicSeg(tmp, y);
23       mergeLemma(list, tmp, y);
24     }
25
26

27       y->next = alloc(struct Node);
28       y->next->val = val;
29       y->next->next = NULL;
30       //@ fold acyclicSeg(y->next->next, NULL);
31       //@ fold acyclicSeg(y->next, NULL);
32       //@ fold acyclicSeg(y, NULL);
33       mergeLemma(list, y, NULL);
34       //@ fold acyclic(list);
35       return list;
36 }
37
38 void mergeLemma (Node* a, Node* b, Node* c)
39   //@ requires acyclicSeg(a, b) &&
40                acyclicSeg(b, c);
41   //@ ensures acyclicSeg(a, c);
42  {
43       if (a == b) {
44       } else {
45            //@ unfold acyclicSeg(a, b);
46            mergeLemma(a->next, b, c);
47            //@ fold acyclicSeg(a, c);
48       }
49 }
```

□ Program code ■ Static specification

Fig. 2. The static verification of `insertLast` from Figure 1.

a `while` loop starting from the root. That is, `insertLast` implicitly assumes the list is non-empty (non-null) and acyclic; and that for multiple successive calls to `insertLast` the list remains acyclic and non-empty after insertion. These facts can be proven explicitly with static verification; the complete static specification is given in Figure 2, highlighted in gray.

List acyclicity is specified with two predicates `acyclicSeg` and `acyclic`:

```
predicate acyclicSeg(Node* s, Node* e) =
    s == e ? true : acc(s->val) && acc(s->next) && acyclicSeg(s->next, e)
predicate acyclic(Node* n) = acyclicSeg(Node* n, NULL)
```

The `acyclicSeg` predicate uses *accessibility predicates* (e.g., **acc**(`s->val`) and **acc**(`s->next`)) and the *separating conjunction* (e.g., &&) from IDF [Smans et al., 2009]. In IDF, *accessibility predicates* are used to denote ownership of heap locations, e.g., **acc**(`s->val`) && `s->val == 2` states that `s->val` is uniquely owned and contains the value 2 and `s != NULL`. Successful verification of a function or loop requires accessibility predicates to be available for each heap location accessed by the function or loop before the access point. For example, an IDF-based verifier (like Viper or Gradual C0) ensures **acc**(`y->next`) is available to *frame* `y->next` in `insertLast`'s loop condition on line 7 in Figure 1 before the loop; otherwise verification fails due to lack of permission to access `y->next`. Accessibility predicates are only gained at allocations, e.g., **acc**(`y->next->val`) and **acc**(`y->next->next`) on line 9 in Figure 1 and lost at de-allocations. Only one function or loop may have ownership of a heap location at a time, so accessibility predicates are transferred between functions and loops as specified by pre- and postconditions and loop invariants. Since

the heap location for `y->next` (i.e., `list->next`) is allocated in `insertLast`'s callee, `insertLast`'s precondition needs to specify **`acc`**`(list->next)` so ownership of the heap location can be passed from the callee to `insertLast` and verification can succeed. Otherwise, the callee will retain ownership and verification of `y->next` fails. Thanks to the aforementioned ownership tracking, IDF-based verifiers can guarantee the absence of access errors, such as dangling or null-pointer dereferences, and memory leaks, such as double frees, mismatched frees, and invalid frees.

The *separating conjunction*, denoted by &&,[2] ensures memory disjointness: **`acc`**`(s->next)` && **`acc`**`(s->next->next)` states that the heap locations `s->next` and `s->next->next` are distinct (i.e., $s \neq$ `s->next`) and are each owned. Together, accessibility predicates and the separating conjunction provide a way for specifications to describe parts of the heap, which is useful for ownership transfer and specifying properties about the heap. For example, the recursive predicate `acyclicSeg(s, e)` denotes that all heap locations in list `s` are distinct up to node `e` by recursively generating accessibility predicates for each node in `s` up to `e`, joined with the separating conjunction. Thus, `acyclicSeg` specifies that a list segment is acyclic. Further, `acyclicSeg(n, NULL)` denotes that all heap locations in list `n` are distinct and so `n` is acyclic, as specified with `acyclic(n)`.

Now that we have specified `acyclic`, we use it in `insertLast`'s precondition (line 5) and postcondition (lines 6–7) to denote preservation of list acyclicity and denote `insertLast` only accesses a given list's heap locations and returns access to the heap locations of `insertLast`'s returned list. We also specify that `insertLast` preserves list non-nullness with simple comparison logic (i.e., `list != NULL` and `\result != NULL`). Ideally, we would stop here and static verifiers would be able to prove `insertLast`'s implementation is correct with respect to this specification; however, as you can see in Figure 2 such tools require many more specifications. In fact, there are 27 lines of auxiliary specifications (comprised of folds, unfolds, loop invariants, and inductive lemmas); in contrast to 18 lines of wanted specifications (the predicates and pre- and postconditions) and program code. Furthermore, these auxiliary specifications are complex, as discussed next.

2.1 Auxiliary Specifications

Static verifiers cannot reliably unroll recursive predicates during verification; so, such tools rely on explicit *fold* and *unfold* statements to control the availability of predicate information in the verifier. This treats predicates *iso-recursively*; while an *equi-recursive* interpretation treats predicates as their complete unrolling [Summers and Drossopoulou, 2013]. Consequently, the `acyclic` and `acyclicSeg` predicates are unfolded and folded often in Figure 2 (lines 9–10, 12, 20–22, 30–32, and 34). Looking closely, we see that `acyclic(list)`, which is assumed true from the precondition, is unfolded on line 9. This consumes `acyclic(list)` and produces its body `acyclicSeg(list, NULL)`, which is subsequently unfolded on line 10. Then, at the fold on line 34, the body of `acyclic(list)` is packed up into the predicate itself to prove the list remains acyclic after insertion.

Additionally, static verifiers cannot tell if or when a loop will end (in our example the verifier cannot tell when the list being iterated over ends), but must verify all paths through the program. Therefore, static verifiers reason about loops using specifications called *loop invariants*, which are properties that are preserved for each execution of the loop including at entry and exit. Further, loop invariants must also provide information necessary for proof obligations after the loop, e.g., that the list in `insertLast` is acyclic after insertion. In Figure 2, these constraints result in the loop invariant on lines 14–16 that segments the list into three disjoint and acyclic parts: from the root up to the current node `y` (`acyclicSeg(list, y)`), the current node `y` (**`acc`**`(y->val)` && **`acc`**`(y->next)`), and from the node after `y` to the end (`acyclicSeg(y->next, NULL)`). Exposing `y` via its accessibility

[2]Most IDF papers use $*$ to denote the separating conjunction. We follow Viper and use && instead.

```
1  Node* insertLast(Node* list, int val)
2    //@ requires acyclic(list) && list != NULL;
3    /*@ ensures acyclic(\result) &&
4                  \result != NULL; @*/
5  {
6    //@ unfold acyclic(list);
7    //@ unfold acyclicSeg(list, NULL);
8    Node* y = list;
9    //@ fold acyclicSeg(list, y);
10   while (y->next != NULL)
11   /*@ loop_invariant acyclicSeg(list, y) &&
12         acc(y->next) && acc(y->val) &&
13         acyclicSeg(y->next, NULL); @*/
14   {
15     Node* tmp = y;
16     y = y->next;
17     //@ unfold acyclicSeg(y, NULL);
18     //@ fold acyclicSeg(tmp->next, y);
19     //@ fold acyclicSeg(tmp, y);
20     mergeLemma(list, tmp, y);
21   }
22
23   y->next = alloc(struct Node);
24   y->next->val = val;
25   y->next->next = NULL;
26   //@ fold acyclicSeg(y->next->next, NULL);
27   //@ fold acyclicSeg(y->next, NULL);
28   //@ fold acyclicSeg(y, NULL);
29   mergeLemma(list, y, NULL);
30   //@ fold acyclic(list);
31   return list;
32 }
```

1^{st} increment (least precise) 2^{nd} increment 3^{rd} increment 4^{th} increment
5^{th} increment 6^{th} increment 7^{th} increment (full spec)

Fig. 3. The incremental verification of `insertLast` from Figure 1.

predicates provides access to `y->next` on line 19 in the loop body; and, `acyclicSeg(list, y)` helps prove `acyclic(list)` holds after the loop, as we will see next.

To prove `acyclic(list)` holds at the end of `insertLast` (line 35), it is sufficient to prove instead that `acyclicSeg(list, NULL)` holds (line 34). After inserting a new node at the end of the list (lines 27–29), we can build up an inductive proof with folds (lines 30–32) that the list is acyclic from the insertion point `y` to the new end, i.e., `acyclicSeg(y, NULL)` holds. We also have that the list is acyclic from the root to `y` (`acyclicSeg(list, y)`) from to the loop invariant, and so, we are done after proving transitivity of acyclic list segments, i.e., `acyclicSeg(list, y)` and `acyclicSeg(y, NULL)` implies `acyclicSeg(list, NULL)`. Sadly, static verifiers cannot automatically discharge such inductive proofs, and so we specify the proof steps in `mergeLemma` on lines 38–49. Then, after using the lemma on line 33, we achieve our proof goal.

As we can see, not only do users of static verifiers need to write a number of auxiliary specifications in support of proof goals, the specifications are often more complex compared to the program code itself even for simple examples like `insertLast`. Worse even, is that while users are developing these complex specifications static tools provide limited feedback on their correctness as demonstrated next (Section 2.2).

2.2 Lack of Early Specification Feedback

Since static verifiers, like Viper, limit themselves to reasoning about predicates iso-recursively and rely on loop invariants to prove properties about loops, feedback on the correctness of specifications early in the specification process is limited. For example, consider that a user named Daisy incorrectly specifies the body of `acyclicSeg` (our recursive predicate) as `(s == e) ? acc(s->val) && acc(s->next) && acyclicSeg(s->next, e) : true`, which swaps the branches of the ternary in the correct specification from Figure 2. Let's see how Daisy comes across this error while using Viper to incrementally specify `insertLast` in Figure 3. Each increment from the first to the last (seventh) is highlighted in a different color. The first increment, highlighted in green, specifies the precondition and postcondition of `insertLast` with `acyclic` and `acyclicSeg` (lines 2–4). Since predicates are black boxes in static verifiers, Viper only tells Daisy that there is insufficient permission to access `y->next` in the loop condition on line 10. So, Daisy specifies the required

acc(y->next) permission (as highlighted in purple for the second increment on line 12) in the loop invariant, which must frame the loop condition. But, alas Viper cannot prove that **acc**(y->next) holds (is available) on entry to the loop. Without realizing the branches of acyclicSeg are out of order, Daisy expects acyclicSeg(list, NULL)'s body to provide **acc**(y->next) at loop entry as list != NULL and y == list; and so, she unfolds acyclic(list) and acyclicSeg(list, NULL) on lines 6–7 making up the third specification increment highlighted in brown/rose gold. Unfortunately, Viper still reports that **acc**(y->next) does not hold on entry to the loop, which alerts Daisy to the problem with acyclicSeg.

With Viper, Daisy required three specification increments to detect a bug in the first unfolding of acyclicSeg, and this problem gets worse the deeper the bug is in the recursive predicate. For example, consider now that acyclicSeg's body is incorrectly specified as (s == e) ?true : **acc**(s->val) && **acc**(s->next) && acyclicSeg(e, s->next), which swaps s->next and e in the recursive call to acyclicSeg. As a result, acyclicSeg asserts in lock-step that the nodes in lists s and e are accessible and separated—which is not the intended behavior of acyclicSeg—and acyclicSeg now always fails when e reaches its end, i.e., is NULL, as it tries to assert **acc**(e->val) and **acc**(e->next). It takes until the fourth specification increment highlighted in blue to discover that acyclicSeg is incorrectly specified using Viper. As before, Daisy is led to specifying the first three increments by Viper's error messages that first require **acc**(y->next) in the loop invariant and then require **acc**(y->next) to hold on entry to the loop. This time, however, acyclicSeg(list, NULL)'s body contains **acc**(y->next) when list != NULL, so Viper can prove **acc**(y->next) holds on loop entry and instead reports that the loop invariant **acc**(y->next) might not be preserved by the loop body. Daisy recalls the loop iterates over all nodes in the list with the current node being y, so preserving **acc**(y->next) in the loop is the same as showing that Viper has accessibility predicates for every node in the list. As a result, she specifies the fourth increment (lines 12–13 and 17), which continuously unfolds the acyclicSeg(list, NULL) predicate on every iteration of the loop and captures the information in its body in the loop invariant. Alas, Viper reports that the new loop invariant does not hold on entry as it cannot prove acyclicSeg(y->next, NULL) holds here. Since this information should come from unfolding acyclicSeg(list, NULL) on line 7, Daisy takes another look at acyclicSeg's body and discovers her specification error. That is, it takes four specification increments for Daisy to realize her mistake and the fourth increment required her to think deeply about her while loop.

Clearly, static verifiers burden their users, like Daisy, by requiring them to write a number of complex auxiliary specifications both for proofs and to receive useful feedback on the correctness of their specifications. Fortunately, as we will show next in Section 3, Gradual C0 overcomes this burden by smoothly supporting the spectrum between static and dynamic checking.

3 Gradual C0 to the Rescue

In this section, we show how Gradual C0's ability to smoothly integrate static and dynamic checking allows users to overcome specification burdens inherent to static verification, e.g., that users have to write complex auxiliary specifications in support of proofs and to receive useful feedback on the correctness of their specifications.

3.1 Ignore Auxiliary Specifications with Gradual C0

In Section 2.1, we saw how static verifiers force users to write complex auxiliary specifications (like folds and unfolds, loop invariants, and inductive lemmas) in support of proof goals. In contrast, Gradual C0 allows users to write as many or as few auxiliary specifications as they want, and instead utilizes dynamic verification to check proof obligations not discharged statically due to missing specifications. For example, consider Figure 3, in which our user Daisy incrementally

specifies `insertLast` for preservation of list acyclicity and non-nullness. With Gradual C0, Daisy only needs to specify the first increment in green, which contains the pre- and postcondition of `insertLast` on lines 2–4. She can completely avoid specifying the auxiliary specifications in the rest of the increments by instead specifying ? in the loop invariant on line 10. Then, Gradual C0 uses the allocation statement on line 23 to statically validate the write accesses of `y->next->val` and `y->next->next` on lines 24–25. It can also prove statically that the list after insertion is non-null. All other proof obligations, which ensure the while loop owns the heap locations it accesses and the list after insertion is acyclic, are checked dynamically as described in detail in Section 4.4.

At a high-level, to dynamically verify heap accesses, Gradual C0 tracks which concrete heap locations are owned at each program point during execution, and disallows heap accesses for locations not owned at that program point (program execution will stop with an error message when the program attempts to access an un-owned location). In the `insertLast` example, run time checking the while loop involves three parts: (1) verifying access to `y->next` in the loop condition (line 10), (2) verifying access to `y->next` in the loop body (line 16), and (3) verifying access to `y->next` in the alloc statement directly after the loop (line 23). That is, Gradual C0 checks that the loop receives ownership of all heap locations in the given list, which are being iterated over by the loop. In particular, when `insertLast` is called, Gradual C0 transfers ownership of dynamic heap locations allocated in `insertLast`'s callee to `insertLast` as specified in `acyclic(list) && list != NULL`. These are the heap locations making up `list`. Since `insertLast`'s loop invariant is ? and thus may denote any accessibility predicate, all heap locations owned by `insertLast` before the loop are conservatively passed to the loop, which successfully verifies its execution. Gradual C0 also successfully verifies `y->next` in the alloc statement (line 23), because Gradual C0 transfers ownership of `list`'s heap locations back to `insertLast` after the loop, as denoted by ?. Note, if `y->next` on line 16 were incorrectly written as `y->next->next`, then Gradual C0 would produce a heap ownership run-time error when `insertLast` is given a list with two nodes (a list with 1 node would not execute the loop body). After the loop body executes once, `y` would be equal to `NULL`, so `y->next` in the loop condition dereferences a null pointer. Gradual C0 catches this, because the only heap locations allocated and thus owned at this program point are the ones for `list`'s two nodes (`NULL->next` can never be owned).

Since `insertLast`'s while loop accesses every node in a given list and Gradual C0 verifies each access at run time, the larger the list the higher the run-time cost of verification. This cost is unacceptable to Daisy, so she statically specifies the while loop with the first four increments in Figure 3 (lines 2–4, 6–7, 12–13, and 17). The new loop invariant (lines 12–13) uses `acyclicSeg` to expose **`acc`**`(y->next)` for verifying access to `y->next` in the loop condition, loop body, and after the loop. The unfolds on lines 6–7 and 17 are used to prove that the loop invariant is preserved by the loop given the precondition on line 2. After all this work, Daisy is not interested in also statically specifying the list after insertion is acyclic, and so, she joins ? with her newly specified loop invariant. As a result, Gradual C0 now additionally checks heap accesses in the while loop statically reducing run-time cost, and only checks `acyclic(\result)` from the postcondition of `insertLast` (line 3) dynamically (i.e., the list returned after insertion is acyclic). Briefly, Gradual C0 checks `acyclic(\result)` by checking whether or not `acyclicSeg(\result, NULL)` is true; if `acyclicSeg(\result, NULL)` is false, Gradual C0 stops execution and produces an error message. Gradual C0 checks `acyclicSeg(\results, NULL)` by unrolling the predicate completely and checking the formulas inside—an equi-recursive treatment. That is, Gradual C0 asserts ownership of all the heap locations in the `\result` list, and also asserts that these heap locations are separated in memory. The `\result` list contains the heap locations from the list after the loop and the newly inserted node on line 23. Dynamic ownership transfer (described in the previous paragraph) guarantees the nodes in the list after the loop are accessible and separated in memory; and,

the allocation on line 23 guarantees ownership of the newly inserted node's heap locations and that these heap locations are distinct in memory from the ones in the list after the loop. So, `acyclicSeg(\result, NULL)` holds at run time and thus `acyclic(\result)` does too (more details can be found in Section 4.4 on how predicates are checked at run time in Gradual C0).

By allowing Gradual C0 to check `acyclic(\result)` dynamically, Daisy saved herself a lot of specification effort. She avoided building up `acyclicSeg` from the previous end of the list to the new one (increment five in orange, lines 26–28), specifying a more complex loop invariant (increment six in red, lines 9, 11, and 18–19), and stating and proving transitivity of acyclic list segments (increment seven in yellow, lines 20 and 29).[3] Daisy is very happy to make this human-effort vs. run-time cost tradeoff.

3.2 Gradual C0 Provides Earlier Feedback with Run-time Checks

As we saw in Section 2.2, static verifiers struggle to provide early feedback on specification errors in predicates. Using Viper, it took until the third specification increment in Figure 3 for Daisy to discover the simple error in `acyclicSeg`'s body, `(s == e) ? acc(s->val) && acc(s->next) && acyclicSeg(s->next, e) : true`, which swaps the branches of the ternary in the correct specification from Figure 2. In contrast, with Gradual C0, Daisy easily discovers this error on the first specification increment (in green, lines 2–4), which specifies the pre- and postcondition of `insertLast`. She additionally specifies `?` on the loop invariant and provides a simple test case that calls `insertLast` on a list with one node. Then, Gradual C0 alerts Daisy to the error in `acyclicSeg` by reporting at run time that `y->next` in the loop condition (line 10) is not owned at this point and therefore cannot be accessed. Daisy's test case allocates this heap location successfully (and `insertLast` up until line 10 is implemented correctly), so this error can only occur if ownership is not transferred to `insertLast` correctly via its precondition. Thus, Daisy takes another look at `acyclicSeg` with this in mind and realizes her specification error. Similarly, the second example of an error in `acyclicSeg`'s specification, which swaps `s->next` and `e` in the recursive call to `acyclicSeg` (`(s == e) ?true : acc(s->val) && acc(s->next) &&acyclicSeg(e, s->next)`), took four specification increments in Figure 3 to be exposed by Viper. Again with Gradual C0, Daisy can detect the error by the first increment as long as she specifies `?` on her loop invariant and supplies a simple test case with a list containing two elements. In this case, Gradual C0 reports at run time that `acc(s->val)` from `acyclicSeg` does not hold for `s` which is `NULL`. Since `acc(s->val)` will never hold when `s == NULL`, Daisy takes another look at `acyclicSeg`'s body and realizes her error. That is, Gradual C0's dynamic checking of partial specifications is helpful for detecting errors in recursive predicates much earlier in the specification process than static verification alone and the errors better capture the inherent problems in the specifications.

To summarize, users of Gradual C0 may write as many or as little auxiliary specifications as they want and still get sound verification of their code by trading off between human-effort and run-time cost. Users can also receive feedback on the correctness of their specifications much earlier in the specification process than if they used static verification alone and the run-time errors reported often closely match the inherent problems with the specification.

4 Gradual C0's Design and Implementation

Gradual C0 is a working gradual verifier for the C0 programming language [Arnold, 2010] that is built as an extension of the Viper static verifier [Müller et al., 2016]. Our goals for the design and

[3]Note, Gradual C0 does not support lemmas due to it being unclear how to compute termination for lemmas containing imprecision. Instead, we use a recursive function in Figures 2 and 3 to achieve the same result. Thus, `mergeLemma` is always executed when called, even when it is fully statically verified.

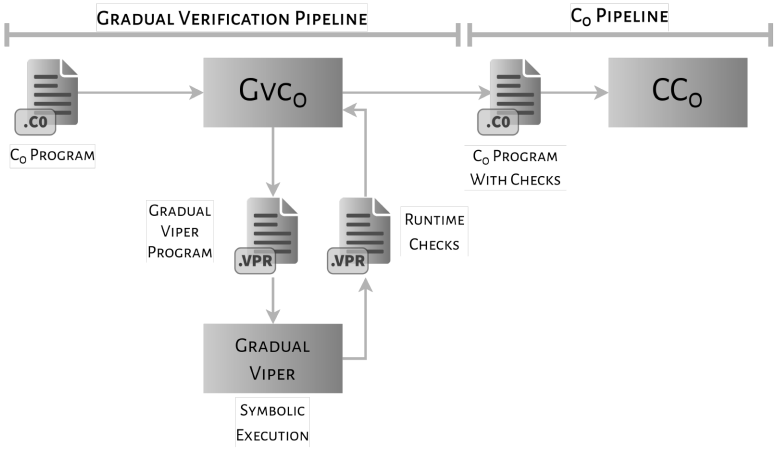

Fig. 4. System design of Gradual C0.

implementation of Gradual C0 are to:

—be easily extensible to other programming languages beyond C0,
—minimize run-time overhead from verification, and
—use symbolic execution for static reasoning.

Consequently, we settled on the design illustrated in Figure 4. Gradual C0 is structured in two major subsystems: (1) the gradual verification pipeline and (2) the C0 pipeline. Within the gradual verification pipeline, a C0 program is first translated AST-to-AST into a Gradual Viper program by Gradual C0's frontend module, GVC0. The GVC0 module implements a simple parser, abstract syntax, and type checker for C0 programs to facilitate the translation. The Gradual Viper module is the backend of Gradual C0 and implements its own parser, abstract syntax, and type checker for its own imperative language called the Gradual Viper language. This language is the Viper language plus imprecise formulas and allows Gradual Viper to support multiple frontend languages, not just C0. We chose to build a C0 frontend first because C0 is a pedagogical version of C designed with dynamic verification in mind, and we plan to use it in the classroom.

Once the C0 program is translated into a Gradual Viper program, it is optimistically statically verified by the Gradual Viper module. Gradual Viper extends Viper's symbolic execution-based verifier to support imprecise formulas and resulting holes in static reasoning as inspired by Wise et al. [2020] and gradual typing [Herman et al., 2010; Siek and Taha, 2006; Siek et al., 2015]. Consequently, by construction Gradual Viper supports full static verification of programs when specifications are complete. Differing from the work by Wise et al. [2020], Gradual Viper also extends the symbolic execution algorithm to create a description of needed run-time checks in support of static holes. The run-time checks are minimized with statically available information during reasoning. Finally, GVC0 takes these run-time checks and encodes them in the original C0 program to produce a sound, gradually verified program. The C0 pipeline takes this C0 program and feeds it to the C0 compiler, CC0, which is used to execute the program. Note, the encoding of checks into C0 source code optimizes for run-time performance and simplifies extending C0 with dynamic verification in our domain.

The rest of this section describes the implementation of Gradual Viper and GVC0's designs in more detail and illustrates the concepts via example. We also highlight design and implementation choices influenced by our goals. Section 4.1 discusses how C0 programs are translated to Gradual Viper programs, along with modifications made to both C0 and Viper for gradual verification.

x	$\in$ *VAR*	(*variables*)	S	$\in$ *STRUCTNAME*	(*struct names*)
v	$\in$ *VAL*	(*values*)	f	$\in$ *FIELDNAME*	(*field names*)
e	$\in$ *EXPR*	(*expressions*)	p	$\in$ *PREDNAME*	(*predicate names*)
s	$\in$ *STMT*	(*statements*)	m	$\in$ *METHODNAME*	(*method names*)
op	$\in$ +, -, /, *, ==, !=, <=, >=, <, >	(*operators*)			

Fig. 5. Shared abstract syntax definitions.

$$
\begin{array}{rcl}
\texttt{P} & ::= & \overline{struct}\ \overline{predicate}\ \overline{method} \\
struct & ::= & \texttt{struct}\ S\ \{\ \overline{T\ f}\ \} \\
predicate & ::= & \texttt{//@predicate}\ p(\overline{T\ x})\ =\ \widetilde{\phi} \\
method & ::= & \tilde{T}\ m(\overline{T\ x})\ contract\ \{\ s\ \} \\
contract & ::= & \texttt{//@requires}\ \widetilde{\phi};\ \texttt{//@ensures}\ \widetilde{\phi}; \\
T & ::= & \texttt{struct}\ S \mid \texttt{int} \mid \texttt{bool} \mid \texttt{char} \\
 & & \mid T* \\
\tilde{T} & ::= & \texttt{void} \mid T \\
s & ::= & s;\ s \mid T\ x \mid T\ x\ =\ e \mid x\ =\ e \\
 & & \mid l\ =\ e \mid e \mid \texttt{assert}(e) \\
 & & \mid \texttt{//@assert}\ \phi \mid \texttt{//@fold}\ p(\overline{\tilde{e}}) \\
 & & \mid \texttt{//@unfold}\ p(\overline{\tilde{e}}) \\
 & & \mid \texttt{if}\ (e)\ \{\ s\ \}\ \texttt{else}\ \{\ s\ \} \\
 & & \mid \texttt{while}\ (e)\ \texttt{//@loop\_invariant}\ \widetilde{\phi}\ \{\ s\ \} \\
 & & \mid \texttt{for}\ (s;\ e;\ s)\ \texttt{//@loop\_invariant}\ \widetilde{\phi}\ \{\ s\ \} \\
e & ::= & v \mid x \mid op(\overline{e}) \mid e\texttt{->}f \mid *e \\
 & & \mid m(\overline{e}) \mid \texttt{alloc}(T) \mid e\ \texttt{?}\ e\ :\ e \\
\tilde{e} & ::= & v \mid x \mid op(\overline{\tilde{e}}) \mid \tilde{e}\texttt{->}f \mid *\tilde{e} \\
l & ::= & x\texttt{->}f \mid *x \mid l\texttt{->}f \mid *l \\
x & ::= & \texttt{\textbackslash result} \mid id \\
v & ::= & n \mid c \mid \texttt{NULL} \mid \texttt{true} \mid \texttt{false} \\
\widetilde{\phi} & ::= & \texttt{?}\ \&\&\ \phi \mid \theta \\
\theta & ::= & \text{a formula } \phi \text{ that is self-framed} \\
\phi & ::= & \tilde{e} \mid p(\overline{\tilde{e}}) \mid \texttt{acc}(l) \mid \phi\ \&\&\ \phi \\
 & & \mid \tilde{e}\ \texttt{?}\ \phi\ :\ \phi
\end{array}
$$

Fig. 6. GVC0 abstract syntax.

$$
\begin{array}{rcl}
\texttt{P} & ::= & \overline{field}\ \overline{predicate}\ \overline{method} \\
field & ::= & \texttt{field}\ f : T \\
predicate & ::= & \texttt{predicate}\ p(\overline{x : T})\ \{\ \widetilde{\phi}\ \} \\
method & ::= & \texttt{method}\ m(\overline{x : T})\ \texttt{returns}\ (\overline{y : T}) \\
 & & contract\ \{\ s\ \} \\
contract & ::= & \texttt{requires}\ \widetilde{\phi}\ \texttt{ensures}\ \widetilde{\phi} \\
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{Ref} \\
s & ::= & s;\ s \mid \texttt{var}\ x : T \mid x\ \texttt{:=}\ e \\
 & & \mid x.f\ \texttt{:=}\ e \mid x\ \texttt{:=}\ \texttt{new}(\overline{f}) \\
 & & \mid \overline{x}\ \texttt{:=}\ m(\overline{e}) \mid \texttt{assert}\ \phi \\
 & & \mid \texttt{fold}\ \textbf{acc}(p(\overline{e})) \\
 & & \mid \texttt{unfold}\ \textbf{acc}(p(\overline{e})) \\
 & & \mid \texttt{if}\ (e)\ \{\ s\ \}\ \texttt{else}\ \{\ s\ \} \\
 & & \mid \texttt{while}\ (e)\ \texttt{invariant}\ \widetilde{\phi}\ \{\ s\ \} \\
e & ::= & v \mid x \mid op(\overline{e}) \mid e.f \\
x & ::= & \texttt{result} \mid id \\
v & ::= & n \mid \texttt{null} \mid \texttt{true} \mid \texttt{false} \\
\widetilde{\phi} & ::= & \texttt{?}\ \&\&\ \phi \mid \theta \\
\theta & ::= & \text{a formula } \phi \text{ that is self-framed} \\
\phi & ::= & e \mid \textbf{acc}(p(\overline{e})) \mid \textbf{acc}(e.f) \mid \phi\ \&\&\ \phi \\
 & & \mid e\ \texttt{?}\ \phi\ :\ \phi
\end{array}
$$

Fig. 7. Gradual Viper abstract syntax.

Representation differs slightly in GVC0 vs. Gradual Viper

Functionality in GVC0 that requires non-trivial translation to Gradual Viper

Fig. 8. Abstract syntax comparison for GVC0 and Gradual Viper.

Then, Sections 4.2 and 4.3 detail Gradual Viper's symbolic execution approach and how it produces minimized run-time checks. Section 4.4 focuses on how GVC0 turns run-time checks from Gradual Viper into C0 code for dynamic verification. Finally, Section 4.5 gives an informal statement of soundness for Gradual C0 and points to related work that has formalized and proven Gradual C0's design sound [Zimmerman et al., 2024].

4.1 Translating C0 Source Code to Gradual Viper Source Code for Verification

The C0 language, with its minimal set of language features and its existing support for specifications, serves well as the target language for our implementation. As its name suggests, C0 borrows heavily from C, but its feature set is reduced to better suit its intended purpose as a tool in computer science

education [Arnold, 2010]. It is a memory-safe subset of C in the same sense as Java. C0 forbids casts, pointer arithmetic, and pointers to stack-allocated memory. C0 produces run-time exceptions, which terminates program execution, when null-pointers are dereferenced and arrays are accessed out of bounds. All pointers are created with heap allocation, and de-allocation is handled by a garbage collector.

The abstract syntax for C0 programs supported by Gradual C0 is given in Figures 5 and 6, i.e., GVC0's abstract syntax. GVC0 programs are made of struct and method declarations that largely follow C syntax. What differs from both C and C0 is GVC0's specification language. Methods may specify constraints on their input and output values as side-effect-free gradual formulas $\widetilde{\phi}$, usually in `//@requires` or `//@ensures` clauses in the method header. Loops and abstract predicates contain invariants and bodies, respectively, that are made of gradual formulas. Such formulas $\widetilde{\phi}$ are imprecise formulas ? && ϕ or complete Boolean formulas ϕ (note, in this case, ϕ must be self-framed as inherited from IDF and defined in Section 4.2.1). In this article, we write ? && ϕ with ? on the left for convenience. It is okay to nest ? arbitrarily in a formula; but, this same formula with ? moved out to the left has the same meaning as the one with nested ? (this follows the interpretation of gradual formulas defined in the work by Wise et al. [2020]). A formula ϕ joins Boolean values, Boolean operators, predicate instances, accessibility predicates, and conditionals via the separating conjunction &&. GVC0 programs also contain `//@fold p(`$\tilde{e}$`)` and `//@unfold p(`$\tilde{e}$`)` statements for predicates and `//@assert` ϕ statements for convenience. Note, in GVC0, $\tilde{e}$ is used in program specifications rather than e to ensure specifications are side-effect-free.

To support the gradual verification of many different imperative programming languages, Gradual Viper verifies programs written in its own custom imperative language, which is designed to ease the translation from other imperative languages into it. The Gradual Viper language's abstract syntax is given in Figures 5 and 7. The GVC0 and Gradual Viper languages are roughly 1-to-1, including their specification languages, so translation is mostly straightforward, but there are some differences as highlighted in orange (trivial) and brown/rose gold (nontrivial) in Figure 8. For example, `for` loops in GVC0 are rewritten as `while` loops in Gradual Viper, and `alloc(struct T)` expressions are translated to `new` statements containing `struct T`'s fields. Additionally, GVC0 allows method calls, `alloc`s, and ternaries in arbitrary expressions, while Gradual Viper only allows such constructs in corresponding program statements.[4] Therefore, GVC0 uses fresh temporary variables to version expressions containing the aforementioned constructs into program statements in Gradual Viper. The temporary variables are then used in the original expression in place of the corresponding method call, `alloc`, or ternary. Nested field assignments, such as `x->y->z = a`, are similarly expanded into multiple program statements using temporary variables. Value type pointers in GVC0 are rewritten as pointers to single-value structs that can be easily translated into Gradual Viper syntax. Finally, `assert(`e`)` statements are essentially ignored in Gradual Viper; e is translated into Gradual Viper syntax to verify its heap accesses, but e is not asserted. Instead, the assert is always kept in the original C0 program and is checked exclusively at run time. Figure 9 provides a simple example program written in both the GVC0 language (Figure 9(a)) and Gradual Viper language (Figure 9(b)) for reference.

Note that GVC0 does not support array and string values since gradually verifying any interesting properties about such constructs requires nontrivial extensions to current gradual verification theory. Similarly, the Gradual Viper language, in contrast to the Viper language, does not support the aforementioned constructs and fractional permissions (see Section 7 for more details).

[4]Note, ternaries correspond to if statements.

```
1  struct Account { int balance; };
2  typedef struct Account Account;
3  /*@ predicate geqTo(Account* a1, Account* a2) =
4     ? && a1->balance >= a2->balance &&
5     a2->balance >= 0; @*/
6  /*@ predicate positive(Account* a) =
7     acc(a->balance) && a->balance >= 0; @*/
8
9  Account* withdraw(Account* a1, Account* a2)
10   //@ requires geqTo(a1,a2);
11   //@ ensures ? && positive(a2) &&
12                positive(\result);
13 {
14   //@ unfold geqTo(a1,a2);
15   if (a1 == NULL || a2 == NULL) {
16     return a1;
17   } else {
18     int newB = a1->balance - a2->balance;
19     a1->balance = newB;
20     //@ fold positive(a1);
21     //@ fold positive(a2);
22     return a1;
23   }
24 }
25
```

(a) A simple bank withdraw example written in the Gradual C0 language

```
1  field balance: Int
2
3  predicate geqTo(a1: Ref, a2: Ref)
4    { ? && a1.balance >= a2.balance &&
5    a2.balance >= 0 }
6  predicate positive(a: Ref)
7    { acc(a.balance) && a.balance >= 0 }
8
9  method withdraw(a1: Ref, a2: Ref)
10 returns (res: Ref)
11   requires acc(geqTo(a1,a2))
12   ensures ? && acc(positive(a2)) &&
13           acc(positive(res))
14 {
15   unfold acc(geqTo(a1,a2))
16   if (a1 == null || a2 == null) {
17     res := a1
18   } else {
19     var newB:Int = a1.balance-a2.balance
20     a1.balance := newB
21     fold acc(positive(a1))
22     fold acc(positive(a2))
23     res := a1
24   }
25 }
```

(b) A simple bank withdraw program written in the Gradual Viper language

Program code | Static specification | Imprecise specification

Ln	Imp.	Opt. Heap	Heap	Var Store	Path Condition	Run-time Checks
14-15	No	∅	geqTo(t1,t2)	a1→t1 ; a2→t2 ; res→t3	∅	∅
15-16	Yes	acc(t1.balance,p1) ; acc(t2.balance,p2)	∅	a1→t1 ; a2→t2 ; res→t3	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0	∅
16-17	-	-	-	-	-	-
17-18	-	-	-	-	-	-
18-19	Yes	acc(t1.balance,p1) ; acc(t2.balance,p2)	∅	a1→t1 ; a2→t2 ; res→t3	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0	∅
19-20	Yes	acc(t1.balance,p1) ; acc(t2.balance,p2)	∅	a1→t1 ; a2→t2 ; res→t3 ; newB→t4	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0 ; t4 = p1 - p2	∅
20-21	Yes	∅	acc(t1.balance,p3)	a1→t1 ; a2→t2 ; res→t3 ; newB→t4	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0 ; t4 = p1 - p2 ; p3 = t4	∅
21-22	Yes	∅	positive(t1)	a1→t1 ; a2→t2 ; res→t3 ; newB→t4	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0 ; t4 = p1 - p2 ; p3 = t4	∅
22-23	Yes	∅	positive(t2)	a1→t1 ; a2→t2 ; res→t3 ; newB→t4	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0 ; t4 = p1 - p2 ; p3 = t4	l_{bc}, ¬(a1 = null || a2 = null) ⟶ l_{c1}, acc(a2.balance) ; l_{bc}, ¬(a1 = null || a2 = null) ⟶ l_{c2}, a2.balance >= 0
23-24	Yes	∅	positive(t2)	a1→t1 ; a2→t2 ; res→t3 ; newB→t4	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0 ; t4 = p1 - p2 ; p3 = t4 ; t3 = t1	l_{bc}, ¬(a1 = null || a2 = null) ⟶ l_{c1}, acc(a2.balance) ; l_{bc}, ¬(a1 = null || a2 = null) ⟶ l_{c2}, a2.balance >= 0
24	l_{bc}, ¬(a1 = null || a2 = null) ⟶ l_{c1}, acc(a2.balance) ; l_{bc}, ¬(a1 = null || a2 = null) ⟶ l_{c2}, a2.balance >= 0 ; l_{bc}, ¬(a1 = null || a2 = null) ⟶ l_{c3}, acc(positive(res))					

(c) Contents of the symbolic state at each program point during Gradual Viper's static verification of `withdraw` in Fig. 9b

Fig. 9. A gradually verified, bank withdraw program that is contrived to illustrate how Gradual Viper works.

4.2 Gradual Viper: Symbolic Execution for Gradual Verification

In this section, we describe the design and implementation of Gradual Viper's symbolic execution-based algorithm supporting the static verification of imprecise formulas. Gradual verification of recursive heap data structures was formalized with weakest liberal preconditions [Wise et al., 2020]. All of the static verifiers supporting separation logic or IDF—such as Viper [Müller et al., 2016], VeriFast [Jacobs et al., 2011], JStar [Distefano and Parkinson, 2008], and SmallFoot [Berdine et al., 2006]—reason with symbolic execution. Therefore, this work serves as a first guide for building gradual verifiers from static verifiers using symbolic execution for reasoning. Additionally, unlike in the work by Wise et al. [2020], we use our static reasoning algorithm not just for optimistic static verification but also for soundly reducing the number of run-time checks required during dynamic verification. That is, during a single execution of Gradual Viper a program is statically verified and a set of minimized run-time checks is produced for program points where the algorithm is optimistic during verification due to imprecision.

Before formalizing Gradual Viper's implementation in Section 4.3, we first demonstrate at a high-level with examples how symbolic execution is used to perform optimistic static verification of programs containing recursive heap data structures and how minimized run-time checks are produced during this process. We also point out novel technical challenges faced and solutions developed thanks to relying on symbolic execution both for static verification and minimizing run-time checks.

4.2.1 Optimistic Static Verification in Gradual Viper by Example. The simple program given in Figure 9(b) implements a `withdraw` function (method), which subtracts the balance in one bank account (the subtrahend) from the balance in another account (the minuend) returning the result.[5] Any client program of `withdraw` must ensure the subtrahend's balance is less than or equal to the minuend's balance and that both balances are positive as specified by `withdraw`'s precondition (line 11). Then, `withdraw` will return an account with a positive balance as specified by `withdraw`'s postcondition (lines 12–13). Additionally, `withdraw`'s postcondition ensures the subtrahend's balance remains positive as well. Note, the `withdraw` example is contrived to better illustrate how Gradual Viper works and its interesting aspects.

Well-Formedness of User Written Specifications. Gradual Viper begins static verification by first checking user written specifications, like predicate bodies, preconditions, and postconditions, for well-formedness. That is, user specifications must be self-framed and cannot contain duplicate accessibility predicates or predicates joined by the separating conjunction &&. Self-framing from IDF [Smans et al., 2009] simply means that a formula must contain accessibility predicates for any heap locations accessed in the formula. In gradual verification [Wise et al., 2020], ? can represent these accessibility predicates. For example, in the `withdraw` program `geqTo`'s body (lines 4–5) is self-framed, because ? can represent **`acc`**`(a1.balance)` and **`acc`**`(a2.balance)` to frame `a1.balance` and `a2.balance`. On the other hand, `positive`'s body (line 7) is classically self-framed as it explicitly contains **`acc`**`(a.balance)` to frame `a.balance`. All of the user written formulas, which are `geqTo`'s body (lines 4–5), `positive`'s body (line 7), `withdraw`'s precondition (line 11), and its postcondition (lines 12–13), are well-formed. Note, static tools implementing IDF or separation logic interpret duplicate accessibility predicates or predicates joined by the separating conjunction in user written specifications as a `false` specification rather than as a well-formedness error like we do. This additional well-formedness check allows Gradual Viper to verify assertions of user written formulas

[5]Note, we refer to the version of the withdraw program written in the Gradual Viper language rather than its Gradual C0 counterpart in Figure 9(a) as we will be discussing how Gradual Viper works in this section.

without interpreting an imprecise state that doesn't already contain `false` as `false`, which violates the interpretation of imprecise formulas (states) in the work by Wise et al. [2020].

Next, Gradual Viper optimistically statically verifies each function in the given program, e.g., the `withdraw` function in our running example. This involves symbolically executing the function from top to bottom and tracking information in a symbolic state. Information is gathered from the execution of both specifications and code, and proof obligations are established by the symbolic state. If any obligations are established optimistically, corresponding run-time checks are stored in the symbolic state. Figure 9(c) displays the contents of the symbolic state at every program point (marked by program lines) during the verification of `withdraw`. In general, a symbolic state can be thought of as writing an intermediate logical formula in a special form. We will discuss the contents of a symbolic state in more detail as we work through the `withdraw` example.

Producing a Precondition. At the start of `withdraw` (lines 14–15), information in the precondition, e.g., `geqTo(a1,a2)` is *produced* or translated into an empty symbolic state resulting in the first state in the table in Figure 9(c). As with formulas, symbolic states may be imprecise or not, meaning information may be missing from the state due to imprecision. In fact, you can think of an imprecise symbolic state as representing an imprecise intermediate formula. Here, the precondition `geqTo(a1,a2)` is precise, so the state remains precise. Had the precondition been imprecise, then the state would become imprecise. Note, precision in a static context as in Gradual Viper means the formula does not contain ? at the top-level. Predicates are treated as black-boxes, so even if their bodies are imprecise, as with `geqTo(a1,a2)`, a formula containing them, such as `geqTo(a1,a2)`, can be precise. This follows the iso-recursive interpretation of precision defined in the work by Wise et al. [2020] for the static side of gradual verification.[6] In contrast, GVC0 utilizes an equi-recursive interpretation of precision for dynamic verification (as inspired by Wise et al. [2020] and discussed in Section 4.4.3). Local variables are mapped to symbolic values in a symbolic, variable store. Since `a1` and `a2` are arguments to withdraw and `res` the return value, they are all assigned fresh symbolic values `t1`, `t2`, and `t3`, respectively, in the store. Then, permissions like accessibility predicates and predicates can be stored in a symbolic heap (either the optimistic heap or heap) in terms of the symbolic values. We call symbolic versions of permissions *heap chunks*. Since `geqTo(a1,a2)` is concretely known it is added directly to the heap as the heap chunk `geqTo(t1,t2)`. An important invariant of the heap is that permissions in it are guaranteed to be separated in memory, i.e., when they are joined by the separating conjunction they return true. The optimistic heap contains heap chunks for accessibility predicates that are optimistically assumed during verification and is introduced in this work to reduce the number of run-time checks produced by Gradual Viper. We will see how this works as we continue to discuss the `withdraw` example. For now, the optimistic heap is empty. The path condition contains constraints on symbolic values that have been collected on the current verification path. The precondition `geqTo(t1,t2)` only contains permission information, so the path condition is empty. Further, producing a formula into the symbolic state does not introduce any run-time checks, so the set of run-time checks also remains empty.

Unfolding a Predicate. Next, Gradual Viper executes the unfold statement on line 15 causing the predicate `geqTo(a1,a2)` to be *consumed* and then its body to be *produced* into the state on lines 15–16. In general, consuming a formula (1) checks whether the formula is established by the symbolic state, (2) generates minimized run-time checks for the state to establish the formula soundly, and (3) removes permissions asserted in the formula from the symbolic state. That is, consume is Gradual Viper's mechanism for checking proof obligations, and as we will see, is used a few different times throughout the verification of `withdraw`. Here, since `geqTo(t1,t2)`

[6]We deviate from this interpretation in some edge cases as described in Section 4.3.7.

is in the heap, `geqTo(a1,a2)` is established by the symbolic state and no run-time checks are required. It is then removed from the heap as it is "consumed." After consumption, `geqTo(a1,a2)`'s body (lines 4–5) is produced into the current state (the state without `geqTo(t1,t2)`). The body of `geqTo` is imprecise, so the symbolic state is made imprecise (as seen in Figure 9(c) at lines 15–16). The rest of `geqTo`'s body is a Boolean expression constraining `a1` and `a2`'s account balances: `a1`'s balance is greater than or equal to `a2`'s balance and `a2`'s balance is positive. Before adding these constraints to the path condition, Gradual Viper first looks for heap chunks in the current symbolic state corresponding to accessibility predicates that frame `a1.balance` and `a2.balance` in `geqTo`'s body. Both the heap and optimistic heap are empty, but the state is imprecise so the missing heap chunks are optimistically assumed to be in the state. In fact, it is sound to make this assumption without any run-time checks, because we are producing (rather than consuming) the predicate body. As a result, fresh symbolic values `p1` and `p2` for `a1.balance` and `a2.balance`, respectively, are generated and used to record constraints on the balances in the path condition. Gradual Viper also records that the receivers `a1` and `a2` are non-null in the path condition and adds heap chunks **`acc`**`(t1,balance,p1)` and **`acc`**`(t2,balance,p2)` to the optimistic heap. Note, these heap chunks are added to the optimistic heap rather than the heap, because `getTo`'s body does not specify whether or not `a1` (`t1`) or `a2` (`t2`) alias. So adding them to the heap would break the heap's invariant. The final symbolic state after consuming `geqTo(a1,a2)` and producing its body is given in Figure 9(c), lines 15–16.

Branching. After the unfold on line 15, Gradual Viper reaches the start of the if statement on the following line 16. As is common with static verifiers based on symbolic execution, Gradual Viper's execution branches at if statements. Execution also branches at other conditioned points, such as logical conditionals and loops. Gradual Viper analyzes the *then* branch (lines 16–18) under the assumption the condition `a1 == null || a2 == null` is true, and the *else* branch (lines 18–24) under the assumption `a1 == null || a2 == null` is false. These assumptions are added to the path condition for each execution path, respectively. However, in our example the symbolic state going into the if statement (Figure 9(c), lines 15–16) states that both `a1` and `a2` are non-null. So the *then* branch is infeasible, and Gradual Viper prunes this execution path resulting in the blank symbolic states in Figure 9(c) from lines 16–18. Both accounts being non-null means the *else* branch condition for sure holds and so execution proceeds down this branch without any changes to the symbolic state. That is, for `withdraw` to be statically verified, this one execution path must successfully verify. If Gradual Viper execute both branches, then determining verification success is a bit more complicated:

- —If the current symbolic state is precise, then both execution paths must successfully verify. This is the default functionality in static verifiers.
- —If the current symbolic state is imprecise, then verification succeeds when one or both paths successfully verify. When only one path succeeds and the state is imprecise, Gradual Viper optimistically assumes the state contains information that forces program execution down the success path only at run time. To ensure the program will never actually execute the failing branch (i.e., to ensure soundness), Gradual Viper adds a run-time check for the success path's condition at the branch point. For example, consider `if (x > 2) //@ assert false; else //@ assert true;` where the symbolic state is imprecise but otherwise empty prior to statically verifying this if statement. Gradual Viper first splits execution and verifies both the `x > 2` branch and `x <= 2` branch; where, the former branch fails due to asserting `false`,[7] while

[7]Note, an imprecise state cannot be assumed to contain information that makes a non-false state false, otherwise Gradual Viper will statically verify all programs trivially [Wise et al., 2020].

the latter succeeds due to asserting `true`. Since one of the execution paths fails, a normal static verifier would report verification of the if statement a failure. However, Gradual Viper's static verifier optimistically assumes ? contains `x <= 2` and the failing branch is unreachable code. So, Gradual Viper successfully verifies the if statement and produces a run-time check for `x <= 2` before the if. This more permissive static functionality is critical for adhering to the gradual guarantee at branch points.

Variable Assignment. Let's look now at how Gradual Viper verifies the `else` branch (lines 18–24). At the variable assignment on line 19, Gradual Viper first evaluates the right-hand expression to the symbolic value `p1 - p2`. To do this, Gradual Viper first looks for heap chunks for `a1.balance` and `a2.balance` in the current symbolic heaps (Figure 9(c), lines 18–19) to both frame the locations and get their values. Both heap chunks are in the optimistic heap, so no run-time checks are required for framing and `p1` and `p2` are used in the evaluation of the right-hand expression. Note, if Gradual Viper did not add the aforementioned heap chunks to the optimistic heap when producing the body of `geqTo(a1,a2)`, then Gradual Viper would create run-time checks for them here in the program. However, these checks would be duplicates, because Gradual Viper also checks that these heap chunks are available when ensuring the precondition `getTo(a1,a2)` holds in client contexts at calls to `withdraw`. So sound tracking of heap chunks in an optimistic heap has helped us avoid duplicating run-time checks! Finally, Gradual Viper adds a new mapping to the variable store for `newB` and its fresh symbolic value `t4`; and then, adds the constraint `t4 = p1 - p2` to the path condition to record information from the assignment in the symbolic state (Figure 9(c), lines 19–20).

Field Assignment. Next, `a1.balance` is assigned `newB`'s value in the field assignment on line 20. Gradual Viper mimics this behavior symbolically by first pulling `newB`'s value `t4` from the symbolic state (Figure 9(c), lines 19–20). Then, Gradual Viper looks for `a1.balance`'s heap chunk in the state for framing, and if there, removes the chunk as `a1.balance`'s value may change in the write, i.e., **`acc`**`(a1.balance)` is *consumed*. Gradual Viper also asserts that `a1` is non-null. The heap chunk for `a1.balance` is in the optimistic heap and `a1 != NULL` is in the path condition, so no run-time checks are needed here. Then, `a1.balance`'s heap chunk is removed from the state; and, unfortunately, this action causes `a2.balance`'s heap chunk to be removed from the state as well. Gradual Viper does not know whether or not `a1` and `a2` alias, because this information does not appear in the current path condition and the optimistic heap does not maintain the separation invariant. Then, since the state is imprecise Gradual Viper could assume that `a1.balance` and `a2.balance` refer to the same heap location, i.e., `a1` and `a2` are aliased. In this case, removing `a1.balance`'s heap chunk requires also removing `a2.balance`'s heap chunk as `a2.balance` may have also changed with the write. On the other hand, the case where they do not alias and `a2.balance`'s heap chunk can stay in the optimistic heap is also possible from Gradual Viper's perspective. To simply and soundly cover both cases Gradual Viper removes `a2.balance`'s heap chunk by default when alias information is unknown. As we will see next, this comes at the cost of additional run-time checks later in the verification of `withdraw`. Finally, Gradual Viper *produces* a new heap chunk for `a1.balance` into the state to track its new, fresh symbolic value `p3` after the write and updates the path condition with the assignment information `p3 = t4` (Figure 9(c), lines 20–21).

Folding a Predicate. After `a1.balance` is assigned a new balance, Gradual Viper executes the fold statement on line 21. Folding a predicate is similar to unfolding a predicate except that the functionality is reversed: `positive(a1)`'s body is *consumed* from the current state (Figure 9(c), lines 20–21) and then `positive(a1)` is *produced* into the state after consumption (Figure 9(c), lines 21–22). The body of `positive(a1)` is **`acc`**`(a1.balance)` && `a1.balance >= 0`, so **`acc`**`(a1.balance)` is consumed first then `a1.balance >= 0` second. The heap chunk **`acc`**`(t1,balance,p3)` corresponding

to **acc**(a1.balance) is in the heap and a1 != null holds in the path condition, so no run-time check is required for consuming **acc**(a1.balance). Then, a1.balance's heap chunk is removed from the heap as seen in Figure 9(c), lines 21–22. Next, the Boolean expression a1.balance >= 0 is evaluated to its symbolic value p3 >= 0. Recall that to do this, Gradual Viper must look up a heap chunk for a1.balance in the symbolic state to both frame the heap location and get its value. However, Gradual Viper just removed this heap chunk from the current state due to the left-to-right execution of consume. To solve this issue, Gradual Viper looks for framing and value information in the state before the fold, i.e., the state before consumption at lines 20–21 in Figure 9(c). As we know, a1.balance's heap chunk is in this state and maps a1.balance to the value p3, so no run-time check is needed for framing. Then, p3 >= 0 is asserted against the current path condition. Since p1 >= p2 >= 0 and p3 = t4 = p1 - p2 are in the path condition, p3 is clearly greater than or equal to 0 and is proven directly by the path condition. That is, no run-time check is needed for p3 >= 0. Finally, positive(a1) is produced into the current state, which adds positive(t1) to the heap resulting in the final version of the state in Figure 9(c), lines 21–22.

Next, Gradual Viper executes the second fold statement on line 22 with the aforementioned state. This fold statement consumes positive(a2)'s body and then produces positive(a2) into the state. So as before, Gradual Viper first consumes **acc**(a2.balance) and then a2.balance >= 0. The receiver a2 is proved to be non-null by the path condition; however this time, the heap chunk for **acc**(a2.balance) is not in either of the heaps. Fortunately for us, the state is imprecise and can optimistically contain this heap chunk, so a run-time check is produced for **acc**(a2.balance) as seen in the state after folding positive(a2) in Figure 9(c), lines 22–23. Recall, "run-time checks" produced by Gradual Viper are descriptions of required run-time checks, not something that can be executed at run time. To see how Gradual C0's frontend GVC0 encodes these descriptions as executables refer to Section 4.4. Note, since this Gradual Viper run-time check occurs down the else branch of the if statement in withdraw, branch information, e.g., l_{bc}, ¬(a1 = null || a2 = null), is included with the check. The location l_{bc} specifies where the branch point originated in the program, e.g., line 16, and ¬(a1 = null || a2 = null) is the assumption made at the branch point for the current execution path. Additionally, l_{c1} contains the location where the check itself is required in the program, e.g., line 22. While it does not happen in the withdraw example, sometimes different checks are required at the same program point down different execution paths. So, Gradual Viper attaches branch information for the entire execution path in a function to each run-time check to allow GVC0 (or other frontends) to apply checks only on the execution path they are required. This prevents Gradual C0 from running checks that do not need to run, which may cause the tool to erroneously fail and produce false positives. Now, Gradual Viper removes **acc**(a2.balance) from the current state (Figure 9(c), lines 21–22), which actually causes positive(t1) to be removed from the heap as well. Predicates are treated as black boxes in Gradual Viper; so unless told otherwise, Gradual Viper conservatively assumes **acc**(a2.balance) is in positive(t1) and removes positive(t1) from the heap alongside **acc**(a2.balance). The only way Gradual Viper can guarantee **acc**(a2.balance) is not in positive(t1) is if a heap chunk for a2.balance and positive(t1) both exist in the heap, as the heap maintains the separation invariant. In this case, positive(t1) can remain in the heap while only **acc**(a2.balance) is removed. Of course, in our example the heap chunk for a2.balance is definitely not in the heap, so positive(t1) is removed.

Continuing, Gradual Viper consumes a2.balance >= 0, which first looks for a heap hunk to frame a2.balance in the state before the consume (Figure 9(c), lines 21–22). However, neither of the heaps contain a heap chunk for a2.balance. As before, Gradual Viper uses imprecision to optimistically assume the heap chunk is in the state and produces a run-time check for **acc**(a2.balance). Since this run-time check for the same location already exists in the state, the two checks are condensed into the first one. Then, Gradual Viper returns a fresh symbolic value for a2.balance,

say p4, to evaluate a2.balance >= 0 down to p4 >= 0. Note, Gradual Viper can only return a fresh value here, because the heaps do not contain a heap chunk recording a2.balance's value in the state. Unfortunately, this means Gradual Viper cannot prove a2.balance >= 0 holds as no constraints exist for p4 in the path condition. But, this also means p4 >= 0 does not contradict existing information in the path condition. So, imprecision in the state can optimistically represent p4 >= 0 and a run-time check for a2.balance >= 0 is generated as seen in Figure 9(c), lines 22–23. A few things of note here:

- —Run-time checks are originally computed in terms of symbolic values, e.g., p4 >= 0, but are ultimately replaced with counterparts written in terms of program variables, e.g., a2.balance >= 0. This replacement by the *translate* function in Gradual Viper simplifies the implementation of run-time checks for frontends like GVC0, which operate on program variables and concrete values not symbolic values. The translate function uses mappings in the symbolic heaps and store to reverse the symbolic execution. Special considerations are made for fresh symbolic values like p4, aliasing between object values, and different variable contexts.
- —On another note, if consuming **acc**(a1.balance) at the field assignment on line 20 did not also consume the heap chunk for a2.balance, then the run-time checks for **acc**(a2.balance) and a2.balance >= 0 would not be necessary. Gradual Viper conservatively assumed a1 and a2 were aliased at the consume, so it removed both chunks from the state. However, in practice a1 and a2 are likely to be distinct objects; and in fact, folding positive(a1) then positive(a2) is a good sign the developer of withdraw expects a1 and a2 to be distinct. In this case, a2.balance's heap chunk does not need to be removed making the aforementioned run-time checks unnecessary. Unfortunately, since we designed Gradual Viper to be conservative, these run-time checks are only eliminated when the developer explicitly specifies that a1 and a2 are not aliased, such as in the precondition of withdraw. Future work should explore ways in which Gradual Viper can be less conservative at consumes.
- —While it does not happen here in our withdraw example, there may be times where parts of a symbolic, Boolean expression are proven statically and the rest optimistically. In this case, Gradual Viper rewrites the expression into **Conjunctive Normal Form (CNF)** and computes the conjuncts in this form that cannot be proven statically by the path condition. These conjuncts (after translation) will then be checked at run time. We call this process computing the *difference* between the expression and the path condition, and it results in minimized run-time checks given statically available information.

Finally, a heap chunk for positive(a2) (e.g., positive(t2)) is produced into the heap resulting in the final form of the state in Figure 9(c), lines 22–23.

Return Value Assignment. Then, Gradual Viper reaches the variable assignment on line 23, which assigns a1 to res—the return value of withdraw. Gradual Viper first looks up the symbolic value t1 for a1 and then the symbolic value t3 for res in the variable store. Gradual Viper stores the information t3 = t1 from the assignment in the path condition resulting in the next symbolic state in Figure 9(c), lines 23–24.

Consuming a Postcondition. Finally, Gradual Viper reaches the end of withdraw down its one and only execution path on line 24. So the last thing Gradual Viper must do to verify the function is to *consume* the postcondition ? && **acc**(positive(a2)) && **acc**(positive(res)) (lines 12–13) in the current symbolic state (Figure 9(c), lines 23–24). Gradual Viper begins by first consuming positive(a2) then positive(res). The heap chunk for positive(a2) is in the heap, so no run-time check is needed for it. Then positive(t2) is removed from the heap leaving both symbolic heaps empty. As a result (and because the state is imprecise), consuming positive(res) in the next step results in

a run-time check for the predicate as seen in the final set of run-time checks required for `withdraw` given in Figure 9(c), line 24. Note, consuming **acc**(`positive(a2)`) && **acc**(`positive(res)`) requires both consuming the predicates individually (which we've done) and ensuring that one predicate does not access heap locations overlapping with the other (in adherence with the separating conjunction &&). Unfortunately, the state does not contain enough information to prove this fact statically, e.g., only the heap chunk for `positive(a2)` appears in the heap, but the state is imprecise! So when Gradual Viper optimistically assumes `positive(res)` holds, it also assumes `positive(res)` is separated from `positive(a2)`. Gradual Viper flags `positive(res)`'s run-time check with this additional check for GVC0 to handle. Additionally, after consuming the static part of an imprecise formula, e.g., **acc**(`positive(a2)`) && **acc**(`positive(res)`) in `withdraw`'s postcondition, Gradual Viper makes the state imprecise and empties both symbolic heaps. The ? in the imprecise formula can represent any permission available in the state, so they must be removed by *consume*.

Takeaways. To summarize, Gradual Viper statically verifies the `withdraw` function successfully, and produces run-time checks for **acc**(`a2.balance`) before line 20, `a2.balance >= 0` also before line 20, and `positive(res)` at the end of `withdraw` (line 24). The `withdraw` function will be completely verified if these checks succeed at run time. During our discussion of the `withdraw` function, we highlighted a number of technical challenges addressed and solutions developed related to designing and implementing Gradual Viper. One of our goals was for Gradual Viper to minimize run-time checks with statically available information. For this we introduced the optimistic heap, which tracks heap chunks that are optimistically assumed during static verification and can be soundly used to reduce run-time checking in successive program statements from where they originated. In `withdraw`, we saw the heap chunks for `a1.balance` and `a2.balance`, which were added to the optimistic heap during the production of `geqTo(a1,a2)`'s body (line 15), be used to eliminate duplicate run-time checks at the assignment on line 19. We had to make careful considerations for the separating conjunction and removal of heap chunks at consumes to ensure sound tracking of heap chunks in the optimistic heap. We also defined and implemented the `diff` function, which utilizes CNF to optimize run-time checks for Boolean expressions. Finally, Gradual Viper conservatively removes heap chunks from the symbolic heaps that may alias with other heap chunks removed at a *consume*. We saw in `withdraw` that this comes at the cost of additional run-time checks: Consuming `a1.balance`'s heap chunk at the field assignment on line 20 also consumed `a2.balance`'s heap chunk resulting in run-time checks for **acc**(`a2.balance`) and `a2.balance >= 0` before line 20. Our strategy is a sound basis for which future work can explore further optimizations.

Another goal for Gradual Viper is for it to use symbolic execution for static reasoning. We accomplished this goal, but not without dealing with some technical challenges. Symbolic execution-based static verifiers generate and discharge proof obligations written in terms of symbolic values, causing Gradual Viper, which extends this system, to follow suit. As a result, Gradual Viper naturally generates run-time checks written in terms of symbolic values as well. Unfortunately, dynamic verifiers only operate on program variables and concrete values not symbolic ones. To bridge this gap between the static and dynamic systems, we implemented a `translate` function in Gradual Viper that rewrites run-time checks containing symbolic values to ones containing program variables and concrete values while being careful about aliases. Finally, execution splitting at branch points led to some trickiness in gradual verification. Different run-time checks may appear at the same program point along different execution paths, so we augmented Gradual Viper to attach branching information to run-time checks. We also augmented Gradual Viper to be more optimistic about verification success when dealing with failing execution paths in the presence of imprecision. This was done in compliance with the gradual guarantee [Wise et al., 2020].

4.3 Gradual Viper: Implemented Algorithm

In this section, we formalize the symbolic execution algorithm implemented by Gradual Viper. A high-level description of how it works is given in Section 4.2. Our algorithm extends Viper's symbolic execution algorithm, and so Gradual Viper's design is heavily influenced by the work of Müller et al. [2016]. Like Viper, Gradual Viper's algorithm consists of four major functions: eval, produce, consume, and exec. The functions evaluate expressions, produce (inhale) and consume (exhale) formulas, and execute program statements, respectively. Following Viper's lead, our four functions are defined in continuation-passing style, where the last argument of each of the aforementioned functions is a continuation Q. The continuation is a function that represents the remaining symbolic execution that still needs to be performed. Note that the last continuation returns a Boolean (λ_.`success()` or λ_.`failure()`), indicating whether or not symbolic execution was successful.

The rest of this section is outlined as follows. Run-time checks and the collections that hold them are described in Section 4.3.1. We described symbolic states in Section 4.3.2 and preliminaries in Section 4.3.3. The definitions for terms and types for functions described in Sections 4.3.1 to 4.3.3 are given collectively in Figure 10. Finally, the four major functions of our algorithm are given in their own sections: eval (Section 4.3.4), produce (Section 4.3.5), consume (Section 4.3.6), and exec (Section 4.3.7). Throughout this section, we make clear where Viper has been extended to support imprecise formulas with yellow highlighting in figures. We also use blue highlighting to indicate extensions for run-time check generation and collection.

4.3.1 Run-time Checks. Run-time checks produced by Gradual Viper are collected in the $\mathfrak{R}$ set. A run-time check is a 4-tuple ($\texttt{bcs}_c, \texttt{origin}_c, \texttt{location}_c, \phi_c$), where $\texttt{bcs}_c$ is a set of branch conditions, $\texttt{origin}_c$ and $\texttt{location}_c$ denote where the run-time check is required in the program, and ϕ_c is what must be checked. A branch condition in $\texttt{bcs}_c$ is also a tuple of ($\texttt{origin}_e, \texttt{location}_e, e$), where $\texttt{origin}_e$ and $\texttt{location}_e$ define the program location at which Gradual Viper's execution branches on the condition e. A `location` is the AST element in the program where the branch or check occurs, denoted as a formula ϕ_l. Sometimes, the condition being checked is defined elsewhere in the program (e.g., in the precondition of a method) but we need to relate it to the method being verified. The `origin` is used to do this. It is `none` when the condition is in the method being verified; otherwise, it contains a method call, fold, unfold, or special loop statement from the method being verified that referenced the check specified in the `location`. An example run-time check is: $(\{(\texttt{none}, x > 2, \neg(x > 2))\}, z := m(y), \textbf{acc}(y.f), \textbf{acc}(y.f))$. The check is for accessing $y.f$, and it is required for m's precondition element $\textbf{acc}(y.f)$ at the method call statement $z := m(y)$. The check is only required when $\neg(x > 2)$, which is evaluated at the program point where the AST element $x > 2$ exists. Since $\neg(x > 2)$'s `origin` is `none`, it comes from an if or assert statement.

Further, $\mathcal{R}$ is used to collect run-time checks down a particular execution path in Gradual Viper. $\mathcal{R}$ is a 3-tuple ($\texttt{bcs}_p, \texttt{origin}_p, \texttt{rcs}_p$) where $\texttt{bcs}_p$ is the set of branch conditions collected down the execution path p, $\texttt{origin}_p$ is the current origin that is set and reset during execution, and `rcs` is the set of run-time checks collected down p. Two auxiliary functions are used to modify $\mathcal{R}$: addcheck and addbc. The addcheck function takes an $\mathcal{R}$ collection $\mathcal{R}_{arg}$, a `location` ϕ_l for a check, and the check itself, and returns a copy of $\mathcal{R}_{arg}$ with the run-time check added to $\mathcal{R}_{arg}$.`rcs`. If necessary, addcheck uses $\mathcal{R}_{arg}$.`origin` and substitution to ensure ϕ_l and the check refer to the correct context. For example, let ϕ_l and check ϕ_c come from asserting a precondition for $z := m(y)$. Then, addcheck performs the substitutions: $\phi_l[t \mapsto m_{arg}]$ (precondition declaration context) and $\phi_c[t \mapsto y]$ (method call context) where t is the symbolic value for y. The addbc function operates similarly to addcheck but for branch conditions.

Run-time checks:

$$\texttt{origin} \in ORIGIN = \{\texttt{none}\} \cup ORIGINS \qquad \textit{(origins)}$$

$$(\texttt{origin}_e, \texttt{location}_e, e) \in \texttt{bcs} \in BCS = \mathcal{P}(ORIGIN \times FORMULA \times EXPR) \qquad \textit{(branch conditions)}$$

$$(\texttt{bcs}_c, \texttt{origin}_c, \texttt{location}_c, \phi_c) \in \mathfrak{R} \in RCHECKS \qquad \textit{(global run-time checks)}$$
$$= \mathcal{P}(BCS \times ORIGIN \times FORMULA \times FORMULA)$$

$$(\texttt{bcs}_p, \texttt{origin}_p, \texttt{rcs}_p) \in \mathcal{R} \in PRCHECKS = \mathcal{P}(BCS \times ORIGIN \times RCHECKS) \qquad \textit{(path run-time checks)}$$

$$\texttt{addcheck}(\mathcal{R}_{arg},\ \texttt{location},\ \phi) : PRCHECKS \longrightarrow FORMULA \longrightarrow FORMULA \longrightarrow PRCHECKS$$

$$\texttt{addbc}(\mathcal{R}_{arg},\ \texttt{location},\ e) : PRCHECKS \longrightarrow FORMULA \longrightarrow EXPR \longrightarrow PRCHECKS$$

Symbolic States:

$$\delta \in SNAPSHOT \qquad \textit{(snapshots)}$$

$$r, \overline{arg}, bc \in SVALUE \qquad \textit{(symbolic values)}$$

$$id(r; \delta) \in SFIELD \qquad \textit{(field chunks)}$$

$$id(\overline{arg}; \delta) \in SPREDICATE \qquad \textit{(predicate chunks)}$$

$$h_?, h \in HEAP = \mathcal{P}(SFIELD \cup SPREDICATE) \qquad \textit{(symbolic heaps)}$$

$$\gamma \in \Gamma = VAR \longrightarrow SVALUE \qquad \textit{(symbolic variable stores)}$$

$$(id, bc, pcs) \in \pi \in \Pi \qquad \textit{(path condition stacks)}$$

$$(\texttt{isImprecise}, h_?, h, \gamma, \pi, \mathcal{R}) = \sigma \in \Sigma \qquad \textit{(symbolic states)}$$
$$= \mathcal{P}(\{\texttt{true}, \texttt{false}\} \times HEAP \times HEAP \times \Gamma \times \Pi \times PRCHECKS)$$

Preliminaries:

$$RESULT = \{\texttt{success}(), \texttt{failure}()\} \qquad \textit{(symbolic execution result)}$$

$$\texttt{pc-add}(\pi,\ t) : \Pi \longrightarrow SVALUE \longrightarrow \Pi$$
$$\texttt{pc-push}(\pi,\ id,\ bc) : \Pi \longrightarrow ID \longrightarrow SVALUE \longrightarrow \Pi$$
$$\texttt{pc-all}(\pi) : \Pi \longrightarrow \mathcal{P}(SVALUE)$$

$$\texttt{unit} : SNAPSHOT$$
$$\texttt{pair}(\delta_1,\ \delta_2) : SNAPSHOT \longrightarrow SNAPSHOT \longrightarrow SNAPSHOT$$
$$\texttt{first}(\delta) : SNAPSHOT \longrightarrow SNAPSHOT$$
$$\texttt{second}(\delta) : SNAPSHOT \longrightarrow SNAPSHOT$$

fresh — *creates fresh snapshots, symbolic values, and other identifiers based on the context*

$$\texttt{havoc}(\gamma,\ \overline{x}) : \Gamma \longrightarrow VAR^* \longrightarrow \Gamma$$
$$\texttt{check}(\pi,\ t) : \Pi \longrightarrow SVALUE \longrightarrow \{\texttt{true}, \texttt{false}\}$$

Fig. 10. Definitions for terms and types for functions described in Sections 4.3.1 to 4.3.3.

4.3.2 Symbolic State. We use $\sigma \in \Sigma$ to denote a symbolic state, which is a 6-tuple $(\texttt{isImprecise}, h_?, h, \gamma, \pi, \mathcal{R})$ consisting of a Boolean `isImprecise`, a symbolic heap $h_?$, another symbolic heap h, a symbolic store γ, a path condition π, and a collection $\mathcal{R}$ (defined in Section 4.3.1). The Boolean `isImprecise` records whether or not the state is imprecise, the symbolic store γ maps local variables to their symbolic values, the path condition π (defined in Section 4.3.3) contains

constraints on symbolic values that have been collected on the current verification path, and $\mathcal{R}$ contains the run-time checks that have been collected on the current verification path.

A symbolic heap is a multiset of heap chunks for fields or predicates that are currently accessible. A field chunk $id(r;\ \delta)$ (representing expression $r.id$) consists of the field name id, the receiver's symbolic value r, and the field's symbolic value δ—also referred to as the *snapshot* of a heap chunk. For a predicate chunk $id(args;\ \delta)$, id is the predicate name, $args$ is a list of symbolic values that are arguments to the predicate, and δ is the snapshot of the predicate. A predicate's snapshot represents the values of the heap locations abstracted over by the predicate. The symbolic, *optimistic* heap $h_?$ contains heap chunks that are accessible due to optimism in the symbolic execution, while h contains heap chunks that are statically accessible. Further, only h maintains the invariant that its heap chunks are separated in memory, and thus, can be joined successfully by the separating conjunction. The *empty symbolic state* is

$\sigma_0 = (\texttt{isImprecise} := \texttt{false}, h_? := \emptyset, h := \emptyset, \gamma := \emptyset, \pi := \emptyset, \mathcal{R} := (\emptyset, \text{none}, \emptyset))$.

4.3.3 Preliminaries. We introduce a few preliminary definitions here that will be helpful later. A path condition π is a stack of tuples (id, bc, pcs). An id is a unique identifier that determines the constraints on symbolic values that have been collected between two branch points in execution. The bc entry is the symbolic value for the branch condition from the first of two branch points, and pcs is the set of constraints that have been collected. Branch points can be from if statements and logical conditionals in formulas. Functions pc-all, pc-add, and pc-push manipulate path conditions and are formally defined in Appendix, Figure A1. The pc-all function collects and returns all the constraints in π, pc-add adds a new constraint to π, and pc-push adds a new stack entry to π. Similarly, snapshots for heap chunks have their own related functions: *unit*, *pair*, *first*, and *second*. The constant *unit* is the empty snapshot, *pair* constructs pairs of snapshots, and *first* and *second* deconstruct pairs of snapshots into their subparts. Further, fresh is used to create fresh snapshots, symbolic values, and other identifiers depending on the context. The havoc function similarly updates a symbolic store by assigning a fresh symbolic value to each variable in a given collection of variables. Finally, $\texttt{check}(\pi, t) = \texttt{pc-all}(\pi) \Rightarrow t$ queries the underlying SAT solver to see if the given constraint t is valid in a given path condition π (i.e., π proves or implies t).

4.3.4 Symbolic Execution of Expressions. The symbolic execution of expressions by the eval function is defined in Figure 11. Using the current symbolic state, eval evaluates an expression to a symbolic value t and returns t and the current state to the continuation Q. Variable values are looked up in the symbolic store and returned. For $op(\overline{e})$, its arguments $\overline{e}$ are each evaluated to their symbolic values $\overline{t}$. A symbolic value $op'(\overline{t})$ is then created and returned with the state after evaluation. Each op has a corresponding symbolic value op' of the same arity. For example, $e_1 + e_2$ results in the symbolic value $add(t_1, t_2)$ where e_1 and e_2 evaluate to t_1 and t_2, respectively.

Finally, the most interesting rule is for fields $e.f$. The receiver e is first evaluated to t resulting in a new state σ_2. Then, eval looks for a heap chunk for $t.f$ first in the current heap h.[8] If a chunk exists, then the heap read succeeds and σ_2 and the chunk's snapshot δ is returned to the continuation. If a chunk does not exist in h, then eval looks for a chunk in the optimistic heap $h_?$, and if found the chunk's snapshot is returned with σ_2. If a heap chunk for $t.f$ is not found in either heap, then the heap read can still succeed when σ_2 is imprecise. As long as $t \neq \texttt{null}$ does not contradict the current path condition $\sigma_2.\pi$ (the call to assert, Appendix, Figure A8), σ_2's imprecision optimistically provides access to $t.f$. Therefore, a run-time check for $\textbf{acc}(e_t.f)$ is created and added to σ_2's set of run-time checks (highlighted in blue). Note that $e_t.f$ is used in the check rather than $t.f$, because—unlike t which is a symbolic value—the expression e_t can be evaluated at run time. Specifically,

[8]Heap lookup in eval also looks for heap chunks that are aliases (according to the path condition) to the chunk in question.

$$\begin{aligned}
&\mathsf{eval}(\sigma,\ e,\ Q) : \Sigma \longrightarrow \mathit{EXPR} \longrightarrow (\Sigma \longrightarrow \mathit{SVALUE} \longrightarrow \mathit{RESULT}) \longrightarrow \mathit{RESULT}\\
&\mathsf{eval}(\sigma,\ t,\ Q) = Q(\sigma,\ t)\\
&\mathsf{eval}(\sigma,\ x,\ Q) = Q(\sigma,\ \sigma.\gamma(x))\\
&\mathsf{eval}(\sigma_1,\ op(\overline{e}),\ Q) = \mathsf{eval}(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .\ Q(\sigma_2,\ op'(\overline{t}))))\\
&\mathsf{eval}(\sigma_1,\ e.f,\ Q) = \mathsf{eval}(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\\
&\quad \texttt{if}\ (\exists\ f(r;\ \delta) \in \sigma_2.h\ .\ \mathsf{check}(\sigma_2.\pi,\ r = t))\ \texttt{then}\\
&\quad\quad Q(\sigma_2,\ \delta)\\
&\quad \texttt{else if}\ (\exists\ f(r;\ \delta) \in \sigma_2.h_?\ .\ \mathsf{check}(\sigma_2.\pi,\ r = t))\ \texttt{then}\\
&\quad\quad Q(\sigma_2,\ \delta)\\
&\quad \texttt{else if}\ (\sigma_2.\mathsf{isImprecise})\ \texttt{then}\\
&\quad\quad \mathit{res},\ _\ :=\ \mathsf{assert}(\sigma_2.\mathsf{isImprecise},\ \sigma_2.\pi,\ t \neq \mathsf{null})\\
&\quad\quad e_t\ :=\ \mathsf{translate}(\sigma_2,\ t)\\
&\quad\quad \mathcal{R}'\ :=\ \mathsf{addcheck}(\sigma_2.\mathcal{R},\ e.f,\ \mathbf{acc}(e_t.f))\\
&\quad\quad \delta\ :=\ \mathsf{fresh}\\
&\quad\quad \mathit{res} \wedge Q(\sigma_2\{h_?\ :=\ \sigma_2.h_? \cup f(t;\ \delta),\ \pi\ :=\ \mathsf{pc\text{-}add}(\sigma_2.\pi,\ \{t \neq \mathsf{null}\}),\ \mathcal{R}\ :=\ \mathcal{R}'\},\ \delta)\\
&\quad \texttt{else}\ \mathsf{failure}())
\end{aligned}$$

Handles imprecision Handles run-time check generation and collection

Fig. 11. Rules for symbolically executing expressions.

translate (described in Appendix, Figure A5) is called on t with the current state σ_2 to compute e_t. Additionally, the AST element $e.f$ is used to denote the check's location.

Afterwards, a fresh snapshot δ is created for $t.f$'s value, and a heap chunk $f(t;\ \delta)$ for $t.f$ and δ is created and added to σ_2's optimistic heap passed to the continuation. Similarly, the constraint $t \neq \mathsf{null}$ is added to σ_2's path condition. By adding $f(t;\ \delta)$ to the optimistic heap, the following accesses of $t.f$ are statically verified by the optimistic heap, which reduces the number of run-time checks produced. Finally, verification of the heap read for $t.f$ fails when none of the aforementioned cases are true. Figures A2 and A3 in the Appendix define variants of eval, called eval-p and eval-c, that are used in produce and consume, respectively. The eval-p variant does not introduce run-time checks and eval-c does not extend the optimistic heap and path condition, because the aforementioned functionalities are not needed in these contexts.

4.3.5 Symbolic Production of Formulas. Produce (Figure 12) is responsible for adding information to the symbolic state, in particular, the path condition and the heap h. Producing an imprecise formula makes the symbolic state imprecise. The produce rule for an expression e evaluates e to its symbolic value and produces it into the path condition. The produce rules for accessibility predicates containing fields and predicates are similar, so we focus on the rule for fields only. The field $e.f$ in $\mathbf{acc}(e.f)$ first has its receiver e evaluated to a symbolic value t. Then, using the parameter δ a fresh heap chunk $f(t;\ \delta)$ is created and added to the heap before invoking the continuation. Note, the disjoint union $\uplus$ ensures $f(t;\ \delta)$ is not already in the heap before $f(t;\ \delta)$ is added; otherwise, verification fails. Further, $\mathbf{acc}(e.f)$ implies $e \neq \mathsf{null}$ and so that fact is recorded in the path condition as $t \neq \mathsf{null}$. When the separating conjunction ϕ_1 && ϕ_2 is produced, ϕ_1 is first produced into the symbolic state, followed by ϕ_2. Finally, to produce a conditional, Gradual Viper branches on the symbolic value t for the condition e splitting execution along two different paths. Along one path ϕ_1 is produced into the state under the assumption that t is true, and along the other path ϕ_2 is produced under the $\neg t$ assumption. Both paths follow the continuation to the end

$$\begin{aligned}
&\text{produce}(\sigma,\ \widetilde{\phi},\ \delta,\ Q) : \Sigma \longrightarrow \widetilde{FORMULA} \longrightarrow SNAPSHOT \longrightarrow (\Sigma \longrightarrow RESULT) \longrightarrow RESULT \\
&\text{produce}(\sigma,\ ?\,\&\&\,\phi,\ \delta,\ Q) = \text{produce}(\sigma\{\text{isImprecise} := \text{true}\},\ \phi,\ \text{second}(\delta),\ Q) \\
&\text{produce}(\sigma_1,\ e,\ \delta,\ Q) = \text{eval-p}\,(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\ Q(\sigma_2\{\pi := \text{pc-add}(\sigma_2.\pi,\ \{t, \delta = unit\})\}))) \\
&\text{produce}(\sigma_1,\ \textbf{acc}(p(\overline{e})),\ \delta,\ Q) = \text{eval-p}\,(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .\ Q(\sigma_2\{h := \sigma_2.h \uplus p(\overline{t};\ \delta)\}))) \\
&\text{produce}(\sigma_1,\ \textbf{acc}(e.f),\ \delta,\ Q) = \text{eval-p}\,(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ . \\
&\qquad Q(\sigma_2\{h := \sigma_2.h \uplus f(t;\ \delta),\ \pi := \text{pc-add}(\sigma_2.\pi,\ \{t \neq \text{null}\})\}))) \\
&\text{produce}(\sigma_1,\ \phi_1\ \&\&\ \phi_2,\ \delta,\ Q) = \text{produce}(\sigma_1,\ \phi_1,\ \text{first}(\delta),\ (\lambda\ \sigma_2\ .\ \text{produce}(\sigma_2,\ \phi_2,\ \text{second}(\delta),\ Q))) \\
&\text{produce}(\sigma_1,\ e\ ?\ \phi_1\ :\ \phi_2,\ \delta,\ Q) = \text{eval-p}\,(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ . \\
&\qquad \text{branch}\,(\sigma_2,\ e,\ t,\ (\lambda\ \sigma_3\ .\ \text{produce}(\sigma_3,\ \phi_1,\ \delta,\ Q)), \\
&\qquad\qquad (\lambda\ \sigma_3\ .\ \text{produce}(\sigma_3,\ \phi_2,\ \delta,\ Q)))))
\end{aligned}$$

Handles imprecision Handles run-time check generation and collection

Fig. 12. Rules for symbolically producing formulas.

$$\begin{aligned}
&\text{branch}(\sigma,\ e,\ t,\ Q_t,\ Q_{\neg t}) : \Sigma \longrightarrow EXPR \longrightarrow SVALUE \longrightarrow (\Sigma \longrightarrow RESULT) \longrightarrow (\Sigma \longrightarrow RESULT) \longrightarrow RESULT \\
&\text{branch}(\sigma,\ e,\ t,\ Q_t,\ Q_{\neg t}) = \\
&\qquad (\pi_T,\ \mathcal{R}_T)\ :=\ (\text{pc-push}(\sigma.\pi,\ \text{fresh},\ t),\ \text{addbc}(\sigma.\mathcal{R},\ e,\ e)) \\
&\qquad (\pi_F,\ \mathcal{R}_F)\ :=\ (\text{pc-push}(\sigma.\pi,\ \text{fresh},\ \neg t),\ \text{addbc}(\sigma.\mathcal{R},\ e,\ \neg e)) \\
&\qquad \texttt{if } (\sigma.\text{isImprecise}) \texttt{ then} \\
&\qquad\quad res_T\ :=\ (\texttt{if } \neg\text{check}(\sigma.\pi,\ \neg t) \texttt{ then } Q_t(\sigma\{\pi := \pi_T,\ \mathcal{R} := \mathcal{R}_T\}) \texttt{ else failure()}) \\
&\qquad\quad res_F\ :=\ (\texttt{if } \neg\text{check}(\sigma.\pi,\ t) \texttt{ then } Q_{\neg t}(\sigma\{\pi := \pi_F,\ \mathcal{R} := \mathcal{R}_F\}) \texttt{ else failure()}) \\
&\qquad\quad \texttt{if } ((res_T \wedge \neg res_F) \vee (\neg res_T \wedge res_F)) \texttt{ then} \\
&\qquad\qquad \mathcal{R}'\ :=\ \text{addcheck}(\sigma.\mathcal{R},\ e,\ (\texttt{if } (res_T) \texttt{ then } e \texttt{ else } \neg e)) \\
&\qquad\qquad \mathfrak{R}\ :=\ \mathfrak{R} \cup \mathcal{R}'.\text{rcs.last} \\
&\qquad\quad res_T \vee res_F \\
&\qquad \texttt{else} \\
&\qquad\quad (\texttt{if } \neg\text{check}(\sigma.\pi,\ \neg t) \texttt{ then } Q_t(\sigma\{\pi := \pi_T,\ \mathcal{R} := \mathcal{R}_T\}) \texttt{ else success()}) \quad \wedge \\
&\qquad\quad (\texttt{if } \neg\text{check}(\sigma.\pi,\ t) \texttt{ then } Q_{\neg t}(\sigma\{\pi := \pi_F,\ \mathcal{R} := \mathcal{R}_F\}) \texttt{ else success()})
\end{aligned}$$

Handles imprecision Handles run-time check generation and collection

Fig. 13. Formally defining the branch function.

of its execution, and a branch condition corresponding to the t assumption made is added to the symbolic state. Paths are pruned when they are infeasible (the assumption about t would contradict the current path conditions). Overall verification success is computed from the results of the two execution paths, and an imprecise state allows this computation to be optimistic when one path successfully verifies and the other doesn't. In this case, branch optimistically marks verification a success when normally it should fail, because the state may optimistically contain information that prunes the failure case. A run-time check is then added for the success path's condition to ensure soundness. This functionality is important for adhering to the gradual guarantee [Wise et al., 2020]. The formal definition of branch is in Figure 13, and other details for branch and produce are given in Appendix A.2. Note, produce only adds run-time checks for branching to the symbolic state.

$$\text{consume}(\sigma,\ \widetilde{\phi},\ Q) : \Sigma \longrightarrow \widetilde{FORMULA} \longrightarrow (\Sigma \longrightarrow SNAPSHOT \longrightarrow RESULT) \longrightarrow RESULT$$

$$\begin{aligned}
\text{consume}(\sigma_1,\ \theta,\ Q) \quad &= \sigma_2 := \sigma_1\{\ h,\ \pi := \text{consolidate}(\sigma_1.h,\ \sigma_1.\pi)\ \} \\
&\quad \text{consume'}(\sigma_2,\ \sigma_2.\text{isImprecise},\ \sigma_2.h_?,\ \sigma_2.h,\ \theta,\ (\lambda\ \sigma_3,\ h'_?,\ h_1,\ \delta_1\ . \\
&\quad\quad Q(\sigma_3\{\ h_? := h'_?,\ h := h_1\},\ \delta_1))) \\
\text{consume}(\sigma_1,\ ?\ \&\&\ \phi,\ Q) \ &= \sigma_2 := \sigma_1\{\ h,\ \pi := \text{consolidate}(\sigma_1.h,\ \sigma_1.\pi)\ \} \\
&\quad \text{consume'}(\sigma_2,\ \text{true},\ \sigma_2.h_?,\ \sigma_2.h,\ \phi,\ (\lambda\ \sigma_3,\ h'_?,\ h_1,\ \delta_1\ . \\
&\quad\quad Q(\sigma_3\{\ \text{isImprecise} := \text{true},\ h_? := \emptyset,\ h := \emptyset\},\ \textit{pair}(\textit{unit},\ \delta_1)\ )))
\end{aligned}$$

$$\begin{aligned}
&\text{consume'}(\sigma,\ f_?,\ h_?,\ h,\ \phi,\ Q) : \Sigma \longrightarrow \{\text{true}, \text{false}\} \longrightarrow \textit{HEAP} \longrightarrow \textit{HEAP} \longrightarrow \textit{FORMULA} \longrightarrow \\
&\qquad (\Sigma \longrightarrow \textit{HEAP} \longrightarrow \textit{HEAP} \longrightarrow \textit{SNAPSHOT} \longrightarrow \textit{RESULT}) \longrightarrow \textit{RESULT} \\
&\text{consume'}(\sigma,\ f_?,\ h_?,\ h,\ (e, t),\ Q) \quad = \textit{res},\ \overline{t} := \text{assert}(\sigma.\text{isImprecise},\ \sigma.\pi,\ t) \\
&\qquad\qquad \mathcal{R}' := \text{addcheck}(\sigma.\mathcal{R},\ e,\ \text{translate}(\sigma,\ \overline{t})) \\
&\qquad\qquad \textit{res} \wedge Q(\sigma\{\ \mathcal{R} := \mathcal{R}'\},\ h_?,\ h,\ \textit{unit}) \\
&\text{consume'}(\sigma_1,\ f_?,\ h_?,\ h,\ e,\ Q) \quad = \text{eval-c}\,(\sigma_1\{\ \text{isImprecise} := f_?\},\ e,\ (\lambda\ \sigma_2,\ t\ . \\
&\qquad\qquad \text{consume'}(\sigma_2\{\ \text{isImprecise} := \sigma_1.\text{isImprecise}\},\ f_?,\ h_?,\ h,\ (e, t),\ Q))) \\
&\text{consume'}(\sigma_1,\ f_?,\ h_?,\ h,\ \textbf{acc}(e.f),\ Q) = \text{eval-c}\,(\sigma_1\{\ \text{isImprecise} := f_?\},\ e,\ (\lambda\ \sigma_2,\ t\ . \\
&\qquad\qquad \sigma_3 := \sigma_2\{\text{isImprecise} := \sigma_1.\text{isImprecise}\} \\
&\qquad\qquad \textit{res},\ \overline{t} := \text{assert}(\sigma_3.\text{isImprecise},\ \sigma_3.\pi,\ t \neq \text{null}) \\
&\qquad\qquad \textit{res}\ \wedge\ (\\
&\qquad\qquad \mathcal{R}' := \text{addcheck}(\sigma_3.\mathcal{R},\ \textbf{acc}(e.f),\ \text{translate}(\sigma_3,\ \overline{t})) \\
&\qquad\qquad (h_1,\ \delta_1,\ b_1) := \text{heap-rem-acc}(\sigma_3.\text{isImprecise},\ h,\ \sigma_3.\pi,\ f(t)) \\
&\qquad\qquad \text{if } (\sigma_3.\text{isImprecise}) \text{ then} \\
&\qquad\qquad\quad (h'_?,\ \delta_2,\ b_2) := \text{heap-rem-acc}(\sigma_3.\text{isImprecise},\ h_?,\ \sigma_3.\pi,\ f(t)) \\
&\qquad\qquad\quad \text{if } (b_1 = b_2 = \text{false}) \text{ then} \\
&\qquad\qquad\qquad \mathcal{R}'' := \text{addcheck}(\mathcal{R}',\ \textbf{acc}(e.f),\ \textbf{acc}(\text{translate}(\sigma_3,\ t).f)) \\
&\qquad\qquad\quad \text{else } \mathcal{R}'' := \mathcal{R}' \\
&\qquad\qquad\quad Q(\sigma_3\{\ \mathcal{R} := \mathcal{R}''\},\ h'_?,\ h_1,\ (\text{if } (b_1) \text{ then } \delta_1 \text{ else } \delta_2)) \\
&\qquad\qquad \text{else if } (b_1) \text{ then } Q(\sigma_3\{\ \mathcal{R} := \mathcal{R}'\},\ \sigma_3.h_?,\ h_1,\ \delta_1) \\
&\qquad\qquad \text{else failure}())))
\end{aligned}$$

Handles imprecision Handles run-time check generation and collection

Fig. 14. *Select rules* for symbolically consuming formulas.

4.3.6 Symbolic Consumption of Formulas. The goals of consume are 3-fold: (1) given a symbolic state σ and formula $\widetilde{\phi}$, check whether $\widetilde{\phi}$ is established by σ, i.e., $\widetilde{\phi}_\sigma \Rrightarrow \widetilde{\phi}$ where $\widetilde{\phi}_\sigma$ is the formula which represents the state σ, (2) produce and collect run-time checks that are minimally sufficient for σ to establish $\widetilde{\phi}$ soundly, and (3) remove accessibility predicates and predicates that are asserted in $\widetilde{\phi}$ from σ. The rules for consume are given in full and described in great detail in Appendix A.3. We give select rules in Figure 14 and an abstract description here.

The functionality of consume is split across two functions: consume, which is the interface to consume accepting only a state, formula, and continuation, and consume', which is a helper

function performing consume's major functionality. Note, before calling consume', consume first adds non-alias information from the heap to the path condition and checks that the heap and path condition are noncontradictory using consolidate [Schwerhoff, 2016].

Then, the two functions work together to accomplish the aforementioned goals. For the first and second goals, heap chunks representing accessibility predicates and predicates in $\widetilde{\phi}$ are looked up in the heap h and optimistic heap $h_?$ from σ. When σ is precise, the heap chunks must be in h or verification fails. If σ is imprecise, then the heap chunks are always justified either by the heaps or imprecision. Run-time checks for heap chunks that are verified by imprecision are collected in $\sigma.\mathcal{R}$. The consume' rule for $\mathbf{acc}(e.f)$ (and the rule for $\mathbf{acc}(p(\overline{e}))$ which is similar) supports this functionality by calling heap-rem-acc (defined in Appendix, Figure A7) for the look-up, assigning the Boolean results to b_1 and b_2, and then using them in `if-then-else` and `else-if` casing. The blue highlighting in the `isImprecise` is `true` case in the aforementioned rule handles the run-time checks. Clauses in $\widetilde{\phi}$ containing logical expressions are first evaluated to a symbolic value t, which is then checked against σ's path condition π. If σ is precise, then $\texttt{pc-all}(\pi) \Rightarrow t$ must hold (i.e., the constraints in π prove t) or verification fails. In contrast, when σ is imprecise, $\bigwedge \texttt{pc-all}(\pi) \wedge t$ must hold (i.e., t does not contradict constraints in π) otherwise verification fails. In this case, a run-time check is added to $\sigma.\mathcal{R}$ for the set of residual symbolic values in t that cannot be proved statically by π. The consume' rules for expressions and symbolic values implement this behavior. The call to assert (defined in Appendix, Figure A8) checks t against π and returns the result and any residual symbolic values. Note, assert uses diff from Appendix A.1 to compute the residuals. The part highlighted in blue adds the run-time check for the residuals to the state. Finally, fields used in $\widetilde{\phi}$ must have corresponding heap chunks in h when σ and $\widetilde{\phi}$ are precise; otherwise when σ or $\widetilde{\phi}$ are imprecise, field access can be justified by either the heaps or imprecision. A run-time check containing an accessibility predicate for the field is added to $\sigma.\mathcal{R}$ when imprecision is relied on. This is all handled by the second argument $f_?$ to consume' and eval-c called by consume' on expressions.

The third goal of consume is to remove heap chunks $\overline{hc_i}$ representing accessibility predicates and predicates in $\widetilde{\phi}$ from σ, and in particular, from heaps h and $h_?$. When σ and $\widetilde{\phi}$ are both precise, the heap chunks in $\overline{hc_i}$ are each removed from h ($h_?$ is empty here). If $\widetilde{\phi}$ is imprecise, then all heap chunks in both heaps are removed as they may be in $\overline{hc_i}$ or $\widetilde{\phi}$ may represent them with imprecision. Finally, when σ is imprecise and $\widetilde{\phi}$ is precise, any heap chunks in h or $h_?$ that overlap with or may potentially overlap with (thanks to σ's imprecision) heap chunks in $\overline{hc_i}$ are removed. The calls to heap-rem-acc (and its counterpart heap-rem-pred) in consume', the extra heaps tracked in consume', and the heap assignments in the continuations from consume come together to implement heap chunk removal.

4.3.7 Symbolic Execution of Statements. The exec rules in Gradual Viper, which symbolically execute program statements, are largely unchanged from Viper. The only differences are (1) the rules now utilize versions of eval, produce, consume, and branch defined previously in this article and (2) the rules track origins where appropriate. To provide an intuition, select rules for exec are given in Figure 15; the full set of rules are listed in Appendix A.4. The exec function takes a symbolic state σ, program statement $stmt$, and continuation Q. Then, exec symbolically executes $stmt$ using σ to produce a potentially modified state σ', which is passed to the continuation.

Symbolic execution of field assignments first evaluates the right-hand side expression e to the symbolic value t. Any field reads in e are either directly or optimistically verified using σ_1. Then, the resulting state σ_2 must establish write access to $x.f$ in consume, i.e., $\sigma_2 \widetilde{\Rightarrow} \mathbf{acc}(x.f)$. Calling consume also removes the field chunk for $\mathbf{acc}(x.f)$ from σ_2 (if it is in there) resulting in σ_3. Therefore,

$$\text{exec}(\sigma,\ s,\ Q) : \Sigma \longrightarrow \mathit{STMT} \longrightarrow (\Sigma \longrightarrow \mathit{RESULT}) \longrightarrow \mathit{RESULT}$$

$$\begin{aligned}
&\text{exec}(\sigma_1,\ x.f\ :=\ e,\ Q) \ = \ \text{eval}\,(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\ \text{consume}\,(\sigma_2,\ \mathbf{acc}(x.f),\ (\lambda\ \sigma_3,\ _\ .\\
&\qquad \text{produce}\,(\sigma_3,\ \mathbf{acc}(x.f)\ \&\&\ x.f = t,\ \mathit{pair}(\text{fresh},\ \mathit{unit}),\ Q)))))\\
&\text{exec}(\sigma_1,\ \overline{z}\ :=\ m(\overline{e}),\ Q) = \text{eval}\,(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .\\
&\qquad \mathcal{R}'\ :=\ \sigma_2.\mathcal{R}\{\texttt{origin}\ :=\ (\sigma_2,\ \overline{z}\ :=\ m(\overline{e}),\ \overline{t})\}\\
&\qquad \text{consume}\,(\sigma_2\{\ \mathcal{R}\ :=\ \mathcal{R}'\ \},\ \mathit{meth}_{pre}[\overline{\mathit{meth}_{args} \mapsto t}],\ (\lambda\ \sigma_3,\ \delta\ .\\
&\qquad\quad \texttt{if (equi-imp}(\mathit{meth}_{pre})\texttt{) then}\\
&\qquad\qquad \sigma_4\ :=\ \sigma_3\{\texttt{isImprecise := true},\ h_?\ :=\ \emptyset,\ h\ :=\ \emptyset,\ \gamma\ :=\ \text{havoc}(\sigma_3.\gamma,\ \overline{z})\}\\
&\qquad\quad \texttt{else}\ \ \sigma_4\ :=\ \sigma_3\{\gamma\ :=\ \text{havoc}(\sigma_3.\gamma,\ \overline{z})\}\\
&\qquad\quad \text{produce}\,(\sigma_4,\ \mathit{meth}_{post}[\overline{\mathit{meth}_{args} \mapsto t}][\overline{\mathit{meth}_{ret} \mapsto z}],\ \text{fresh},\\
&\qquad\qquad (\lambda\ \sigma_5\ .\ Q(\sigma_5\{\ \mathcal{R}\ :=\ \sigma_5.\mathcal{R}\{\texttt{origin := none}\}\ \}))))))))
\end{aligned}$$

Handles imprecision Handles run-time check generation and collection

Fig. 15. *Select rules* for symbolically executing program statements.

the call to produce can safely add a fresh field chunk for $\mathbf{acc}(x.f)$ alongside $x.f = t$ to σ_3 before it is passed to the continuation Q. Under the hood, run-time checks are collected and passed to Q.

The method call rule evaluates the arguments $\overline{e}$ to symbolic values $\overline{t}$, consumes the method precondition (substituting arguments with $\overline{t}$) while making sure the origin is set properly for check and branch condition insertion, havocs existing assumptions about the variables being assigned to, produces knowledge from the postcondition, and finally continues after resetting the origin to none. An in-depth explanation is in the Appendix, along with the other exec rules and equi-imp definition.

Note that while Gradual Viper treats predicates iso-recursively in all other cases, it makes an exception when consuming preconditions at method calls (and loop invariants before entering loops), which can be seen in the `if-then` in the method call rule (Figure 15). If Gradual Viper determines the precondition (invariant) is equi-recursively imprecise (using `equi-imp`, Appendix, Figure A10), then it will conservatively remove all the heap chunks from both symbolic heaps and make the state imprecise after the consume. This exception ensures the static verification semantics in Gradual Viper lines up with the equi-recursive, dynamic verification semantics encoded by GVC0 (described in Section 4.4) such that Gradual C0 is sound. Interestingly, the gradual verifier of Wise et al. [2020] does not need this special case, because it does not optimize run-time checks with statically available information. Once optimization is introduced, the semantics across the two systems need to be more tightly integrated to ensure soundness. Zimmerman et al. [2024] alerted us to this issue and proposed the aforementioned solution.

4.3.8 Valid Gradual Viper Programs. Finally, putting everything together, a Gradual Viper program is checked by examining each of its method and predicate definitions to ensure they are well-formed (formally defined in Appendix, Figure A11). The formal definitions are given in Figure 16, and a more detailed description of the rules is given in Appendix A.5. Intuitively, for each method, we define symbolic values for the method arguments, and then create an initial symbolic state by calling the produce function on the method precondition.[9] We then call the exec function on the method body, which symbolically executes the body and ensures that all operations are valid based on that precondition. Finally, we invoke the consume function on the final symbolic state and the postcondition, verifying that the former implies the latter. Throughout these operations a

[9]Note, produce is part of well-formed.

$$\text{verify}(decl) : MDECL \cup PDECL \longrightarrow RESULT$$

$$\text{verify}(\texttt{method}\ \ m(\overline{x : T})\ \texttt{returns}\ (\overline{y : T})) =$$
$$\texttt{well-formed}\,(\sigma_0\{\gamma\ :=\ \sigma_0.\gamma[\overline{x \mapsto \text{fresh}}][\overline{y \mapsto \text{fresh}}]\},\ meth_{pre},\ \text{fresh},\ (\lambda\ \sigma_1\ .$$
$$\texttt{well-formed}\,(\sigma_1\{\ \texttt{isImprecise := false},\ h_? := \emptyset\ \},\ h\ :=\ \emptyset\},\ meth_{post},$$
$$\text{fresh},\ (\lambda\ \_\ .\ \text{success}()))$$
$$\wedge$$
$$\texttt{exec}\,(\sigma_1,\ meth_{body},\ (\lambda\ \sigma_2\ .$$
$$\texttt{consume}\,(\sigma_2,\ meth_{post},\ (\lambda\ \sigma_3,\ \_\ .$$
$$\mathfrak{R}\ :=\ \mathfrak{R} \cup \sigma_3.\mathcal{R}.\texttt{rcs}\ \ ;\ \text{success}()))))))$$

$$\text{verify}(\texttt{predicate}\ \ p(\overline{x : T})) = \texttt{well-formed}\,(\sigma_0\{\gamma\ :=\ \sigma_0.\gamma[\overline{x \mapsto \text{fresh}}]\},\ pred_{body},\ \text{fresh},\ (\lambda\ \_\ .\ \text{success}()))$$

Handles imprecision Handles run-time check generation and collection

Fig. 16. Rules defining a valid Gradual Viper program.

set of run-time checks is built up, which (along with success or failure) is the ultimate result of gradual verification.

4.4 Dynamic Verification: Encoding Run-time Checks into C0 Source Code

After static verification, Gradual Viper returns a collection of run-time checks $\mathfrak{R}$ that are required for soundness to GVC0. Then, GVC0 creates a C0 program from the run-time checks in $\mathfrak{R}$ and the original C0 program by encoding the checks in C0 source code. The C0 program is sent to the C0 compiler to be compiled, executed, and thus dynamically verified. We chose to encode the run-time checks directly in source code to avoid complexities from augmenting the C0 compiler with support for dynamic verification. Further, since C0 is a simple imperative language, any more expressive language should be able to encode the checks far more easily. That is, we hope this work serves as a guide to the developers of Gradual Viper frontends for other languages on how to implement efficient dynamic verification for gradual verification—especially, when modifying the compiler for their language is difficult. The rest of this section illustrates GVC0's encoding of run-time checks into C0 source code via example. We also highlight design points in the encoding that minimize run-time overhead of the checks during execution.

Now, consider the C0 program in Figure 17 that implements a method for inserting a new node at the end of a list, called `insertLastWrapper`. Note, when passed a non-empty list, `insertLastWrapper` calls `insertLast` from Figure 1 to perform insertion (line 17). Here, `insertLast` is gradually verified with the simpler and fully specified (precise) `acyclic` predicate given on lines 1–4 in Figure 17. For our purposes, we only need to know that `insertLast`'s precondition is `? && acyclic(list) && list != NULL` and its postcondition is `acyclic(\result) && \result != NULL`. The `insertLastWrapper` method is also gradually specified: Its precondition is `?` (line 7)—requiring unknown information—and its postcondition is `acyclic(\result)` (line 8)—ensuring the list after insertion is acyclic. Figure 17 also contains run-time checks generated by Gradual Viper for `insertLastWrapper`, as highlighted in blue. The first check (lines 15–16) ensures the list `l` sent to `insertLast` (line 17) is acyclic, and is only required when `l` is non-empty (non-null). The second check (lines 20–21) ensures the list returned from `insertLastWrapper` is acyclic, and is only required when `insertLastWrapper`'s parameter `l` is empty (null). These checks are not executable by the C0 compiler; therefore, GVC0 takes the program and checks in Figure 17 and returns the executable program in Figure 18. That is, GVC0 encodes branch conditions (lines 15 and 20), predicates (lines 16 and 21), accessibility predicates (**`acc`**`(l->val)` and **`acc`**`(l->next)` in `acyclic`'s body, lines 1–4), and separating conjunctions (also in `acyclic`'s body) from Gradual

```
1  /*@ predicate acyclic(Node* l) =
2        l == NULL ? true :
3          acc(l->val) && acc(l->next) &&
4          acyclic(l->next); @*/
5
6  Node* insertLastWrapper(Node* l, int val)
7    //@ requires ?;
8    //@ ensures acyclic(\result);
9  {
10   if (l == NULL) {
11     l = alloc(struct Node);
12     l->val = val;
13     l->next = NULL;
14   } else {
15   ({(none, l == NULL, ¬(l == NULL))},
16      (l=insertLast(l,val), acyclic(l), acyclic(l))
17     l = insertLast(l, val);
18   }
19   return l;
20   ({(none, l == NULL, l == NULL)},
21      (none, acyclic(\result), acyclic(\result))
22 }
```

Run-time checks from Gradual Viper

Fig. 17. Original `insertLastWrapper` program with run-time checks from Gradual Viper.

```
1  Node* insertLastWrapper(Node* l, int val,
2    OwnedFields* _owned, int* _instCtr)
3  {
4    bool _cond = l == NULL;
5    if (l == NULL) {
6      l = alloc(struct Node);
7      l->_id = *_instCtr;
8      *_instCtr++;
9      runtime_addAll(_owned, l->_id, 2);
10     l->val = val; l->next = NULL;
11   } else {
12     if (!_cond) { assert_acyclic(l, _owned); }
13     OwnedFields* _temp = runtime_init();
14     add_acyclic(l, _temp);
15     l = insertLast(l, val, _owned, _instCtr);
16   }
17   if (_cond) { assert_acyclic(l, _owned); }
18   OwnedFields* _temp2 = runtime_init();
19   add_acyclic(l, _temp2);
20   return l;
21 }
```

Run-time checks from GVC0

Fig. 18. GVC0 generated `insertLastWrapper` program with run-time checks.

Viper into C0 source code. We discuss the aforementioned encodings in Sections 4.4.1 to 4.4.3, respectively. While not in the `insertLastWrapper` example, GVC0 translates checks of simple logical expressions into C0 assertions: e.g., `assert(y >= 0);`.

4.4.1 Encoding Branch Conditions. Run-time checks contain branch conditions that denote the execution path for which a check is required. For example, in Figure 17 `acyclic(\result)` should only be checked at lines 20–21 when `l == NULL`, as indicated by the branch condition `(none,l == NULL,l == NULL)`. Therefore, GVC0 first encodes the condition `l == NULL` into C0 code. In general, conditions are encoded as logical expressions in C0 and assigned to fresh Boolean variables at the program point where they originated—we call this *versioning*. Then, the Boolean variable is used in checks in place of the condition. For example, the origin and location pair `(none,l == NULL)` tells GVC0 that `l == NULL` must be evaluated at the program point in `insertLastWrapper` containing the `l == NULL` AST element. As a result, in Figure 18 a Boolean variable `_cond` is introduced on line 4 to hold the value of `l == NULL`. The condition variable `_cond` is then used in the C0 run-time check for `acyclic(\result)` later in the program (line 17). To reduce run-time overhead, `_cond` is also used in the check for `acyclic(l)` on line 12, which relies on the same branch point `(none,l == NULL)`. Further, while not demonstrated here, GVC0 adds guards so that the versioned conditions are false if evaluating the path condition would result in a null dereference.

4.4.2 Encoding Predicates. Now that GVC0 has versioned the branch conditions in Figure 17 into variables, GVC0 can use the variables to develop C0 run-time checks. The Gradual Viper check `({(none,l == NULL,¬(l == NULL))}, (l=insertLast(l,val),acyclic(l),acyclic(l)))` is translated into `if (!_cond) {assert_acyclic(l,_owned);}` on line 12 in Figure 18. GVC0 places this C0 check according to the origin, location pair `((l=insertLast(l, val),acyclic(l)))`, which points to the program point just before the call to `insertLast` on line 15. The branch condition becomes the if statement with condition `!_cond` (Section 4.4.1), and `acyclic(l)` is turned

into the C0 function call `assert_acyclic(l,_owned)`. The `assert_acyclic` function implements `acyclic`'s predicate body as C0 code: It asserts true for empty lists and recursively verifies accessibility predicates (using `_owned`) for nodes in non-empty lists. That is, predicates are encoded and treated equi-recursively by GVC0. For efficiency, separation of list nodes is encoded separately on lines 13–14. We discuss the dynamic verification of accessibility predicates and the separating conjunction in C0 code next (Section 4.4.3). Finally, a similar C0 check is created for `acyclic(\result)` on lines 17–19. Note, our encoding of predicates, which turns them into Boolean functions in code, is made possible by Gradual C0 not allowing existentials in its specification language.

4.4.3 Encoding Accessibility Predicates and Separating Conjunctions. GVC0 implements run-time tracking of owned heap locations in C0 programs to verify accessibility predicates and uses of the separating conjunction.

Encoding Owned Fields in C0 Source Code. An owned field is a tuple (*id*, *field*) where *id* is an integer identifying a struct instance (object in C0) and *field* is an integer indexing a field in the struct. The `OwnedFields` struct, which is implemented as a dynamic hash table to improve check performance, contains currently owned fields. That is, hashed object identifiers (*id*) and then field identifiers (*field*) are used to index into `OwnedFields` where a Boolean that determines whether or not the field is currently owned is stored. Since objects are tracked with integers, all struct definitions in a C0 program are modified to contain an additional `_id` field.

Semantics of Tracking Owned Fields (Inspired by Wise et al. [2020]). At the entry point to a C0 program (e.g., `main`), an empty `OwnedFields` struct, which we call `_owned`, is allocated and initialized. This is not shown in Figure 18. Then, when a new struct instance is created—such as allocating a new node on line 6 in Figure 18—the `_id` field is initialized with the value of a global counter `_instCtr` that uniquely identifies the instance (lines 7–8). The call to library function `runtime_addAll` on line 9 adds all fields in the struct instance (e.g., `l->val:(l->_id,0)` and `l->next:(l->_id,1)`) to `_owned` and marks them as owned. The only other times `_owned` can change are at method/function calls and loops. Methods (loops), like `insertLast` and `insertLastWrapper`, may add or drop owned fields during their executions. They may also contain run-time checks, such as the one for `acyclic(l)` on line 12, that need owned fields for verification. So, GVC0 adds an additional parameter to their declarations (e.g., line 2, Figure 18) to accept, initialize, and then modify `_owned` in their contexts. A callee's pre- and postcondition (a loop's invariant) controls what owned fields are passed to and from the callee (loop body) via this new parameter. When a method's precondition (loop's invariant) is imprecise, then any caller (program before loop execution) will pass all of its owned fields to the method (loop), as on line 15 for the call to `insertLast`. Note, here, a formula is also imprecise if it contains predicates that expose ? when fully unrolled—an equi-recursive treatment appropriate for dynamic systems [Wise et al., 2020] (this is in contrast with the iso-recursive treatment in Gradual Viper, see Section 4.2 for more information). After execution, the callee method (loop body) returns all of its owned fields to the caller (program after the loop). When a method's precondition (loop's invariant) is precise, then any caller (program before the loop) only passes its owned fields specified by the precondition (invariant) to the method (loop body). If the method's postcondition is imprecise, then after execution the callee method returns all of its owned fields as before; otherwise, only the owned fields specified by the postcondition (loop invariant) are returned. Finally, as an optimization, in precisely specified (no external—pre- and postconditions—or internal—loop invariants, unfolds, folds, etc.—specifications contain imprecision and no run-time checks are required) methods (loops), GVC0 does not implement any tracking of owned fields. In this case, GVC0 uses the callee's pre- and postcondition (loop's invariant) to modularly update `_owned` in the caller (program outside the loop).

Verifying Accessibility Predicates and Separating Conjunctions with Owned Fields. Now, `_owned` is used to verify accessibility predicates and uses of the separating conjunction. Run-time checks for accessibility predicates are turned into assertions that ensure the presence of their heap location in `_owned`. For example, **acc**(`l->val`) looks like `runtime_assert(_owned,l->_id,0);` in C0 code, where 0 is the index for `val` in the `Node` struct. The `runtime_assert` library function indexes into `_owned` using `l->_id` and 0 and ensures the corresponding flag is `true`; otherwise, `runtime_assert` throws an error. Wherever GVC0 must check separation of heap locations (as indicated in runtime checks from Gradual Viper via a flag)[10]—such as for the nodes in list `l` at lines 12–14—it creates (with the library method `run-time_init`) a new (empty) instance `_temp` of `OwnedFields`. We check that heap cells are disjoint by adding them one at a time to `_temp` and failing if the cell has been already added. GVC0 generates a `add_X` method for each predicate `X` to actually perform this operation; when done, it discards `_temp`, as its only purpose is to check separation. Similar checks are created for the `acyclic(\result)` check on lines 17–19.

4.5 Gradual C0 Soundness

Gradual C0 adheres to the following informal soundness statement:

> *For a C0 program p, if the symbolic execution algorithm in Gradual Viper succeeds for the Gradual Viper version of p, the run-time checks produced by Gradual Viper are inserted by GVC0 into p, and the run-time checks are assumed to succeed, then when executed p is guaranteed to not step into an invalid state with respect to its specifications and IDF-styled heap ownership.*

Put another way, all violations of a C0 program's specifications and all violations of heap ownership by the program code will be caught either statically (by Gradual Viper) or dynamically (by GVC0). A formal statement of soundness for Gradual C0 with proof is given by Zimmerman et al. [2024]. Note, Zimmerman et al. [2024] formalize Gradual C0's symbolic execution algorithm in sets of inference rules, rather than in CPS-style (as in this work) to facilitate stating and proving soundness.

5 Empirical Evaluation

The seminal work on gradual typing [Siek and Taha, 2006] selectively inserts run-time casts in support of optimistic static checking: For instance, whenever a function application is deemed well-typed only because of imprecision—such as passing an argument of the unknown type to a function that expects an integer—the type-directed cast insertion procedure inserts a run-time check. But if the application is definitely well-typed, no cast is inserted. This approach ensures that a fully precise program does not incur any overhead related to run-time type checking. While it is tempting to assume that more precision necessarily results in better performance, the reality has been shown to be more subtle: Both the nature of the inserted checks (such as higher-order function wrappers) and when/how often they are executed is of utmost importance [Muehlboeck and Tate, 2017; Takikawa et al., 2016], and anticipating the performance impact of precision is challenging [Campora et al., 2018].

The performance of gradual verification has never been studied until now, due to the lack of a working gradual verifier. Here, we explore the relation between minimizing dynamic check insertion with statically available information and observed run-time performance in gradual verification with Gradual C0. Specifically, we explore the performance characteristics of Gradual C0 for thousands of partial specifications generated from four data structures, as inspired by the work of Takikawa et al. [2016] in gradual typing. In particular, we observe how adding or removing individual atomic formulas and ? within a specification impacts the degree of static and dynamic

[10]Note, this flag is not formalized in this article for simplicity.

verification and, as a result, the run-time overhead of the program. Additionally, we compare the run-time performance of Gradual C0 to a fully dynamic approach, as readily available in C0. The aforementioned ideas are captured in the following **Research Questions (RQs)**:

RQ1: As specifications are made more precise, can more verification conditions be eliminated statically?
RQ2: Does gradual verification result in less run-time overhead than a fully dynamic approach?
RQ3: Are there particular types of specification elements that have significant impact in run-time overhead, and can high overhead be avoided?

A reproducibility package for the evaluation presented in this section can be found in the supplementary material for this article.

5.1 Creating Performance Lattices

We define a *complete* specification as being statically verifiable when all ?s are removed, and then a *partial* specification as a subset of formulas from a complete specification that are joined with ?. Like Takikawa et al. [2016], we model the gradual verification process as a series of steps from an unspecified program to a statically verifiable specification where, at each step, an *element* is added to the current, partial specification. An element is an atomic conjunct (excluding Boolean primitives) in any type of method contract, assertion, or loop invariant. We form a lattice of partial specifications by varying which elements of the complete specification are included. We also similarly vary the presence of ? in formulas that are complete—contain the same elements as their counterparts in the statically verifiable specification—and have related fold and unfold statements in the partial specification. Otherwise, ? is always added to incomplete formulas. This strategy creates lattices where the bottom entry is an empty specification containing only ?s and the top entry is a statically verifiable specification. A *path* through a lattice is the set of specifications created by appending n elements or removing ?s one at a time from the bottom to the top of the lattice. The large array of partial specifications created in each lattice closely approximates the positive specifications supported by the gradual guarantee [Wise et al., 2020], which are less precise variants of successfully verified programs. For reference, we give a more formal statement of the gradual guarantee:

> *Let p_1 and p_2 be Gradual C0 programs where $p_1 \sqsubseteq p_2$ (i.e., the formulas in p_1 are more precise than those in p_2). If p_1 statically verifies, then p_2 statically verifies. Additionally, p_2 must execute at least as far as p_1 executes at run time.*

Now, to illustrate the aforementioned approach, consider the following loop invariant:

```
//@ loop_invariant sortedSeg(list, curr, curr->val) && curr->val <= val;
```

The invariant is made of two elements: the `sortedSeg` predicate instance and the Boolean expression `curr->val <= val;`. The lattice generated for a program with this invariant has five unique specifications: Four contain a combination of the two elements joined with ?, and the fifth is the complete invariant above.

5.2 Data Structures

To apply this methodology, we implemented and fully specified four recursive heap data structures with Gradual C0: **Binary Search Tree (BST)**, sorted linked list, composite tree, and AVL tree. Their implementations with specifications can be found in the supplementary material published alongside this work. We chose these data structures because complete static specifications exist

Table 1. Description of Benchmark Examples

Example	Unverified Complexity	# Specs	Contents of Complete Spec					
			Fold	*Unfold*	*Pre.*	*Post.*	*Pred. Body*	*Loop Inv.*
BST	$O(n\ log(n))$	3,473	43	23	0/20/21/24	0/22/6/24	6/6/7/4	0/2/4/2
Linked List	$O(n)$	1,745	17	10	8/6/15/5	4/5/6/5	4/3/4/3	4/3/5/2
Composite	$O(n\ log(n))$	2,577	28	15	0/10/2/12	0/11/1/12	32/9/17/3	0/3/2/3
AVL	$O(n\ log(n))$	3,057	25	14	3/4/5/9	3/6/9/9	25/8/21/3	1/1/2/1

For each example, the table shows the complexity of the test program without verification, the number of sampled partial specifications, and the distribution of specification elements for the complete specification. Element counts are formatted as "*Accessibility Predicate/Predicate Instance/Boolean Expression/Imprecision.*"

for them in prior work and they are interesting use cases for gradual verification. Linked list is implemented with a while loop rather than recursion. BST is a more complex data structure with a more complex property (BST property) than a linked list and uses recursion. Composite tree implements a structure where modifications do not have to start at the root, but can be applied directly to any node in the tree. Its invariant also applies to any node in the tree. Finally, AVL tree implements the most complex invariant (the balanced property) and data structure with many interdependent functions and predicates related to tree rotations. Each data structure has a test program that contains its implementation and a main function that adds elements to the structure based on a workload parameter ω. We design the test programs to incur as little run-time overhead as possible outside of structure size and run-time checks. For each example and corresponding test program, Table 1 displays the distribution of elements in the complete specification, as well as the run-time complexity of the test program and the number of unique partial specifications generated by our benchmarking tool.

BST. The implementation of the BST is typical; each node contains a value and pointers to left and right nodes. We completely statically specify BST with required ownership specifications and also to verify preservation of the BST property—that is, any node's value is greater than any value in its left subtree and less than any value in its right subtree. The test program creates a root node with value ω and sequentially adds and removes a set of ω values in the range $[0, 2\omega]$. Note that values are removed in the same order they were added.

Linked List. We implement a linked list with insertion similar to the one given in Figure 1. Insertion is statically specified with heap ownership specifications as well as those denoting preservation of list sortedness. Its test program creates a new list and inserts ω arbitrary elements.

Composite. The composite data structure is a binary tree where each node tracks the size of its subtree—this is verified by its specification along with its heap accesses. Its test program starts with a root node and builds a tree of size ω by randomly descending from the root until a node without a left or right subtree is reached. A new node is added in the empty position, and then traversal backtracks to the root.

AVL Tree. The implementation of AVL tree with insertion is standard except that the height of the left and right subtrees is stored in each node (instead of the overall height of the tree). This allows us to easily state the AVL balanced property—for every node in the tree the height difference between its left and right children is at most 1—without using functions in specifications or the `unfolding` `p(`$\overline{e}$`)` `in` e construct [Müller et al., 2016], which Gradual C0 does not currently support (we discuss this further in Section 5.5). In addition to specifying the AVL balanced property for insertion, we also specify required ownership specifications. The AVL test program starts with a root node and builds a tree of size ω by inserting randomly valued nodes into the tree using balanced insertion.

5.3 Experimental Setup

With upwards of 100 elements in the specifications for each data structure, it is combinatorially infeasible to fully explore every partial specification. Therefore, unlike Takikawa et al. [2016], we proceed by *sampling* a subset of partial specifications in a lattice, rather than executing them all. Specifically, we sample 16 unique paths through the lattice from randomized orderings of specification elements. We chose partial specifications along lattice paths to explore trends in migration from no specifications to complete specifications, which is how we imagine developers may use our tool. We also randomly sampled paths, rather than using another heuristic, to be prescriptive to users of Gradual C0. We wanted to find and recommend new specification patterns that users should apply or avoid depending on their performance. Every step is executed with three workloads chosen arbitrarily to ensure observable differences in timing. Each timing measurement is the median of 50 iterations. Programs were executed on eight physical 13th Gen Intel i7-13700k 3.4 GHz Cores with 32 GB of RAM. Hyperthreading, turbo boost, and low-power cores were disabled. Power management for the remaining cores was set to performance.

We introduce two baseline verifiers to compare Gradual C0 against. The *dynamic verifier* transforms every specification into a run-time check and inserts accessibility predicate checks for field dereferences—thereby emulating a fully dynamic verifier. The *framing verifier* only performs the accessibility predicate checks, and therefore represents the minimal dynamic checks that must be performed in a language that checks ownership.[11] We implement the baseline verifiers ourselves using the dynamic semantics of Wise et al. [2020], which checks everything at run time, as a guide. The work by Wise et al. [2020] is the only work that we are aware of that handles run-time checking of both ownership and recursive predicates.

5.4 Evaluation

Figure 19 shows how the total number of verification conditions (proof obligations) changes as more of each benchmark is specified (green curve). The figure also similarly shows the number of verification conditions that are statically verified as each benchmark is specified (purple curve). From the green curve, we see that even when there are no specifications, there are verification conditions, e.g., before a field is accessed, the object reference must be non-null and the field must be owned. Some of these verification conditions can be verified statically as illustrated by the purple curve. As more of a benchmark is specified, there are more verification conditions (green curve); but also, more of these verification conditions are discharged statically and do not have to be checked dynamically (purple curve). Towards the right end of the plots, the two curves converge until they meet when all the verification conditions are discharged statically. As a result, the answer to *RQ1* is *yes*. Note, the number of verification conditions does decrease when enough of the benchmark is specified. This is due to being able to prune symbolic execution paths with new static information.

The plots in Figure 20 display the run-time performance (in red) of dynamically checking the verification conditions from Figure 19. The plots also show how the run-time performance of the dynamic verifier (in green) and framing verifier (in purple) change as more of each benchmark is specified. The green lines show that as more properties are specified, the cost of run-time verification increases. With Gradual C0, some of these properties can be checked statically; therefore, the run-time cost of gradual verification, shown in red, starts equivalent but eventually ends up significantly lower than the cost of pure run-time verification.

Notably, the purple lines are significantly lower than the red and greens ones until they exhibit a dramatic increase starting at around 80% specified all the way to 100%. Eventually (after about 95% specified), the purple lines end above the red ones (but below the green ones) where running

[11] These framing checks could fail, for example, if some function lower in the call stack owns data that are accessed by the currently executing function.

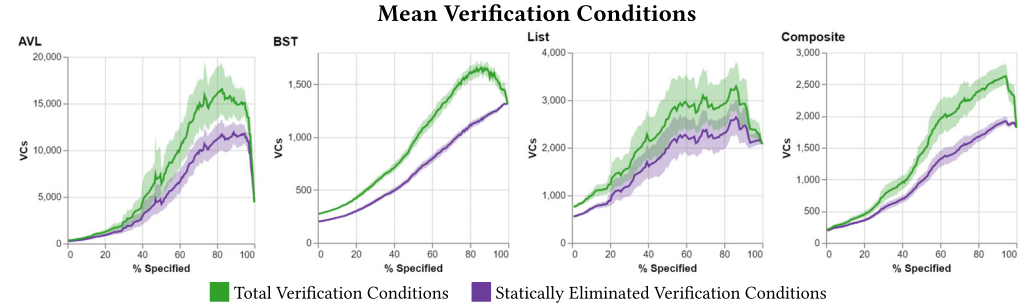

Fig. 19. For each example, the average quantity of verification conditions and the subset that were eliminated statically at each level of specification completeness across all paths sampled. Shading indicates the standard deviation.

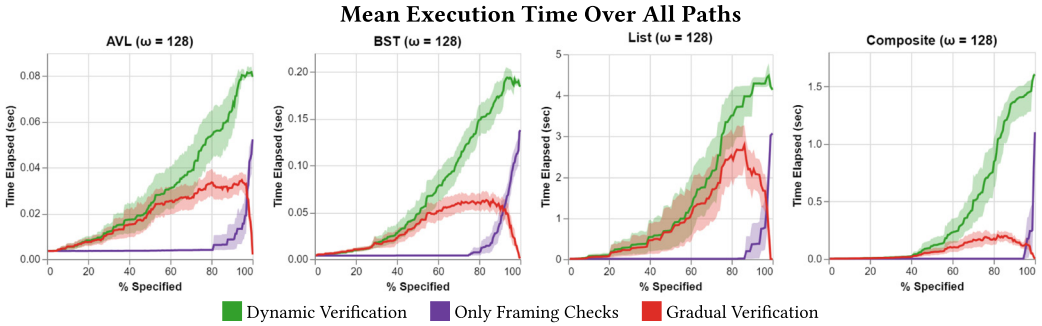

Fig. 20. The mean time elapsed at each step over the 16 paths sampled. Shading indicates the confidence interval of the mean for each verification type.

time is orders of magnitude higher than at the start of the incline. The framing verifier (in purple) only checks that heap accesses are safe—i.e., they are owned and their receivers are non-null. So unsurprisingly, the dynamic and gradual verifiers, which check more properties like heap separation, nearly always have significantly higher run-time verification overhead than the framing verifier. Eventually, Gradual C0 outperforms the framing verifier when enough properties, including framing, are checked statically.

The dramatic increase in the framing verifier's run-time performance is caused by the owned fields passing strategy employed at method boundaries and loops (described in Section 4.4.3) to verify heap accesses at run time. To respect precondition (loop invariant) abstractions, only owned fields specified by a callee's precondition (loop's invariant) are passed by the caller (program before the loop) to the callee (loop body) when the precondition (invariant) is precise. Similarly, when a callee's postcondition (loop invariant) is precise, then only the owned fields specified by the postcondition (invariant) are passed back to the caller (program after the loop). Computing owned fields from precise contracts and loop invariants is costly, and even more-so for contracts and loop invariants containing recursive predicates, like in our benchmarks. Further, our benchmarks call such methods and loops frequently during execution. As a result, execution time increases significantly at each path step where one of the aforementioned methods gets a precise pre- or postcondition or loop gets a precise loop invariant from ? removal. This, of course, happens more frequently as more of a benchmark is specified. At 100% specified, every method contract and loop invariant is precise, and so the owned fields passing strategy is used at every method call and loop. This leads to the highest run-time costs for the framing verifier. In contrast, Gradual C0 checks

Table 2. Summary Statistics for the Performance of Each Example over 16 Paths at Selected Workloads (ω), Comparing Gradual Verification (GV) against Dynamic Verification (DV)

Example	ω	% Δt, GV vs. DV				% Steps GV < DV for Paths DV < GV				% Paths GV < DV
		Mean	*SD*	*Max*	*Min*	*Mean*	*SD*	*Max*	*Min.*	
AVL	32	−7.1	23.9	197.9	−87.5	56.7	17.8	85.9	24.6	0.0
	64	−14.0	28.3	219.5	−95.7	66.9	20.4	90.6	30.4	0.0
	128	−15.9	31.7	194.7	−98.0	69.4	21.2	94.8	28.8	0.0
BST	32	−25.9	27.2	13.7	−92.5	75.6	7.6	89.4	61.3	0.0
	64	−27.8	30.5	25.5	−98.1	71.0	8.5	83.9	55.8	0.0
	128	−26.1	33.2	51.4	−99.5	68.4	10.0	86.2	48.4	0.0
Linked List	32	−16.3	23.5	18.5	−96.2	76.5	20.6	94.5	28.4	0.0
	64	−22.4	28.5	28.9	−99.8	84.2	12.9	100.0	54.1	6.3
	128	−25.0	29.7	40.4	−100.0	86.9	8.6	100.0	67.0	6.3
Composite	32	−37.7	34.4	21.4	−98.9	84.2	13.2	99.4	50.3	0.0
	64	−39.4	35.8	41.2	−99.8	84.4	13.5	99.4	49.7	0.0
	128	−40.2	37.3	82.6	−100.0	85.8	11.7	98.8	62.1	0.0

The grouped column "% in Δt, GV vs. DV" displays summary statistics for the percent decrease in time elapsed for each step when using GV versus DV. The column "% Steps GV < DV for Paths DV < GV" shows the distribution of steps that performed best under GV that were part of paths containing steps that performed better under DV. The final column shows the percentage of paths in which every step performed better under GV.

fully specified methods and loops completely statically and does not use the owned field passing strategy for calls to these methods and loops. As a result, looking at the red lines, Gradual C0 is not heavily affected by this phenomena—we see slight increases starting at 90% specified but they are significantly less costly. Additionally, once a critical mass of specifications have been written, Gradual C0's run-time verification cost decreases until reaching zero—which is the same as running the raw C0 version of the benchmark. If the spikes around 90% specified are too costly, production gradual verifiers can reduce them by employing more optimal permission passing strategies.

In general, according to the red lines, Gradual C0's performance increases gradually as more proof obligations are specified but are not yet statically verified; and thus, must be checked at run time. When a critical mass of specifications are written, more and more of these proof obligations can be proven statically. This causes run-time performance to start to decrease until reaching the spikes around 90% specified caused by owned fields passing. After the spikes, performance decreases to the benchmark's raw baseline. This trend is consistent with speculations made in the work by Wise et al. [2020] and confirms that increasing precision in gradual verification does not always correspond with decreased run-time overhead from dynamic verification.

Table 2 displays summary statistics for Gradual C0's performance on every sampled partial specification compared to the dynamic verification baseline. Depending on the workload and example, Gradual C0 reduces run-time overhead by 7.1–40.2% on average (Table 2, Column 3) compared to the dynamic verifier. Note that the speed-ups are consistent or increase as ω increases: −7.1%, −25.9%, −16.3%, and −37.7% at the lowest ω values compared to −15.9%, −26.1%, −25.0%, and −40.2% at the largest. While Gradual C0 generally improves performance, there are some outliers in the data (Table 2, Column 5) where Gradual C0 is slower than dynamic verification by 13.7–219.5%. Fortunately, for lattice paths that produce these poor-performing specifications, gradual verification still outperforms dynamic verification (on average) for 56.7–86.9% (Table 2, Column 7) of all steps. Further, the majority of these outliers appear under 50% specified. These outcomes are likely caused by the bookkeeping we insert to track conditionals, which is unoptimized and could be improved, and measurement error. Figure 20 displays the average run-time cost across all paths under each of our benchmarks and verifiers. In all the plots, for some early parts of the path, the cost of Gradual C0 is comparable to or slightly exceeds the cost of the dynamic verifier. But after 50% completion, static optimization kicks in, and Gradual C0 begins to significantly outperform it. Further, Table 2 shows

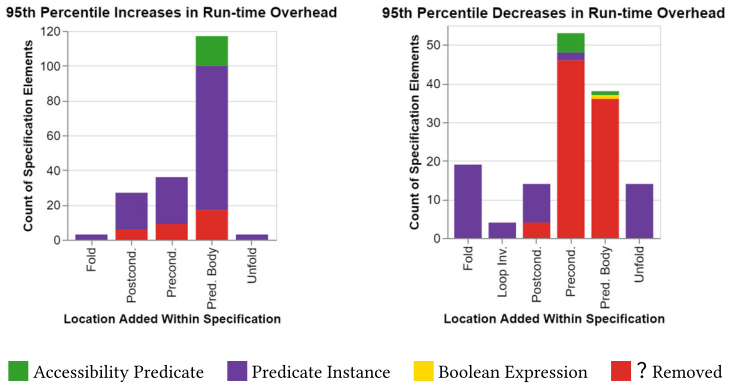

Fig. 21. The quantity of specification elements, grouped by type and location, that caused the highest ($P_{99\%}$) increases and decreases in time elapsed out of every path sampled.

that on average, Gradual C0 reduces run-time overhead by 7.1–40.2% compared to the dynamic verifier. Therefore, the answer to *RQ2* is *yes*.

Figure 21 captures the impact that different types of specification elements (accessibility predicates, predicates, and Boolean expressions) have on Gradual C0's run-time performance when specified in different locations. It also captures the impact removing ? from a formula has on performance. Elements that when added or ? that when removed from one step in a lattice path to another increase run-time overhead significantly (in the top 1%) are counted in the left subfigure, and ones that decrease run-time overhead significantly (top 1%) are counted in the right subfigure. The count for accessibility predicates is colored in green, predicates in purple, Boolean expressions in yellow, and ? removal in red.

Adding predicates to preconditions, postconditions, and predicate bodies is the most frequent cause (70.4%) of dramatic increases in run-time verification overhead during the specification process. When these predicates are added to preconditions and postconditions, they create additional proof obligations for them in callers and callees (respectively) that are frequently checked at run time. Similarly, when they are added to predicate bodies (often as recursive calls) any proof obligations for the enclosing predicate that are checked at run time become far more expensive. Fortunately, folding or unfolding a predicate can decrease run-time cost when doing so discharges such proof obligations statically (as seen in the right subfigure). Therefore, users of Gradual C0 may consider specifying proofs of recursive predicates in frequently executed code to significantly reduce checking costs.

Removing ? from preconditions, postconditions, and predicate bodies when the costly owned fields passing strategy is still required in corresponding methods is the second most frequent cause (17.2%) of increases in Gradual C0's run-time overhead. This corresponds with the spikes at 90% specified in Figure 20 for Gradual C0: Removal of ? in the aforementioned locations leads to precise pre- and postconditions that trigger the use of this costly strategy. Eventually, a critical mass of specifications are written so that when ?s are removed further, this costly strategy is no longer necessary (i.e., when callee methods are full statically verified), and so run-time performance improves dramatically—the downward trends seen prior to full static specification in Figure 20. This is reflected in the right subfigure in Figure 21, where removing ? from preconditions, postconditions, and predicate bodies is the most frequent cause (60.6%) of significant decreases in run-time overhead. This suggests a strategy for avoiding high checking costs: Specify frequently executed code in critical-mass chunks that are fully statically verifiable, leaving boundaries between statically and dynamically verified code in places that are executed less frequently.

Overall, the answer to *RQ3* is *yes*; we have identified some key contributors to run-time overhead, whose optimization is a promising direction for future work, and we have also identified strategies for minimizing run-time overhead in practice.

Finally, all of the partial specifications evaluated in our study were successfully verified by Gradual C0. Since they originated from complete and correct specifications on code, we can conclude Gradual C0 adheres to the gradual guarantee for these partial specifications and likely adheres to the gradual guarantee for common use cases of Gradual C0.

Takeaways. Gradual C0 successfully statically verifies more verification conditions (proof obligations) as more specifications are written (*RQ1*). But, while this does translate to 7.1–40.2% less run-time overhead on average for Gradual C0 compared to a fully dynamic approach (*RQ2*), we discovered that increasing precision does not always correspond with decreased run-time overhead from dynamic verification. In fact, we observed that Gradual C0's run-time performance increases as precision increases—and expensive proof obligations are introduced without static proof—until a critical mass of specifications are written after which run-time performance decreases—as more of these proof obligations are checked statically (this trend is consistent with speculations made in prior work [Wise et al., 2020]). In particular, we observed that adding expensive-to-runtime-check specification constructs, such as predicates (specifically recursive ones), to pre- and postconditions without corresponding specifications (folds and unfolds) that allow Gradual C0 to check them statically results in significant increases to run-time overhead (*RQ3*). The overhead is also worse when such specifications are completely precise and specify the shape of the heap, which triggers our costly owned fields passing strategy. This leads us to two recommendations: (1) production gradual verifiers should employ more optimal permission passing strategies, and (2) users of gradual verification should specify frequently executed code in critical-mass chunks that are fully statically verifiable, leaving boundaries between statically and dynamically verified code in places that are executed less frequently.

Threats to Validity. While the test programs we used are of sufficient complexity to demonstrate interesting empirical trends, they are not representative of all software. Further, the baseline we used for dynamic verification is entirely unoptimized as we naively insert a check for each written element of a specification. Finally, due to computational constraints, only a small subset of over 2^{100} possible imprecise specifications were sampled, and we did not use a formal criteria to choose our workload values. As such, while our results reveal interesting trends, including significant performance improvements by Gradual C0 over dynamic verification, more work is needed to validate the robustness of those trends.

5.5 Qualitative Experience with AVL Tree

Incremental Feedback from Gradual C0 is Useful for Developing Complete Specifications. Notably, it was our experience that the incrementality of gradual verification was very helpful for developing a complete specification of the AVL tree example. In particular, a run-time verification error from a partial specification helped us realize the contract for the `rotateRight` helper function was not general enough. We fully specified `rotateRight` and proved it correct. However, `insert`'s pre- and postconditions were left as `?`, and so static verification could not show us that the contract proved for `rotateRight` was insufficiently general. Nevertheless, we ran the program; gradual verification inserted run-time checks, and the precondition for `rotateRight` failed. This early notification allowed us to identify the problem with the specification and fix it immediately. Otherwise, we would have had to get deep into the static verification of `insert`—a complicated function, 50 lines long, with lots of tricky logic and invariants—before discovering the error, and a lot of verification work built on the faulty specification would have had to be redone. Interestingly, it is conventional

```
1  struct Node {
2    int key;
3    struct Node* left;
4    struct Node* right;
5    int height;
6  };
7
8
9  /*@
10 predicate avlh(Node* root) =
11   root == NULL ?
12     true
13     :
14     acc(root->left) && acc(root->right) &&
15     acc(root->key) &&
16     acc(root->height) &&
17     avlh(root->left) &&
18     avlh(root->right) &&
19     unfolding avlh(root->left) in
20       unfolding avlh(root->right) in
21         -1 <= getBalance(root) &&
22         getBalance(root) <= 1 &&
23         root->left->height >= 0 &&
24         root->right->height >= 0 &&
25         root->height ==
26           max(root->left->height,
27               root->right->height)+1
28     ;
29 @*/
```

Fig. 22. AVL tree balance property specified with pure functions and unfolding-in.

```
1  struct Node {
2    int key;
3    struct Node* left;
4    struct Node* right;
5    int lheight;
6    int rheight;
7  };
8
9  /*@
10 predicate avlh(Node* root,int height) =
11   root == NULL ?
12     height == 0
13     :
14     acc(root->left) && acc(root->right) &&
15     acc(root->key) &&
16     acc(root->lheight) &&
17     acc(root->rheight) &&
18     avlh(root->left,root->lheight) &&
19     avlh(root->right,root->rheight) &&
20     root->lheight - root->rheight < 2 &&
21     root->rheight - root->lheight < 2 &&
22     root->lheight >= 0 &&
23     root->rheight >= 0 &&
24     (root->lheight > root->rheight ?
25        height == root->lheight+1 :
26        height == root->rheight+1)
27     ;
28 @*/
```

Fig. 23. AVL tree balance property specified in our AVL benchmark program.

Ownership specs | Recursive call | Balance property
Positive left-right heights | Current node height

wisdom that one of the benefits of static checking is that you get feedback early, when it is easier to correct mistakes. Here, we encountered a scenario where gradual verification had a similar benefit over static verification! We found an error (in a specification) earlier than we would have otherwise, presumably saving time.

Gradual C0 Suffers from Expressiveness Limitations. The specification language of Gradual C0 is missing pure functions and the `unfolding p(`$\overline{e}$`) in` e construct from Viper. This forced us to deviate from the standard implementation of AVL tree to statically specify and prove our implementation preserves the balance property—for every node in the tree the height difference between its left and right children is at most 1. To illustrate, consider Figure 22, which contains the traditional `Node` implementation for an AVL tree: Each node contains a key (value), pointers to its left and right child, and its height in the tree. Figure 22 also contains the recursive `avlh` predicate (lines 10–28), which specifies that the tree starting at the given root node is balanced using pure functions and `unfolding p(`$\overline{e}$`) in` e. Note, `unfolding p(`$\overline{e}$`) in` e is used to allow predicates to frame expressions; i.e., `p(`$\overline{e}$`)` is temporarily unfolded once and the body of `p(`$\overline{e}$`)` is used when determining whether or not e is framed. For example, `x->f == 2` is not framed, but `p(x) && unfolding p(x) in x->f == 2` is framed if `p(x)`'s body contains **`acc`**`(x->f)`. Now, let's take a closer look at `avlh` in Figure 22. When the current node is `NULL` then the balanced property trivially holds for the tree starting at this node (line 12). Otherwise, (1) ownership of the heap locations in the node are specified as highlighted in

green (lines 14–16), (2) the node's left and right subtrees must be balanced as specified recursively in orange (lines 17–18), (3) the difference between the heights of the subtrees must be at most 1 (specified in blue on lines 21–22), (4) the heights of the subtrees must be positive (specified in red on lines 23–24), and (5) the current node's height must be 1 more than the height of its tallest subtree (specified in yellow on lines 25–27). Since the specifications in 3–5 constrain the left and right subtrees' heights using `root->left->height` and `root->right->height`, the heap locations in `root- >left->height` and `root->right->height` must be framed by corresponding accessibility predicates. The locations `root->left` and `root->right` are framed by accessibility predicates specified on line 14; and notably, **acc**(`root->left->height`) and **acc**(`root->right->height`), which are contained in `avlh(root->left)` and `avlh(root->right)`, respectively, are provided for framing via the unfolding-ins on lines 19–20. Additionally, the pure functions `getBalance`—which returns `root->left->height - root->right->height` when root is non-null—and `max`—which returns the maximum of the two arguments—are reused from code in `avlh`'s body for convenience on lines 21–22 and 25–27.

Without `unfolding p(`$\overline{e}$`) in` e and pure functions, framing access to the left and right subtrees' heights in `avlh` becomes tricky because we do not have a way to reach inside the recursive calls to `avlh` where the accessibility predicates for those heap locations are. Our solution in the AVL benchmark is to modify the implementation of `Node` to track the left and right subtrees' heights directly instead of the height of the tree starting at the current node. This can be seen in Figure 23 on lines 1–7. Then `avlh` can be specified without unfolding-ins and pure functions as seen in Figure 23 on lines 10–27. Compared to `avlh` from Figure 22, `avlh` in Figure 23 has (1) ownership specs for the left and right subtree heights (lines 16–17) instead of the overall height, (2) tracks the overall height as an argument (lines 10, 18–19), (3) constrains the left and right subtree heights directly via the current node (lines 18–26) instead of via the left and right nodes, and (4) computes the balance factor (lines 20–21) and max of the subtree heights (lines 24–26) directly instead of via pure function calls. As such, Gradual C0's specification language is expressive enough to support the complete static specification and verification of AVL tree, but requires round-a-bout solutions and nontraditional adjustments to its implementation. Fortunately, Gradual C0's algorithms presented in this article can be straightforwardly extended to support `unfolding p(`$\overline{e}$`) in` e and pure functions by following solutions for other closely related constructs, such as predicates, nonpure functions, and accessibility predicates.

6 Related Work

Much of the closely related work, particularly work on gradual verification [Bader et al., 2018; Wise et al., 2020; Zimmerman et al., 2024], gradual typing [Herman et al., 2010; Siek and Taha, 2006; Siek et al., 2015; Takikawa et al., 2016], and static verification [Müller et al., 2016; Parkinson and Bierman, 2005; Reynolds, 2002; Smans et al., 2009], has been discussed throughout the article. Note, this article and the paper by Zimmerman et al. [2024] have distinct contributions. Our work includes the original development of symbolic execution-based gradual verification and describes the first implementation of the same, as well as related empirical results. The paper by Zimmerman et al. [2024] proves that the approach is sound. Now, we discuss additional related work.

Gradual Typing. Additional related work in gradual typing includes richer type systems such as gradual refinement types [Lehmann and Tanter, 2017] and gradual dependent types [Eremondi et al., 2019; Lennon-Bertrand et al., 2022]. These systems focus on pure functional programming, while Gradual C0 targets imperative programs. There is an extensive body of work on optimizing run-time checks in gradual type systems. Muehlboeck and Tate [2017] show that in languages with nominal type systems, such as Java, gradual typing does not exhibit the usual slowdowns induced by

structural types. Feltey et al. [2018] reduce run-time overhead from redundant contract checking by contract wrappers. They eliminate unnecessary contract checking by determining—across multiple contract checking boundaries for some datatype or function call—whether some of the contracts being checked imply others. While the results in Section 5 are promising, we may be able to draw from the extensive body of work in gradual typing to achieve further performance gains.

Static Verification. Work in formal verification contains approaches that try to reduce the specification burden of users—a goal of Gradual C0. Furia and Meyer [2010] infer loop invariants with heuristics that weaken postconditions into invariants. When that approach fails, verification also fails because invariants are missing. Similarly, several tools (SmallFoot [Berdine et al., 2006], JStar [Distefano and Parkinson, 2008], and Chalice [Leino et al., 2009]) use heuristics to infer fold and unfold statements for verification. In contrast, Gradual C0 does not fail solely because invariants, folds, or unfolds are missing; imprecision begets optimism. However, Gradual C0 may benefit from similar heuristic approaches by leveraging additional static information to further reduce run-time overhead.

Abductive reasoning (*abductive inference*) tries to find an explanatory hypothesis for a desired outcome [Dillig et al., 2012]. In static verification, the desired outcome is a proof obligation (O), facts (F) are invariants derived from the program and specifications using some analysis, and the explanatory hypothesis (E) are invariants that do not contradict the derived facts (SAT($F \wedge E$)) and are required to discharge the proof obligation ($F \wedge E \models O$). Ideally, F should be sufficient to discharge O, but missing or insufficient specifications often results in F being too weak to prove O leading to false positives (alarms) in tools. So, work in applying abductive reasoning to static verification [Blackshear and Lahiri, 2013; Calcagno et al., 2009; Chandra et al., 2009; Das et al., 2015; Dillig et al., 2012] aims to compute E in order to prioritize—with minimal human intervention—verification failures caused by bugs in a program and de-emphasize false positives (alarms) caused by missing or incomplete specifications. In angelic verification [Blackshear and Lahiri, 2013; Das et al., 2015] and the work by Calcagno et al. [2009], entire specifications, such as preconditions, postconditions, and loop invariants, are generated as explanatory hypotheses. Dillig et al. [2012] instead compute smaller, intermediate formulas as explanatory hypotheses.

Similar to prior abductive reasoning work [Blackshear and Lahiri, 2013; Calcagno et al., 2009; Das et al., 2015; Dillig et al., 2012], Gradual C0's static system reasons around missing or incomplete specifications to compute facts F as part of imprecise formulas $? \wedge F$. At proof obligations, we approximate the weakest formula that can replace $?$ in $? \wedge F \models O$ and SAT($? \wedge F$) successfully. So, like Dillig et al. [2012] we compute intermediate explanatory hypotheses rather than whole specifications like Blackshear and Lahiri [2013], Calcagno et al. [2009], and Das et al. [2015]. But, rather than relying on users to validate generated hypotheses [Blackshear and Lahiri, 2013; Calcagno et al., 2009; Das et al., 2015; Dillig et al., 2012], we check their correctness at run time. This significantly simplifies their computation—since they do not need to be human readable and can statically mark code as unreachable—and allows Gradual C0 to be sound (prior abduction work is not).

Dynamic Verification. Meyer [1988] introduced the Eiffel language, which automatically performs dynamic verification of pre- and postconditions and class invariants in first-order logic. Nguyen et al. [2008] extended dynamic verification to support separation logic assertions. More recently, Agten et al. [2015] applied dynamic checking at the boundaries between statically verified and unverified code to guarantee that no assertion failures or invalid memory accesses occur at run time in any verified code. Their approach improved on the approach by Nguyen et al. [2008] in terms of performance by allowing unverified code to read arbitrary memory. Further, unlike Nguyen et al. [2008], the approach by Agten et al. [2015] only needs access to verified code rather than the entire

codebase. As with the work by Nguyen et al. [2008], Gradual C0 supports dynamic verification of ownership and first-order logic. Gradual C0 additionally supports run-time checking of recursive predicates. Similarly to Agten et al. [2015], Gradual C0 applies dynamic checking at the boundaries between verified and unverified code to protect verified code. However, in Gradual C0 unverified code must be accessible to the verifier as it is gradually verified as well. Future work in gradual verification should incorporate insights from the work by Agten et al. [2015] to avoid requiring entire codebases for verification and to improve verification performance.

Hybrid Verification. Another closely related work is soft contract verification [Nguyen et al., 2014], which verifies dynamic contracts statically where possible and dynamically where necessary by utilizing symbolic execution. This hybrid technique does not rely on a notion of *precision*, which is central to gradual approaches and their metatheory [Siek et al., 2015]. Nguyen et al. [2014] use symbolic execution results directly to discharge proof obligations where possible, while Gradual C0 strengthens symbolic execution results to discharge proof obligations adhering to the theory of imprecise formulas from Wise et al. [2020]. Further, the work by Nguyen et al. [2014] is targeted at dynamic functional languages, while our work focuses on imperative languages.

7 Conclusion

Gradual verification is a promising approach to supporting incrementality and enhance adoptability of program verification. Users can focus on specifying and verifying the most important properties and components of their systems and get immediate feedback about the consistency of their specifications and the correctness of their code. By relying on symbolic execution, Gradual C0 overcomes several limitations of prior work on gradual verification of recursive, heap-manipulating programs. The experimental results show that our approach can reduce overhead significantly compared to purely dynamic checking and confirms performance trends speculated in prior work.

Limitations and Future Work. However, more work remains to extend gradual verification (and Gradual C0) to the expressiveness of state-of-the-art static program verifiers, such as Viper. In particular, Gradual C0 lacks support for specifications containing quantification, pure functions, unfoldings, magic wands (lemmas), and fractional permissions. Pure functions and unfoldings are natural extensions to the work presented in this article: Pure functions can be treated similarly to predicates and nonpure functions, and unfoldings are semantically similar to an unfold, assert, and fold sequence. As seen in Section 5.5, such constructs are convenient for verifying recursive heap data structures. Quantification provides a natural way to specify properties about arrays (and thus strings) [Müller et al., 2016], magic wands eliminate the need to specify lemmas when converting between multiple views of the same structure [Müller et al., 2016], and fractional permissions aide in the verification of concurrent programs [Boyland, 2003]. However, these features require nontrivial extensions to current gradual verification theory; for example, it is not clear what ? means under a quantifier nor what permissions may be transferred to threads when contracts or invariants are underspecified. So, we leave such extensions to future work and provide Gradual C0 as a solid foundation for such work.

Additionally, we did not build any front-ends for Gradual Viper beyond the C0 one, so we did not concretely evaluate the extensibility of our design to other languages. However, Gradual Viper's intermediate language is a core subset of Viper's language, which supports front-ends for imperative languages such as Java, Rust, and Python. Furthermore, we were able to encode run-time checks in the much less expressive C0 language, so any more expressive language (like Java, Rust, and Python) would also be able to encode them. Therefore, developing front-ends for a core subset of such imperative languages should be a straightforward process, and, as the expressiveness of Gradual C0's specification language increases to match Viper's, then the gradually verified core for these languages can also increase correspondingly. As such, we leave these explorations to future work.

Finally, while our empirical study (Section 5) revealed interesting trends for Gradual C0 (and gradual verification), such as significant performance improvements over dynamic verification, our study has limitations. In particular, our test programs (while sufficiently complex) are smaller in size and focused on recursive heap data structures, and the dynamic verification approach we compared Gradual C0 against is unoptimized. We leave confirmation of how broadly the aforementioned trends apply to future work. Our study showed that Gradual C0's run-time checking strategies for owned fields and recursive predicates were significant causes of performance degradation and require further optimizations. Additionally, more aspects of Gradual C0's design, such as Gradual C0's approach for checking branch conditions at run time, should be studied more thoroughly in future work for the impact they have on Gradual C0's run-time performance.

Despite the aforementioned limitations, we believe the symbolic-execution approach to gradual verification presented in this article and implemented in Gradual C0 provides a solid foundation for future work in gradual verification and shows promise to make verification more adoptable in software development practice.

Acknowledgments

We thank the reviewers across multiple rounds of submission (*OOPSLA'22*, *OOPSLA'23*, and *TOPLAS*) for their feedback on our article. Our article has been much improved, thanks to the detailed feedback we received.

References

Pieter Agten, Bart Jacobs, and Frank Piessens. 2015. Sound modular verification of C code executing in an unverified context. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, 581–594. DOI: https://doi.org/10.1145/2676726.2676972

Rob Arnold. 2010. *C0, an Imperative Programming Language for Novice Computer Scientists*. Master's thesis. Department of Computer Science, Carnegie Mellon University.

Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual program verification. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 25–46.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2006. SmallFoot: Modular automatic assertion checking with separation logic. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO '06)*. Revised Lectures 4, Springer, 115–137.

Sam Blackshear and Shuvendu K. Lahiri. 2013. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 209–218.

John Boyland. 2003. Checking interference with fractional permissions. In *Proceedings of the International Static Analysis Symposium*. Springer, 55–72.

Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 289–300.

John Peter Campora, Sheng Chen, and Eric Walkingshaw. 2018. Casts and costs: Harmonizing safety and performance in gradual typing. *PACM on Programming Languages* 2, ICFP (Sept. 2018), 98:1–98:30.

Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: A powerful approach to weakest preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 363–374.

Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. 2015. Angelic verification: Precise verification modulo unknowns. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV '15)*, Proceedings, Part I 27. Springer, 324–342.

Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* 18, 8 (Aug. 1975), 453–457.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated error diagnosis using abductive inference. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*, 181–192. DOI: https://doi.org/10.1145/2254064.2254087

Dino Distefano and Matthew J. Parkinson. 2008. JStar: Towards practical verification for Java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA '08)*, 213–226. DOI: https://doi.org/10.1145/1449764.1449782

Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019. Approximate normalization for gradual dependent types. *Proceedings of the ACM on Programming Languages* 3, ICFP, Article 88 (Jul. 2019), 30 pages. DOI: https://doi.org/10.1145/3341692

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible contracts: Fixing a pathology of gradual typing. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 133 (Oct. 2018), 27 pages. DOI: https://doi.org/10.1145/3276503

Carlo Alberto Furia and Bertrand Meyer. 2010. Inferring loop invariants using postconditions. In *Fields of Logic and Computation*. Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig (Eds.), Lecture Notes in Computer Science, Vol 6300, Springer, 277–300. Retrieved from https://link.springer.com/chapter/10.1007/978-3-642-15025-8_15

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (Jun. 2010), 167–189.

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 *Proceedings of the NASA Formal Methods Symposium*. Springer, 41–55.

James C. King. 1976. Symbolic execution and program testing. *Communications of the ACM* 19, 7 (Jul. 1976), 385–394.

Nico Lehmann and Éric Tanter. 2017. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '17)*, 775–788.

K. Rustan M. Leino, Peter Müller, and Jan Smans. 2009. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*. Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri (Eds.), Lecture Notes in Computer Science, Vol. 5705, Springer, 195–222. Retrieved from https://link.springer.com/chapter/10.1007/978-3-642-03829-7_7

Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 2022. Gradualizing the calculus of inductive constructions. *ACM Transactions on Programming Languages and Systems* 44, 2 (Jun. 2022), 1–82. DOI: https://doi.org/10.1145/3495528

Bertrand Meyer. 1988. Eiffel: A language and environment for software engineering. *Journal of Systems and Software* 8, 3 (1988), 199–246.

Fabian Muehlboeck and Ross Tate. 2017. Sound gradual typing is nominally alive and well. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 56 (Oct. 2017), 30 pages. DOI: https://doi.org/10.1145/3133880

P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A verification infrastructure for permission-based reasoning. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '16)* (LNCS, Vol. 9583). B. Jobstmann and K. R. M. Leino (Eds.), Springer-Verlag, 41–62.

Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. 2008. Runtime checking for separation logic. In *Proceedings of the International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 203–217.

Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft contract verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, 139–152. DOI: https://doi.org/10.1145/2628136.2628156

Matthew Parkinson and Gavin Bierman. 2005. Separation logic and abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, 247–258. DOI: https://doi.org/10.1145/1040305.1040326

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74.

Malte H. Schwerhoff. 2016. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. Ph.D. Dissertation. ETH Zurich.

Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, Vol. 6, 81–92.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined criteria for gradual typing. In *Proceedings of the 1st Summit on Advances in Programming Languages (SNAPL '15), LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 274–293.

Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer, 148–172.

Alexander J. Summers and Sophia Drossopoulou. 2013. A formal semantics for isorecursive and equirecursive state abstractions. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer, 129–153.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*, 456–468. DOI: https://doi.org/10.1145/2837614.2837630

Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual verification of recursive heap data structures. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.

Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2024. Sound gradual verification with symbolic execution. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2547–2576.

Appendix A

In eval-p (Figure A2), a special case (highlighted in blue) for unfold statements is added that creates run-time checks for field accesses in the unfolded predicate's body. This case ensures soundness when introducing branch condition variables in C0 programs during run-time verification. In our implementation of Gradual C0, these checks are optimized further as they are only produced for branch conditions in the predicate body rather than for the whole body.

A.1 Diff and Translate

The DIFF (Figure A4) function finds a minimal run-time check from an optimistically asserted formula containing statically known information. It accomplishes this by first performing a standard transformation to CNF on the optimistically asserted formula, to extract the maximal number of top level conjuncts. It then attempts to call *check()* on each conjunct; it accumulates each conjunct for which the call does not succeed. The set of conjuncts which could not be statically discharged are returned as the final check.

$$\text{pc-add}(\pi,\ t) : \Pi \longrightarrow \mathit{SVALUE} \longrightarrow \Pi$$
$$\text{pc-push}(\pi,\ id,\ bc) : \Pi \longrightarrow \mathit{ID} \longrightarrow \mathit{SVALUE} \longrightarrow \Pi$$
$$\text{pc-all}(\pi) : \Pi \longrightarrow \mathcal{P}(\mathit{SVALUE})$$

$$\text{pc-add}(\pi,\ t) = \texttt{Let}\ (id,\ bc,\ pcs) :: \texttt{suffix match}\ \pi$$
$$(id,\ bc,\ pcs \cup \{t\}) :: \texttt{suffix}$$
$$\text{pc-push}(\pi,\ id,\ bc) = (id,\ bc,\ \emptyset) :: \pi$$
$$\text{pc-all}(\pi) = \text{foldl}(\pi,\ \emptyset,\ (\lambda\ (id_i,\ bc_i,\ pcs_i),\ all_i\ .\ all_i \cup \{bc_i\} \cup pcs_i))$$

Fig. A1. Path condition helper functions.

$$\text{eval-p}(\sigma,\ e,\ Q) : \Sigma \longrightarrow \mathit{EXPR} \longrightarrow (\Sigma \longrightarrow \mathit{SVALUE} \longrightarrow \mathit{RESULT}) \longrightarrow \mathit{RESULT}$$
$$\text{eval-p}(\sigma,\ t,\ Q) = Q(\sigma,\ t)$$
$$\text{eval-p}(\sigma,\ x,\ Q) = Q(\sigma,\ \sigma.\gamma(x))$$
$$\text{eval-p}(\sigma_1,\ op(\overline{e}),\ Q) = \text{eval-p}(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .\ Q(\sigma_2,\ op'(\overline{t}))))$$
$$\begin{aligned}
\text{eval-p}(\sigma_1,\ e.f,\ Q) = \text{eval-p}(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\ & \\
& \texttt{if}\ (\exists\ f(r;\ \delta) \in \sigma_2.h\ .\ \text{check}(\sigma_2.\pi,\ r = t))\ \texttt{then} \\
& \quad Q(\sigma_2,\ \delta) \\
& \texttt{else if}\ (\exists\ f(r;\ \delta) \in \sigma_2.h_?\ .\ \text{check}(\sigma_2.\pi,\ r = t))\ \texttt{then} \\
& \quad Q(\sigma_2,\ \delta) \\
& \texttt{else if}\ (\sigma_2.\text{isImprecise})\ \texttt{then} \\
& \quad \texttt{if}\ (\sigma_2.\mathcal{R}.\text{origin} = (_,\ \texttt{unfold}\ \textbf{acc}(_),\ _))\ \texttt{then} \\
& \quad\quad e_t\ :=\ \text{translate}(\sigma_2,\ t) \\
& \quad\quad \mathcal{R}'\ :=\ \text{addcheck}(\sigma_2.\mathcal{R},\ e.f,\ \textbf{acc}(e_t.f)) \\
& \quad \texttt{else} \\
& \quad\quad \mathcal{R}'\ :=\ \sigma_2.\mathcal{R} \\
& \quad \delta\ :=\ \text{fresh} \\
& \quad Q(\sigma_2\{\ h_?\ :=\ \sigma_2.h_? \cup f(t;\ \delta),\ \pi\ :=\ \text{pc-add}(\sigma_2.\pi,\ \{t \neq \texttt{null}\}),\ \mathcal{R}\ :=\ \mathcal{R}'\ \},\ \delta) \\
& \texttt{else}\ \text{failure}())
\end{aligned}$$

Fig. A2. Rules for symbolically executing expressions without introducing run-time checks (except for a special case for unfold).

$$
\begin{aligned}
&\text{eval-c}(\sigma,\ e,\ Q) : \Sigma \longrightarrow \mathit{EXPR} \longrightarrow (\Sigma \longrightarrow \mathit{SVALUE} \longrightarrow \mathit{RESULT}) \longrightarrow \mathit{RESULT}\\
&\text{eval-c}(\sigma,\ t,\ Q) = Q(\sigma,\ t)\\
&\text{eval-c}(\sigma,\ x,\ Q) = Q(\sigma,\ \sigma.\gamma(x))\\
&\text{eval-c}(\sigma_1,\ op(\overline{e}),\ Q) = \text{eval-c}(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .\ Q(\sigma_2,\ op'(\overline{t}))))\\
&\text{eval-c}(\sigma_1,\ e.f,\ Q) = \text{eval-c}(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\\
&\qquad \texttt{if}\ (\exists\ f(r;\ \delta) \in \sigma_2.h\ .\ \text{check}(\sigma_2.\pi,\ r = t))\ \texttt{then}\\
&\qquad\quad Q(\sigma_2,\ \delta)\\
&\qquad \texttt{else if}\ (\exists\ f(r;\ \delta) \in \sigma_2.h_?\ .\ \text{check}(\sigma_2.\pi,\ r = t))\ \texttt{then}\\
&\qquad\quad Q(\sigma_2,\ \delta)\\
&\qquad \texttt{else if}\ (\sigma_2.\text{isImprecise})\ \texttt{then}\\
&\qquad\quad \mathit{res},\ \_\ :=\ \text{assert}(\sigma_2.\text{isImprecise},\ \sigma_2.\pi,\ t \neq \text{null})\\
&\qquad\quad e_t\ :=\ \text{translate}(\sigma_2,\ t)\\
&\qquad\quad \mathcal{R}'\ :=\ \text{addcheck}(\sigma_2.\mathcal{R},\ e.f,\ \mathbf{acc}(e_t.f))\\
&\qquad\quad \mathit{res} \wedge Q(\sigma_2\{\ \mathcal{R}\ :=\ \mathcal{R}'\ \},\ \text{fresh})\\
&\qquad \texttt{else}\ \text{failure}())
\end{aligned}
$$

Fig. A3. Rules for symbolically executing expressions without modifying the optimistic heap and path condition.

Algorithm 1 Generating minimal checks

```
1: function Diff(φ)
2:     conjuncts ← CNF(φ)
3:     φ′ ← ∅
4:     for c ← conjuncts do
5:         if !check(c) then
6:             φ′ ← φ′ + c
7:         end if
8:     end for
9:     return φ′
10: end function
```

Fig. A4. Algorithm for computing the diff between two symbolic values.

The TRANSLATE (Figure A5) function lifts symbolic values to concrete values. Most symbolic values are directly translated to their concrete counterparts via recursive descent; the exception is variables, whose concrete values must be reconstructed by searching the program state known by the verifier. This is done by retrieving the states of the symbolic store, which contains mappings from concrete variables to symbolic variables, and the heap, which contains field and predicate permissions. When TRANSLATE encounters a symbolic variable, it first retrieves all possible aliasing information from Gradual Viper's state. This includes all variables known to be equivalent to the translation target according to the path condition and the heap. If the translation target or one of its aliases exists as a value in the symbolic store, then the translator finds a key corresponding to it in the store and returns it. Note that multiple valid keys may exist for a particular symbolic variable, because Gradual Viper may have determined that multiple concrete values are equivalent at a particular program point. If the translation target is a field, then only the top level receiver (the variable on which fields are being accessed) or one of its aliases will exist in the store. The fields being accessed are resolved by mapping their corresponding heap entries, or any aliased heap entries, to a value in the symbolic store, and resolving the store entry as described. In particular contexts, TRANSLATE may be asked to translate a precondition for a method call, or a predicate

Algorithm 2 Variable resolution procedure

```
1:  function Translate-var(s, v)
2:      store ← ∅
3:      if s.oldStore then
4:          store ← s.oldStore
5:      else
6:          store ← s.store
7:      end if
8:      aliasList ← aliases(v, s.pathConditions + +s.heap + +s.optimisticHeap)
9:      heap ← s.heap + +s.optimisticHeap
10:     outputs ← ∅
11:     for v ← aliasList do
12:         if c ← store.lookup(v) then
13:             outputs ← outputs + c
14:         else
15:             if h ← heap.lookup(v)&&c ← store.lookup(h) then
16:                 outputs ← outputs + c
17:             end if
18:         end if
19:     end for
20:     return selectLongest(outputs)
21: end function
```

Fig. A5. Translate's procedure for resolving variables.

body for an (un)fold statement. In these cases, an old store attached to the current symbolic state as described in Section 4.3.7 is retrieved, and its symbolic store and heap are used for translation. This causes variables in a precondition or predicate to be resolved to their concrete values at the call site, or site of unfolding. This enables run-time checks produced via translate to be straightforwardly emitted to the frontend. The portion of translate related to translating variables is shown in Figure A5.

A.2 Symbolic Production of Formulas

The rules for produce are given in Figure 12. Essentially, produce takes a formula and snapshot δ (mirroring the structure of the formula) and adds the information in the formula to the symbolic state, which is then returned to the continuation Q. An imprecise formula ? && ϕ has its static part ϕ produced into the current state σ alongside $\texttt{second}(\delta)$. Note the snapshot δ for an imprecise formula looks like $(unit, \texttt{second}(\delta))$ where $unit$ is the snapshot for ? and $\texttt{second}(\delta)$ is the snapshot for ϕ. An imprecise formula also turns σ imprecise to produce the unknown information represented by ? into σ. For example, if the state is represented by the formula θ, then this rule results in ? && θ && ϕ. A symbolic value t is produced into the path condition of the current state σ. Also, the snapshot δ for t must be $unit$, so this fact is also stored in σ's path condition. Then, σ is passed to Q.

The produce rule for expression e first evaluates e to its symbolic value t using eval-p. Then, t is produced into the path condition of the current state σ_2 using the aforementioned symbolic value rule. Imprecision in the symbolic state can always provide accessibility predicates for fields also in the state. Therefore, when fields in e are added to an imprecise state, heap chunks for those fields do not have to already be in the state, e.g., the state ? && true becomes ? && true && e. This

functionality is permitted by eval-p. Similarly, an imprecise formula always provides accessibility predicates for fields in its static part, e.g., the state true and produced formula ? && e results in the state ? && true && e. The goal of produce is not to assert information in the state, but rather add information to the state. So we reduce run-time overhead by ensuring no run-time checks are produced by produce even for verifying field accesses.

The rules for producing field and predicate accessibility predicates into the state σ_1 operate in a very similar manner. Thus, we will focus on the rule for fields only. The field $e.f$ in $\mathbf{acc}(e.f)$ first has its receiver e evaluated to t by eval-p, resulting in σ_2. Then, using the parameter δ a fresh heap chunk $f(t;\ \delta)$ is created and added to σ_2's heap h, which represents $\mathbf{acc}(e.f)$ in the state. Note, the disjoint union $\uplus$ ensures $f(t;\ \delta)$ is not already in the heap before adding $f(t;\ \delta)$ in there. If the chunk is in the heap, then verification will fail. Further, $\mathbf{acc}(e.f)$ implies $e \neq \texttt{null}$ and so that fact is recorded in σ_2's path condition as $t \neq \texttt{null}$.

When the separating conjunction ϕ_1 && ϕ_2 is produced, ϕ_1 is first produced and then afterwards ϕ_2 is produced into the resulting symbolic state. Note that the snapshot δ is split between the two formulas using $\texttt{first}(\delta)$ and $\texttt{second}(\delta)$. Finally, to produce a conditional, Gradual Viper branches on the symbolic value t for the condition e splitting execution along two different paths. Along one path only the true branch ϕ_1 is produced into the state, and along the other path only the false branch ϕ_2 is produced. Both paths follow the continuation to the end of its execution. More details about branching are provided next, as we describe Gradual Viper's branch function.

The branch function in Figure 13 is used to split the symbolic execution into two paths in a number of places in our algorithm: during the production or consumption of logical conditionals and during the execution of if statements. One path (Q_t) is taken under the assumption that the parameter t is true, and the other ($Q_{\neg t}$) is taken under the assumption that t is false. For each path, a branch condition corresponding to the assumption made is added to $\sigma.\mathcal{R}$, as highlighted in blue. Additionally, paths may be pruned using check when Gradual Viper knows for certain a path is infeasible (the assumption about t would contradict the current path conditions). Now, normally, if either of the two paths fail verification, then branch marks verification as failed ($\wedge$ the results). This is still true when σ (the current state) is precise. However, when σ is imprecise, branch can be more permissive as highlighted in yellow. If verification fails on one of two paths only (one success, one failure), then branch returns success ($\vee$ the results). In this case, a run-time check (highlighted in blue) is added to $\mathfrak{R}$ to force run-time execution down the success path only. Of course, two failures result in failure and two successes result in success ($\vee$ the results). No run-time checks are produced in these cases, as neither path can be soundly taken or both paths can be soundly taken at run time, respectively. Note that Gradual Viper being flexible in the aforementioned way is critical to adhering to the gradual guarantee at branch points.

A.3 Symbolic Consumption of Formulas

The goals of consume are 3-fold: (1) given a symbolic state σ and formula $\widetilde{\phi}$ check whether $\widetilde{\phi}$ is established by σ, i.e., $\widetilde{\phi_\sigma} \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi}$ where $\widetilde{\phi_\sigma}$ is the formula which represents the state σ, (2) produce and collect run-time checks that are minimally sufficient for σ to establish $\widetilde{\phi}$ soundly, and (3) remove accessibility predicates and predicates that are asserted in $\widetilde{\phi}$ from σ. Note that $\mathrel{\widetilde{\Rightarrow}}$ is the consistent implication formally defined by Wise et al. [2020]. The rules for consume are given in Figure A6.

The consume function always begins by consolidating information across the given heap $\sigma_1.h$ and path condition $\sigma_1.\pi$. The invariant on the heap $\sigma_1.h$ ensures all heap chunks in $\sigma_1.h$ are separated in memory, e.g., $f(x;\ \delta_1) \in \sigma_1.h$ and $f(y;\ \delta_2) \in \sigma_1.h$ implies $x \neq y$. Similarly, $f(x;\ \delta_1) \in \sigma_1.h$ implies $x \neq \texttt{null}$. Therefore, such information is added to the path condition $\sigma_1.\pi$ during consolidation.

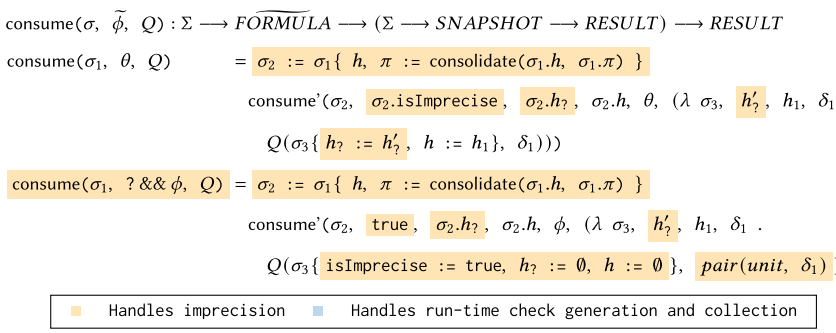

Fig. A6. Rules for symbolically consuming formulas (1/3), (2/3), and (3/3).

Further, consolidate ensures $\sigma_1.h$ and $\sigma_1.\pi$ are consistent, i.e., do not contain contradictory information. We use the definition of consolidate from Schwerhoff [2016], without repeating it here.

After consolidation, consume calls a helper function consume', which performs the major functionality of consume. Along with the state σ_2 from consolidation, consume' accepts a Boolean flag, optimistic heap $\sigma_2.h_?$, regular heap $\sigma_2.h$, the formula to be consumed $\widetilde{\phi}$, and a continuation. The Boolean flag sent to consume' controls how σ_2 provides access to fields in $\widetilde{\phi}$. When $\widetilde{\phi}$ is precise (is θ), then σ_2 provides access to fields in θ through heap chunks or imprecision where applicable. Therefore, in this case, the Boolean flag is set to σ_2.`isImprecise`. However, when $\widetilde{\phi}$ is imprecise (i.e., ? && ϕ), then the Boolean flag is set to `true` so access to fields in $\widetilde{\phi}$ is always justified: first by σ_2 if applicable and second by imprecision in $\widetilde{\phi}$. Copies of the optimistic heap $\sigma_2.h_?$ and regular heap $\sigma_2.h$ are sent to consume' where heap chunks from $\widetilde{\phi}$ are removed from them. If consume' succeeds, then when $\widetilde{\phi}$ is precise execution continues with the residual heap chunks. When $\widetilde{\phi}$ is imprecise execution continues with empty heaps, because $\widetilde{\phi}$ may require and assert any heap chunk in σ_2. Residual heap chunks are instead represented by imprecision, i.e., execution continues with an imprecise state. Finally, consume' also sends snapshots collected for removed heap chunks to the continuation.

Rules for consume' can also be found in Figure A6. Cases for expressions e, the separating conjunction ϕ_1 && ϕ_2, and logical conditionals e ? $\phi_1 : \phi_2$ are straightforward. Expressions are evaluated to symbolic values that are then consumed with the corresponding rule. In a separating conjunction, ϕ_1 is consumed first, then afterward ϕ_2 is consumed. The rule for logical conditionals evaluates the condition e to a symbolic value, and then uses the branch function to consume ϕ_1 and ϕ_2 along different execution paths. The case for $\mathbf{acc}(p(\overline{e}))$ is also very similar to the case for $\mathbf{acc}(e.f)$ that we discuss later in this section.

When a symbolic value t is consumed, the current state σ must establish t, i.e., $\sigma \widetilde{\Rightarrow} t$, or verification fails. The assert function (defined in Figure A8) implements this functionality. In particular, assert returns `success()` when π can statically prove t or when σ is imprecise and t does not contradict constraints in π—here, t is optimistically assumed to be true. Otherwise, assert returns `failure()`. When assert succeeds, it also returns a set of symbolic values $\overline{t}$ that are residuals of t that cannot be proved statically by π. If t is proven entirely statically, then assert returns `true`. A run-time check is created for the residuals $\overline{t}$ and is added to σ to be passed to the continuation Q. Note that translate is used to create an expression from $\overline{t}$ that can be evaluated at run time. Further, the location e is the expression that evaluates to t and is passed to consume' alongside t. The heaps $h_?$ and h are passed unmodified to Q alongside the snapshot $unit$.

The consume' rule for accessibility predicates $\mathbf{acc}(e.f)$, first evaluates the receiver e to t using eval-c, the current state σ_1, and the parameter $f_?$. The parameter $f_?$ is the Boolean flag mentioned

Fig. A6. (Continued).

previously. Assigning $f_?$ to σ_1.isImprecise during evaluation allows $f_?$ to control whether or not imprecision verifies field accesses. This occurs in all of the consume' rules where expressions and thus fields are evaluated. After evaluation, the isImprecise field is reset resulting in σ_3, and assert is used to ensure the receiver t is non-null. If $t \neq$ null is optimistically true, a run-time check for $t \neq$ null at location $\mathbf{acc}(e.f)$ is created and added to $\sigma_3.\mathcal{R}$. Next, heap-rem-acc is used to

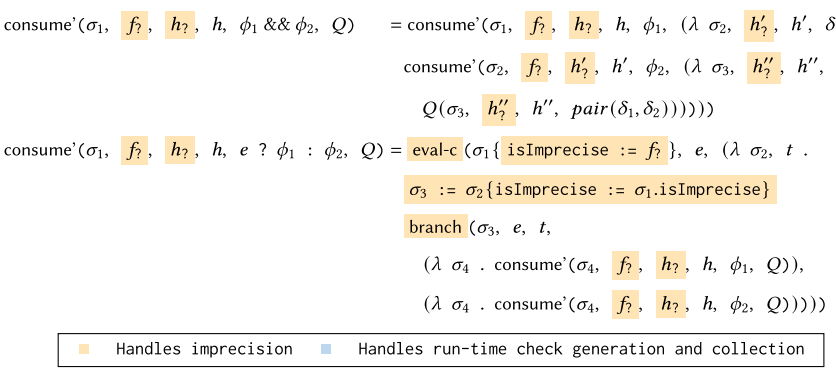

Fig. A6. (Continued).

remove the heap chunks from heap h that overlap with or may potentially overlap with $\mathbf{acc}(e.f)$ in memory. The heap-rem-acc function is formally defined alongside a similar function for predicates (heap-rem-pred) in Figure A7. If a field chunk is not statically proven to be disjoint from $\mathbf{acc}(e.f)$, then it is removed. Further, since predicates are opaque, Gradual Viper cannot tell whether or not their predicate bodies overlap with $\mathbf{acc}(e.f)$. Therefore, predicate chunks are almost always considered to potentially overlap with $\mathbf{acc}(e.f)$. The only time this is not the case is if they both exist in the heap h, which ensures its heap chunks do not overlap in memory. The heap-rem-acc function also checks that $\mathbf{acc}(e.f)$ has a corresponding heap chunk in h. If so, its snapshot δ_1 is returned and b_1 is assigned true. Otherwise, a fresh snapshot is returned with false. If the current state σ_3 is imprecise, then heap chunks are similarly removed from $h_?$ and $\mathbf{acc}(e.f)$ is checked for existence in $h_?$. If a field chunk for $\mathbf{acc}(e.f)$ is not found in either heap, then a run-time check is generated for it and passed to the continuation Q alongside the two heaps after removal and $\mathbf{acc}(e.f)$'s snapshot. Without imprecision, consume' will fail when a field chunk for $\mathbf{acc}(e.f)$ is not found in h.

A.4 Symbolic Execution of Statements

The exec rules (Figure A9) for sequence statements, variable declarations and assignments, allocations, and if statements are pretty much unchanged from Viper. The only difference is that Gradual Viper's versions of eval, produce, branch, and consume (defined previously) are used instead of Viper's. Statements in a sequence are executed one after another, and variable declarations introduce a fresh symbolic value for the variable into the state. Variable assignments evaluate the right-hand side to a symbolic value and update the variable in the symbolic store with the result. Allocations produce fresh heap chunks for fields into the state. Finally, if statements have their condition evaluated and then branch is used to split execution along two paths to symbolically execute the true and false branches.

Symbolic execution of field assignments first evaluates the right-hand side expression e to the symbolic value t with the current state σ_1 and eval. Any field reads in e are either directly or optimistically verified using σ_1. Then, the resulting state σ_2 must establish write access to $x.f$ in consume, i.e., $\sigma_2 \overset{=}{\Rightarrow} \mathbf{acc}(x.f)$. The call to consume also removes the field chunk for $\mathbf{acc}(x.f)$ from σ_2 (if it is in there) resulting in σ_3. Therefore, the call to produce can safely add a fresh field chunk for $\mathbf{acc}(x.f)$ alongside $x.f = t$ to σ_3 before it is passed to the continuation Q. Under the hood, run-time checks are collected where required for soundness and passed to Q.

The exec rule for method calls similarly uses eval to evaluate the given args $\overline{e}$ to symbolic values $\overline{t}$, asserts the method's precondition $meth_{pre}$ holds in the current state, consumes the heap chunks

$$\texttt{heap-rem-pred(isImprecise, } h, \pi, p(\overline{t})\texttt{)} : \{\texttt{true}, \texttt{false}\} \longrightarrow \mathit{HEAP} \longrightarrow \Pi \longrightarrow \mathit{IDPREDICATE} \longrightarrow \mathit{HEAP} \times \mathit{SNAPSHOT} \times \{\texttt{true}, \texttt{false}\}$$

$$\texttt{heap-rem-acc(isImprecise, } h, \pi, f(t)\texttt{)} : \{\texttt{true}, \texttt{false}\} \longrightarrow \mathit{HEAP} \longrightarrow \Pi \longrightarrow \mathit{IDFIELD} \longrightarrow \mathit{HEAP} \times \mathit{SNAPSHOT} \times \{\texttt{true}, \texttt{false}\}$$

$$\begin{aligned}
&\texttt{heap-rem-pred(isImprecise, } h, \pi, p(\overline{t})\texttt{)} = \texttt{if } \exists\ (p(\overline{r};\ \delta) \in h\ .\ \texttt{check}(\pi,\ \textstyle\bigwedge \overline{t = r}))\ \texttt{then} \\
&\qquad (h \setminus \{p(\overline{r};\ \delta)\},\ \delta,\ \texttt{true}) \\
&\quad \texttt{else } (\emptyset,\ \texttt{fresh},\ \texttt{false}) \\
&\texttt{heap-rem-acc(isImprecise, } h, \pi, f(t)\texttt{)} = h' := \texttt{foldl}(h,\ \emptyset,\ (\lambda\ f_{src}(\overline{r};\ \delta),\ h_{dst}\ . \\
&\qquad \texttt{if } (\neg(|\overline{r}| = 1) \parallel \neg(f = f_{src}) \parallel \neg\texttt{check(isImprecise, } \pi,\ t = r\texttt{))}\ \texttt{then} \\
&\qquad\quad h_{dst} \cup f_{src}(\overline{r};\ \delta) \\
&\qquad \texttt{else } h_{dst})) \\
&\quad \texttt{if } \exists\ f(r;\ \delta) \in h\ .\ \texttt{check}(\pi,\ t = r)\ \texttt{then} \\
&\qquad (h',\ \delta,\ \texttt{true}) \\
&\quad \texttt{else} \\
&\qquad h' := \texttt{foldl}(h',\ \emptyset,\ (\lambda\ f_{src}(\overline{r};\ \delta),\ h_{dst}\ . \\
&\qquad\quad \texttt{if } (f_{src}(\overline{r};\ \delta) \texttt{ is a field chunk) then} \\
&\qquad\quad\quad h_{dst} \cup f_{src}(\overline{r};\ \delta) \\
&\qquad\quad \texttt{else } h_{dst})) \\
&\qquad (h',\ \texttt{fresh},\ \texttt{false})
\end{aligned}$$

Fig. A7. Heap remove function definitions.

$$\texttt{check}(\pi,\ t) : \Pi \longrightarrow \mathit{SVALUE} \longrightarrow \{\texttt{true}, \texttt{false}\}$$

$$\texttt{check(isImprecise, } \pi,\ t\texttt{)} : \{\texttt{true}, \texttt{false}\} \longrightarrow \Pi \longrightarrow \mathit{SVALUE} \longrightarrow \{\texttt{true}, \texttt{false}\} \times \mathit{SVALUE}^*$$

$$\texttt{assert(isImprecise, } \pi,\ t\texttt{)} : \{\texttt{true}, \texttt{false}\} \longrightarrow \Pi \longrightarrow \mathit{SVALUE} \longrightarrow \mathit{RESULT} \times \mathit{SVALUE}^*$$

$$\texttt{check}(\pi,\ t) = \text{pc-all}(\pi) \Rightarrow t$$

$$\texttt{check(}\ \texttt{isImprecise}\ \texttt{, } \pi,\ t\texttt{)} = \begin{cases} \texttt{true},\ \texttt{true} & \texttt{if check}(\pi,\ t) \\ \texttt{true},\ \texttt{diff}(\text{pc-all}(\pi), t) & \texttt{if (isImprecise} \wedge (\textstyle\bigwedge \text{pc-all}(\pi) \wedge t)_{\text{SAT}}\texttt{)} \\ \texttt{false},\ \emptyset & \texttt{otherwise} \end{cases}$$

$$\texttt{assert(}\ \texttt{isImprecise}\ \texttt{, } \pi,\ t\texttt{)} = \begin{cases} \texttt{success}(),\ \overline{t} & \texttt{if } (b = \texttt{true}) \texttt{ where } b,\ \overline{t} := \texttt{check(isImprecise, } \pi,\ t\texttt{)} \\ \texttt{failure}(),\ \emptyset & \texttt{otherwise} \end{cases}$$

Handles imprecision Handles run-time check generation and collection

Fig. A8. Check and assert function definitions.

in the precondition, and produces the method's postcondition $meth_{post}$ into the continuation. Run-time checks are also collected where necessary (under the hood) and passed to the continuation. Gradual Viper makes an exception when consuming preconditions at method calls (and loop invariants before entering loops), which can be seen in the `if-then` in the method call rule. If Gradual Viper determines the precondition (invariant) is equi-recursively imprecise (as defined by Figure A10), then it will conservatively remove all the heap chunks from both symbolic heaps after the consume. This exception ensures the static verification semantics in Gradual Viper lines up with the equi-recursive, dynamic verification semantics encoded by GVC0 in Section 4.4 such that Gradual C0 is sound. Note that the origin field of $\mathcal{R}$ is set to $\overline{z}$ `:=` $m(\overline{e})$ before consuming $meth_{pre}$ and reset to none after producing $meth_{post}$. Setting the origin indicates that run-time checks or branch conditions for $meth_{pre}$ or $meth_{post}$ should be attached to the method call statement rather than where they are declared. The origin arguments σ_2 and $\overline{t}$ are used to reverse the substitution

$\text{exec}(\sigma,\ s,\ Q) : \Sigma \longrightarrow \mathit{STMT} \longrightarrow (\Sigma \longrightarrow \mathit{RESULT}) \longrightarrow \mathit{RESULT}$

$\text{exec}(\sigma_1,\ s_1;\ s_2,\ Q) = \text{exec}(\sigma_1,\ s_1,\ (\lambda\ \sigma_2\ .\ \text{exec}(\sigma_2,\ s_2,\ Q)))$

$\text{exec}(\sigma,\ \texttt{var}\ x : T,\ Q) = Q(\sigma\{\gamma\ :=\ \text{havoc}(\sigma.\gamma,\ x)\})$

$\text{exec}(\sigma_1,\ x\ :=\ e,\ Q) = \text{eval}\,(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\ Q(\sigma_2\{\gamma\ :=\ \sigma_2.\gamma[x \mapsto t]\})))$

$\text{exec}(\sigma_1,\ x.f\ :=\ e,\ Q) = \text{eval}\,(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\ \text{consume}\,(\sigma_2,\ \mathbf{acc}(x.f),\ (\lambda\ \sigma_3,\ \_\ .$
$\text{produce}\,(\sigma_3,\ \mathbf{acc}(x.f)\ \&\&\ x.f = t,\ \mathit{pair}(\text{fresh},\ \mathit{unit}),\ Q)))))$

$\text{exec}(\sigma,\ x\ :=\ \text{new}(\overline{f}),\ Q) = \text{produce}\,(\sigma\{\gamma\ :=\ \text{havoc}(\sigma.\gamma,\ x)\},\ \overline{\mathbf{acc}(x.f)},\ \text{fresh},\ Q)$

$\text{exec}(\sigma_1,\ \overline{z}\ :=\ m(\overline{e}),\ Q) = \text{eval}\,(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .$
$\mathcal{R}'\ :=\ \sigma_2.\mathcal{R}\{\texttt{origin}\ :=\ (\sigma_2,\ \overline{z}\ :=\ m(\overline{e}),\ \overline{t})\}$
$\text{consume}\,(\sigma_2\{\mathcal{R}\ :=\ \mathcal{R}'\},\ \mathit{meth}_{pre}[\overline{\mathit{meth}_{args} \mapsto t}],\ (\lambda\ \sigma_3,\ \delta\ .$
$\texttt{if (equi-imp}(\mathit{meth}_{pre})\texttt{) then}$
$\sigma_4\ :=\ \sigma_3\{\texttt{isImprecise := true},\ h_?\ :=\ \emptyset,\ h\ :=\ \emptyset,$
$\gamma\ :=\ \text{havoc}(\sigma_3.\gamma,\ \overline{z})\}$
$\texttt{else}\ \ \sigma_4\ :=\ \sigma_3\{\gamma\ :=\ \text{havoc}(\sigma_3.\gamma,\ \overline{z})\}$
$\text{produce}\,(\sigma_4,\ \mathit{meth}_{post}[\overline{\mathit{meth}_{args} \mapsto t}][\overline{\mathit{meth}_{ret} \mapsto z}],\ \text{fresh},$
$(\lambda\ \sigma_5\ .\ Q(\sigma_5\{\mathcal{R}\ :=\ \sigma_5.\mathcal{R}\{\texttt{origin := none}\}\})))))))$

$\text{exec}(\sigma_1,\ \texttt{assert}\ \phi,\ Q) = \text{consume}\,(\sigma_1,\ \phi,\ (\lambda\ \sigma_2,\ \delta\ .$
$\text{well-formed}\,(\sigma_2,\ ?\ \&\&\ \phi,\ \delta,\ (\lambda\ \sigma_3\ .$
$Q(\sigma_1\{\pi\ :=\ \sigma_3.\pi,\ \mathcal{R}\ :=\ \sigma_3.\mathcal{R}\}))))))$

$\text{exec}(\sigma_1,\ \texttt{fold}\ \mathbf{acc}(p(\overline{e})),\ Q) = \text{eval}\,(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .$
$\mathcal{R}'\ :=\ \sigma_2.\mathcal{R}\{\texttt{origin}\ :=\ (\sigma_2,\ \texttt{fold}\ \mathbf{acc}(p(\overline{e})),\ \overline{t})\}$
$\text{consume}\,(\sigma_2\{\mathcal{R}\ :=\ \mathcal{R}'\},\ \mathit{pred}_{body}[\overline{\mathit{pred}_{args} \mapsto t}],\ (\lambda\ \sigma_3,\ \delta\ .$
$\text{produce}\,(\sigma_3\{\mathcal{R}\ :=\ \sigma_3.\mathcal{R}\{\texttt{origin := none}\}\},\ \mathbf{acc}(p(\overline{t})),\ \delta,$
$Q)))))$

$\text{exec}(\sigma_1,\ \texttt{unfold}\ \mathbf{acc}(p(\overline{e})),\ Q) = \text{eval}\,(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .$
$\mathcal{R}'\ :=\ \sigma_2.\mathcal{R}\{\texttt{origin}\ :=\ (\sigma_2,\ \texttt{unfold}\ \mathbf{acc}(p(\overline{e})),\ \overline{t})\}$
$\text{consume}\,(\sigma_2\{\mathcal{R}\ :=\ \mathcal{R}'\},\ \mathbf{acc}(p(\overline{t})),\ (\lambda\ \sigma_3,\ \delta\ .$
$\text{produce}\,(\sigma_3,\ \mathit{pred}_{body}[\overline{\mathit{pred}_{args} \mapsto t}],\ \delta,\ (\lambda\ \sigma_4\ .$
$Q(\sigma_4\{\mathcal{R}\ :=\ \sigma_4.\mathcal{R}\{\texttt{origin := none}\}\}))))))))$

$\text{exec}(\sigma_1,\ \texttt{if}\ (e)\ \{\ \mathit{stmt}_1\ \}\ \texttt{else}\ \{\ \mathit{stmt}_2\ \},\ Q) = \text{eval}\,(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .$
$\text{branch}\,(\sigma_2,\ e,\ t,\ (\lambda\ \sigma_3\ .\ \text{exec}(\sigma_3,\ \mathit{stmt}_1,\ Q)),\ (\lambda\ \sigma_3\ .\ \text{exec}(\sigma_3,\ \mathit{stmt}_2,\ Q)))))$

Handles imprecision Handles run-time check generation and collection

Fig. A9. Rules for symbolically executing program statements (1/2).

$[\overline{\mathit{meth}_{args} \mapsto t}]$ in run-time checks and branch conditions for meth_{pre} and meth_{post}. The rule for (un)folding predicates operates the same as for method calls where meth_{pre} is the predicate body (predicate instance) and meth_{post} is the predicate instance (predicate body). The `origin` is set to `fold` $\mathbf{acc}(p(\overline{e}))$ and `unfold` $\mathbf{acc}(p(\overline{e}))$, respectively.

In contrast, ϕ in `assert` ϕ maintains a none origin field, because ϕ's use and declaration align at the same program location `assert` ϕ. The assert rule relies on consume to assert ϕ holds in the current state σ_1. If the consume succeeds, the state σ_1 is passed to the continuation nearly

```
exec(σ1, while (e) invariant φ̃ { stmt }, Q) =   γ2 := havoc(σ1.γ, x̄)
    res_body := well-formed(
      σ1{isImprecise := false, h? := ∅, h := ∅, γ := γ2,
        R := σ1.R{origin := (σ1, while (e) invariant φ̃ { stmt }, beginning)},
      φ̃ && e, fresh, (λ σ3{R := σ3.R{origin := none}} .
        exec(σ3, stmt, (λ σ4 .
          eval(σ4{R := σ4.R{origin := (σ4, while (e) invariant φ̃ { stmt }, end)}},
            e, (λ σe, _ .
              consume(σ4{R := σ4.R{origin := σe.R.origin, rcs := σe.R.rcs}},
                φ̃, (λ σ5, _ . R := R ∪ σ5.R.rcs ; success()))))))))
    res_after := eval(σ1{R := σ1.R{origin := (σ1, while (e) invariant φ̃ { stmt }, before)}},
      e, (λ σe, _ . consume(
        σ1{R := σ1.R{origin := σe.R.origin, rcs := σe.R.rcs}},
        φ̃, (λ σ2, _ .
          if (equi-imp(φ̃)) then
            σ3 := σ2{isImprecise := true, h? := ∅, h := ∅, γ := γ2}
          else σ3 := σ2{γ := γ2}
          produce(σ3{σ3.R{origin := (σ3, while (e) invariant φ̃ { stmt }, after)}},
            φ̃ && !e, fresh, Q)))))
    if (σ1.isImprecise) then
      if (¬res_body ∧ res_after) then
        R′ := addcheck(σ1.R, e, ¬e)
        R := R ∪ R′.rcs.last
      (¬res_body ∨ res_after) ∧ (res_body ∨ res_after)
    else
      res_body ∧ res_after
  where x̄ are variables modified by the loop body
```

Handles imprecision Handles run-time check generation and collection

Fig. A9. (Continued).

unmodified. Path condition constraints from ϕ hold in σ_1 either directly or optimistically. Therefore, these constraints are added to σ_1 to avoid producing run-time checks for them in later program statements. Run-time checks from the consume are also passed to the continuation. Note that ϕ is checked for well-formedness here (Figure A11). A formula is well-formed if it contains ? or accessibility predicates that verify access to the formula's fields (*self-framing*). Additionally, the formula cannot contain duplicate accessibility predicates or predicate instances. Finally, well-formed adds the formula's information to the given symbolic state. Here, ϕ does not need to be self-framed, and so it is joined with ? in the call to well-formed. ? verifies access to all of ϕ's fields.

Finally, while the while loop rule is the largest rule and looks fairly complex, it just combines ideas from other rules that are discussed in great detail in this section and from the branch rule described in Appendix A.2.

$$
\begin{aligned}
&\text{equi-imp}(\widetilde{\phi}) : \widetilde{FORMULA} \longrightarrow \{\texttt{true}, \texttt{false}\} \\
&\text{equi-imp}(? \;\&\&\; \phi) &&= \texttt{true} \\
&\text{equi-imp}(\phi_1 \;\&\&\; \phi_2) &&= \text{equi-imp}(\phi_1) \vee \text{equi-imp}(\phi_2) \\
&\text{equi-imp}(e \;?\; \phi_1 \;:\; \phi_2) &&= \text{equi-imp}(\phi_1) \vee \text{equi-imp}(\phi_2) \\
&\text{equi-imp}(\textbf{acc}(p(\overline{e}))) &&= \texttt{if } (p \in \texttt{VisitedPreds}) \texttt{ then} \\
&&&\quad \texttt{false} \\
&&&\texttt{else} \\
&&&\quad \texttt{VisitedPreds := VisitedPreds} \cup p \\
&&&\quad b \texttt{ := equi-imp}(pred_{body}) \\
&&&\quad \texttt{VisitedPreds := VisitedPreds} \setminus p \\
&&&\quad b \\
&\text{equi-imp}(\_) &&= \texttt{false}
\end{aligned}
$$

Fig. A10. Boolean function determining if a gradual formula is equi-recursively imprecise or not.

$$
\begin{aligned}
&\text{well-formed}(\sigma,\ \widetilde{\phi},\ \delta,\ Q) : \Sigma \longrightarrow \widetilde{FORMULA} \longrightarrow SNAPSHOT \longrightarrow (\Sigma \longrightarrow RESULT) \longrightarrow RESULT \\
&\text{well-formed}(\sigma_1,\ \widetilde{\phi},\ \delta,\ Q) = \text{produce}(\sigma_1,\ \widetilde{\phi},\ \delta,\ (\lambda\ \sigma_2\ . \\
&\qquad\qquad \text{produce}(\sigma_1\{\ \pi\ :=\ \sigma_2.\pi\ \},\ \widetilde{\phi},\ \delta,\ Q)))
\end{aligned}
$$

Fig. A11. Well-formed formula function definition.

A.5 Valid Program

A Gradual Viper program is valid if all of its method and predicate declarations are verified successfully as defined in Figure 16. In particular, a method m's declaration is verified first by checking well-formedness of m's precondition $meth_{pre}$ and postcondition $meth_{post}$ using the empty state σ_0 (well-formedness is described in Appendix A.4). Note, fresh symbolic values are created and added to σ_0 for m's argument variables $\overline{x}$ and return variables $\overline{y}$. If $meth_{pre}$ and $meth_{post}$ are well-formed, then the body of m ($meth_{body}$) is symbolically executed (Appendix A.4) starting with the symbolic state σ_1 containing $meth_{pre}$. Recall, well-formed additionally produces the formula that is being checked into the symbolic state. The symbolic state σ_2 is produced after the symbolic execution of $meth_{body}$. Then, $meth_{post}$ is checked for validity against σ_2, i.e., σ_2 must establish $meth_{post}$ (Appendix A.3). If $meth_{post}$ is established, then verification succeeds; and as a result, the run-time checks collected during verification are added to $\mathfrak{R}$ (highlighted in blue). A valid predicate p is simply valid if p's body $pred_{body}$ is well-formed. As before, fresh symbolic values are created for p's argument variables $\overline{x}$. Note, no run-time checks are added to $\mathfrak{R}$ here, because well-formedness checks do not produce any run-time checks.

Received 19 December 2023; revised 31 July 2024; accepted 24 October 2024