

# Gradual C0: Symbolic Execution for Efficient Gradual Verification

JENNA DIVINCENZO, Carnegie Mellon University, USA

IAN MCCORMACK, Carnegie Mellon University, USA

HEMANT GOUNI, University of Minnesota, USA

JACOB GORENBURG, Haverford College, USA

MONA ZHANG, Columbia University, USA

CONRAD ZIMMERMAN, Brown University, USA

JOSHUA SUNSHINE, Carnegie Mellon University, USA

ÉRIC TANTER, University of Chile, Chile

JONATHAN ALDRICH, Carnegie Mellon University, USA

Current static verification techniques have allowed users to specify and verify more code than ever before. 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 exhibit a pay-as-you-go cost model, which rewards users with increased static correctness guarantees and decreased dynamic checking as specifications are refined. In fact, all properties are checked dynamically regardless of any static guarantees, resulting in significant run-time overhead.

In this paper, we present the first gradual verifier for recursive heap data structures that can be used on real programs, called Gradual C0. Additionally, Gradual C0 reasons with symbolic execution and supports a pay-as-you-go cost model. Our approach addresses technical challenges related to symbolic execution with imprecise specifications, heap ownership, and that branches across program statements and specifications. Finally, we empirically evaluated the pay-as-you-go cost model of Gradual C0 and found that, on average, Gradual C0 decreases run-time overhead between 50-75% compared to the fully-dynamic approach used in prior work. Further, the worst-case scenarios for performance are predictable and avoidable.

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

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

## 1 INTRODUCTION

In recent years, research advances have introduced new ways to formally specify and verify software. *Separation logic*, introduced by Reynolds [2002], supports the static verification of the heap. Users can talk about the heap with the  $\mapsto$  operator and the separating conjunction  $*$ . The  $\mapsto$  operator asserts both ownership of a heap location and its value, e.g.  $x.f \mapsto 2$  states that the location  $x.f$  is uniquely owned and contains the value 2. Further, the separating conjunction ensures that two sub-heaps are distinct in memory. For example,  $x.f \mapsto 2 * y.f \mapsto 2$  states that the heap locations

\*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' addresses: Jenna DiVincenzo, Carnegie Mellon University, USA, jlwise@andrew.cmu.edu; Ian McCormack, Carnegie Mellon University, USA, icmccorm@cs.cmu.edu; Hemant Gouni, University of Minnesota, USA, hemant@hemantgouni.com; Jacob Gorenburg, Haverford College, USA, jgorenburg@haverford.edu; Mona Zhang, Columbia University, USA, mz2781@columbia.edu; Conrad Zimmerman, Brown University, USA, me@conradz.com; Joshua Sunshine, Carnegie Mellon University, USA, sunshine@cs.cmu.edu; Éric Tanter, University of Chile, Chile, etanter@dcc.uchile.cl; Jonathan Aldrich, Carnegie Mellon University, USA, jonathan.aldrich@cs.cmu.edu.

$x.f$  and  $y.f$  are distinct (i.e.  $x \neq y$ ), are each owned, and each contain the value 2. Years later, Smans et al. [2009] presented *implicit dynamic frames* (IDF) as an alternative to separation logic, which asserts ownership of a heap location and its value separately. Ownership is ensured through the use of an *accessibility predicate* e.g.  $\text{acc}(x.f)$ . Then,  $\text{acc}(x.f) * x.f == 2$  states that  $x.f$  is uniquely owned and contains the value 2.

Finally, Parkinson and Bierman [2005] and Smans et al. [2009] extended separation logic and IDF respectively to support *recursive abstract predicates* and thus recursive heap data structures, such as trees, lists, and graphs. *Abstract predicates* can be thought of as pure boolean functions. For example, consider the following predicate, which specifies that a list is acyclic:

```
predicate acyclic(Node root) =
  root == null ? true : acc(root.val) * acc(root.next) * acyclic(root.next)
```

Note, *acyclic* recursively generates accessibility predicates for each node in a list, and joins the predicates with the separating conjunction. Thus, *acyclic*(*l*) simply denotes that all heap locations in list *l* are distinct, i.e. that *l* is acyclic.

While the aforementioned advances allow users to specify and verify more code than ever before, formal verification is still largely unused due to the burden it places on its users. CompCert [Leroy 2009], an optimizing compiler, took 6 person-years to verify [Kästner et al. 2017]. Similarly, verifying seL4 [Klein et al. 2009] required 20 person-years of effort [Klein et al. 2009] compared to the only 2.2 person-years of effort required to write the kernel itself. Even for a simple program that verifies memory safety of list insertion, Wise et al. [2020] found a roughly 2:1 ratio of specification to program lines of code.

In response, Bader et al. [2018] introduced *gradual verification*, which allows users to write complete, partial, or empty specifications. *Partial, imprecise specifications* contain incomplete static information and are marked with  $\text{?}$ , e.g.  $\text{?} * x.f == 2$  where  $x.f == 2$  is the *static part*. *Empty specifications* are completely imprecise, e.g.  $\text{?}$  or  $\text{?} * \text{true}$ . Then, during static verification, imprecise specifications are strengthened (in non-contradictory ways) to support proof goals. Wherever these strengthenings occur, dynamic checks are inserted to preserve soundness. Bader et al. [2018]’s approach smoothly supports the spectrum between static and dynamic verification via the gradual guarantee, conservative extension, and pay-as-you-go properties. The *gradual guarantee*, formalized following the refined criteria in gradual typing [Siek et al. 2015], says that relaxing specifications should not introduce new (static or dynamic) verification errors. A gradual verifier is a *conservative extension* of a static verifier if the two verifiers coincide on fully-precise programs. Finally, a gradual verifier exhibits a *pay-as-you-go* cost model when users are rewarded with increased static correctness guarantees and decreased dynamic checking as specifications are refined.

In follow-up work, Wise et al. [2020] designed and formalized the first gradual verification system for recursive heap data structures. It supports the strengthening of imprecise specifications with accessibility predicates and abstract predicates; and thus, also (in theory) the run-time verification of these constructs. Wise et al. [2020] proved that their system is sound, is a conservative extension of their static verifier, and adheres to the gradual guarantee. However, their gradual verifier does not exhibit a pay-as-you-go cost model. In fact, their dynamic verifier always checks all memory safety and functional properties regardless of any static guarantees. Further, Wise et al. [2020]’s design and formalization, which is targeted to proofs, has not been implemented in a tool and is far from being implementable.

Therefore, this paper presents the design and implementation of *Gradual C0*—the first gradual verifier that can be used on real programs containing recursive heap data structures<sup>1</sup>. Gradual C0

<sup>1</sup>Gradual C0 is hosted on Github: <https://github.com/gradual-verification/gvc0>.

is built on top of the Viper [Müller et al. 2016] static verifier, which supports IDF and recursive abstract predicates. While Gradual C0’s design is inspired by Wise et al. [2020]’s foundational theory, Gradual C0 differs in significant ways and makes novel contributions. Wise et al. [2020]’s approach relies on *weakest liberal preconditions* to reason statically; instead, Gradual C0 (like Viper) relies on *symbolic execution*. Further, Gradual C0 supports a pay-as-you-go cost model, while Wise et al. [2020]’s approach does not. We addressed new technical challenges in gradual verification related to these differences:

- Gradual C0’s symbolic execution algorithm is responsible for statically verifying programs with imprecise specifications and producing minimally sufficient run-time checks for soundness. Achieving these goals with symbolic execution is nuanced. 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 are 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.

We also evaluate Gradual C0’s adherence to a pay-as-you-go cost model with a first of its kind empirical evaluation of a gradual verifier. Emulating Takikawa et al. [2016]’s performance lattice method, we explore the performance characteristics for partial specifications of four data structures. Though imprecision introduces unavoidable run-time checks, gradual verification decreases run-time overhead by an average of 50-75% compared to dynamic verification and thus Wise et al. [2020]’s approach. 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.

## 2 SYMBOLIC EXECUTION VS. WEAKEST LIBERAL PRECONDITIONS

In the exposition of their paper, Wise et al. [2020] describe how their gradual verifier operates on a simple list insertion program. In doing so, they present a naive approach to fulfilling the pay-as-you-go cost property. Their strategy utilizes a version of weakest liberal preconditions that handles imprecision, called  $\widetilde{\text{WLP}}$ . Therefore, the goal of this section is two-fold. First, we demonstrate the flaws in this naive approach—such as violating the gradual guarantee and producing too many run-time checks—by example. While some surface-level issues with the naive approach are fixable with a more complicated algorithm, the basic issue is fundamental: the backwards reasoning inherent to  $\widetilde{\text{WLP}}$  requires the verifier to choose between inserting dynamic checks too early (and breaking the gradual guarantee) and inserting too many dynamic checks (incurring unnecessary overhead). Second, we walk through Gradual C0’s verification process for the same example, while pointing out how Gradual C0 avoids these pitfalls by using symbolic execution.

Since Gradual C0 is designed to operate on C0 programs (see §3.1), throughout this section examples will be written in C0 [Arnold 2010]. C0 is a simple, safe subset of C designed for education. Concrete differences between C and C0 are described in §3.1.

Fig. 1 shows the C0 implementation of a linked list and a method for inserting a new node at the end of a non-empty list, called `insertLast`. Note, the `insertLast` method iterates over the list

for insertion using a while loop and thus diverges if given a cyclic list. Therefore, `insertLast` is gradually specified for both memory safety and the acyclicity of list insertion. The `insertLast` program in Fig. 1 is inspired by a similar example program in the exposition of Wise et al. [2020].

```

1 struct Node { int val; struct Node *next; };
2 typedef struct Node Node;
3
4 //@ predicate acyclic(Node* root) = ?;
5
6 Node* insertLast(Node* list, int val)
7     //@ requires ? && acyclic(list) && list != NULL;
8     //@ ensures acyclic(\result);
9 {
10     //@ unfold acyclic(list);
11     Node* y = list;
12     while (y->next != NULL)
13         //@ loop_invariant ? && y != NULL;
14         { y = y->next; }
15     y->next = alloc(struct Node);
16     y->next->val = val;
17     y->next->next = NULL;
18     //@ fold acyclic(list);
19     return list;
20 }
```

■ Imprecise specification  
■ Precise specification

Fig. 1. Partially specified non-empty linked list insertion for C0

The gradual specification of `insertLast` uses both precise and imprecise specifications highlighted in gray and yellow respectively. Acyclicity of a list is defined abstractly with the `acyclic` predicate on line 4. Here, the body of `acyclic` is defined by the imprecise formula `?`. The `acyclic` predicate is then used to specify acyclicity of list insertion in `insertLast`'s pre- and postconditions (lines 7-8). Further, `insertLast`'s precondition ensures `insertLast` only operates on non-empty lists with the `list != NULL` clause (line 7). Note that the concrete Gradual C0 syntax uses `&&` for the separating conjunction. Otherwise, `insertLast`'s precondition introduces optimism into the verification with imprecision as marked by `?`. Similarly, the loop invariant, `? && y != NULL` on line 13 is also imprecise. As a result, only `y != NULL`—that is, the current node of the list is not null—is enforced either statically or dynamically at every iteration of the loop. Accessibility predicates for memory safety are also implicitly enforced. In general, the imprecise formulas in Fig. 1 (lines 4, 7, and 13) are used to avoid specifying accessibility predicates.

Thus, the only interesting properties that can be verified by this gradual specification is whether `y != NULL` is preserved by the loop and whether heap accesses are justified. More details about the verification are given in §2.1 and §2.2.

As an aside, `insertLast`'s gradual specification also includes (un)folding the `acyclic` predicate (lines 10 and 18). Unfolding `acyclic` consumes the *predicate instance* `acyclic(list)` and introduces its body to the analysis. Folding `acyclic` does the reverse; `acyclic(list)`'s body is consumed in favor of its instance. The unfold on line 10 introduces more optimism into the analysis as `acyclic(list)`'s body is `?`. The fold on line 18 satisfies `insertLast`'s postcondition (line 8). Using folds and unfolds to control the availability of predicate information is considered an iso-recursive interpretation of predicates. Note, static verifiers rely on iso-recursive reasoning, while dynamic verifiers often reason about predicates equi-recursively [Summers and Drossopoulou 2013]—i.e. treat a predicate instance as equal to its complete unfolding.

## 2.1 Gradually Verifying List Insertion with $\widetilde{\text{WLP}}$

Fig. 2 shows how Wise et al. [2020]'s  $\widetilde{\text{WLP}}$  approach gradually verifies `insertLast` from Fig. 1.  $\widetilde{\text{WLP}}$  begins with `insertLast`'s postcondition `acyclic(\result)` on line 39 and is applied to each program statement step-by-step until the start of `insertLast` is reached on line 6. The specifications highlighted in purple are intermediate conditions produced by  $\widetilde{\text{WLP}}$  as it is applied in this backward fashion. Each intermediate condition is minimally sufficient to verify the following program statement and following intermediate condition. For example, `?` on line 35 satisfies the body of

```

1  Node* insertLast(Node* list, int val)
2  /*@ requires ? && acyclic(list)
3      && list != NULL; @*/
4  //@ ensures acyclic(\result);
5  {
6      ? && acyclic(list) && list != NULL
7       $\Rightarrow$  ? && acyclic(list)
8      ? && acyclic(list)
9      //@ unfold acyclic(list);
10     ?  $\Rightarrow$  ? && list != NULL &&
11         acc(list->next)
12     ? && list != NULL && acc(list->next)
13     Node* y = list;
14     ? && y != NULL && acc(y->next)
15     while (y->next != NULL)
16         //@ loop_invariant ? && y != NULL;
17     {
18         ? && y != NULL && acc(y->next) && y->next != NULL
19          $\Rightarrow$  ? && acc(y->next->next) &&
20             acc(y->next) && y->next != NULL
21         ? && acc(y->next->next) &&
22             acc(y->next) && y->next != NULL
23         y = y->next;
24         ? && y != NULL && acc(y->next)
25     }
26     ? && y != NULL && y->next == NULL  $\Rightarrow$ 
27     ? && acc(y->next)
28     ? && acc(y->next)
29     y->next = alloc(struct Node);
30     ? && acc(y->next) && acc(y->next->next) &&
31         acc(y->next->val)
32     y->next->val = val;
33     ? && acc(y->next) && acc(y->next->next)
34     y->next->next = NULL;
35     ?
36     //@ fold acyclic(list);
37     acyclic(list)
38     return list;
39     acyclic(\result)
40 }

```

<span style="background-color: #e0e0ff; border: 1px solid black; padding: 2px;"> </span> Intermediate condition produced by $\widetilde{\text{WLP}}$	<span style="background-color: #ffe0e0; border: 1px solid black; padding: 2px;"> </span> Dynamically checked right side of $\Rightarrow$
<span style="background-color: #e0e0ff; border: 1px solid black; padding: 2px;"> </span> Left side of $\Rightarrow$	<span style="background-color: #e0ffe0; border: 1px solid black; padding: 2px;"> </span> Statically checked right side of $\Rightarrow$

Fig. 2. The gradual verification of insertLast from Fig. 1 using  $\widetilde{\text{WLP}}$

acyclic(list) on line 36, which in turn satisfies acyclic(list) on line 37 (i.e. acyclic(\result), line 39). Similarly, the assignments to y->next->next and y->next->val on lines 34 and 32 require access to those locations in addition to y->next. Therefore, the intermediate condition on lines 30-31 contains accessibility predicates for those locations joined with ? (? satisfies the previous intermediate condition on line 35). Alloc on line 29 provides access to y->next->next and y->next->val, so only ? && acc(y->next) is required by line 28.

When  $\widetilde{\text{WLP}}$  cannot soundly propagate a condition backwards, a *consistent implication*  $\Rightarrow$  check is injected. According to Wise et al. [2020], these checks are required at the beginning of a method (lines 6-7), at the beginning of a loop body (lines 18-20), at the end of a loop with an imprecise invariant (lines 26-27), after unfolding an abstract predicate with an imprecise body (lines 10-11), and after a method call with an imprecise postcondition. So a consistent implication is created at lines 26-27. The loop invariant and negated loop guard are joined for the left-hand side (line 26), and the right-hand side is the current intermediate condition (line 27). The left-hand side cannot statically prove the right-hand side, but can do so optimistically with imprecision. Therefore, the right-hand side is dynamically checked as highlighted in red. Note that if the left-hand side of a consistent implication cannot possibly imply the right-hand side (e.g.  $? \&\& x == 3 \Rightarrow x != 3$ ), then the program is statically rejected.

To verify the loop body,  $\widetilde{\text{WLP}}$  starts with the loop invariant joined with accessibility predicates required for the loop guard (line 24). For variable assignment, like the one at line 23,  $\widetilde{\text{WLP}}$  substitutes the right-hand side of the assignment (y->next) for the left-hand side (y) in the current intermediate condition. This produces the intermediate condition before the assignment on lines 21-22. Additionally, an accessibility predicate is added for y->next. Upon reaching the start of the loop body, a consistent implication (lines 18-20) is inserted to check the current intermediate condition (lines 19-20). Here, the left-hand side (line 18) is the loop invariant, guard, and accessibility predicates

1	Node* insertLast(Node* list, int val)	29	? && y != NULL && acc(y->next) &&
2	/*@ requires ? && acyclic(list) &&	30	y->next == NULL
3	list != NULL; @*/	31	? && y != NULL && acc(y->next) &&
4	//@ ensures acyclic(\result);	32	y->next == NULL $\Rightarrow$ acc(y->next)
5	{	33	y->next = alloc(struct Node);
6	? && acyclic(list) && list != NULL	34	? && y != NULL && y->next != NULL &&
7	? && acyclic(list) && list != NULL	35	acc(y->next) && acc(y->next->next) &&
8	$\Rightarrow$ acyclic(list)	36	acc(y->next->val)
9	//@ unfold acyclic(list);	37	? && ... && acc(y->next->next) &&
10	? && list != NULL	38	acc(y->next->val) $\Rightarrow$ acc(y->next->val)
11	Node* y = list;	39	y->next->val = val;
12	? && list != NULL && y == list	40	? && ... && acc(y->next->next) &&
13	? && list != NULL && y == list	41	acc(y->next->val) && y->next->val == val
14	$\Rightarrow$ ? && acc(y->next) &&	42	? && ... && acc(y->next->next) $\Rightarrow$
15	y != NULL	43	acc(y->next->next)
16	while (y->next != NULL)	44	y->next->next = NULL;
17	//@ loop_invariant ? && y != NULL;	45	? && ... && acc(y->next->next) &&
18	{	46	y->next->next == NULL
19	? && y != NULL && acc(y->next) && y->next != NULL	47	? && ... && acc(y->next->next) &&
20	? && y != NULL && acc(y->next) &&	48	y->next->next == NULL $\Rightarrow$ ?
21	y->next != NULL $\Rightarrow$ acc(y->next)	49	//@ fold acyclic(list);
22	y = y->next;	50	? && acyclic(list)
23	? && y' != NULL && acc(y'->next) &&	51	return list;
24	y'->next != NULL && y == y'->next	52	? && acyclic(list) && list == \result
25	? && y' != NULL && acc(y'->next) &&	53	? && acyclic(list) && list == \result $\Rightarrow$
26	y'->next != NULL && y == y'->next $\Rightarrow$	54	acyclic(\result)
27	? && acc(y->next) && y != NULL	55	}
28	}		

■ Intermediate condition produced by Gradual C0	■ Dynamically checked right side of $\Rightarrow$
■ Left side of $\Rightarrow$	■ Statically checked right side of $\Rightarrow$

Fig. 3. The gradual verification of insertLast from Fig. 1 using Gradual C0

required by the guard. The loop guard and its accessibility predicate imply part of the right-hand side (the current intermediate condition), so that part is statically discharged as highlighted in green (line 20). The rest is dynamically checked, as before (line 19).

The condition prior to the loop (line 14) consists of the the loop invariant and  $\text{acc}(y \rightarrow \text{next})$  for the loop guard. As before,  $\widetilde{\text{WLP}}$  performs substitution on the condition on line 14 for the assignment on line 13 to produce the condition on line 12. This condition becomes the right-hand side of the consistent implication (lines 10-11) introduced for the unfold on line 9. The left-hand side is the body of  $\text{acyclic}(\text{list})$ , i.e. ?. The right-hand side is entirely dynamically checked, because ? provides zero static information.

The condition prior to the unfold (line 8) contains the predicate that is unfolded ( $\text{acyclic}(\text{list})$ ) and ?. The ? represents information not contained within  $\text{acyclic}(\text{list})$ . Finally, insertLast's precondition (the left-hand side on line 6) is used to check the condition prior to the unfold (the right-hand side on line 7). Since  $\text{acyclic}(\text{list})$  is implied by the precondition, the right-hand side is discharged statically.

## 2.2 Gradually Verifying List Insertion with Gradual C0



Fig. 3 demonstrates how Gradual C0 gradually verifies `insertLast` from Fig. 1. Note, the details of the symbolic execution are abstracted to make comparison with Wise et al. [2020]’s  $\overline{\text{WLP}}$  approach easier. For example, intermediate conditions (highlighted in purple) are represented as formulas rather than symbolic states. Also, consistent implication  $\Rightarrow$  is utilized instead of its algorithmic counterpart that is implemented in our tool. Concrete details can be found in §3.2.

Gradual C0 starts verifying `insertLast` on line 6 with `insertLast`’s precondition  $? \&\& \text{acyclic}(\text{list}) \&\& \text{list} \neq \text{NULL}$ . Gradual C0 analyzes each program statement until it reaches `insertLast`’s end on line 54. Intermediate conditions are created during this process, as highlighted in purple. Here, an intermediate condition contains maximal information propagated forward from the previous program statement and previous intermediate condition. If Gradual C0 cannot soundly propagate a condition forward, a consistent implication check is injected. This time, checks are required for preconditions at method calls, loop invariants at loop entry (lines 13-15) and loop body end (lines 25-27), predicates at unfolds (lines 7-8), predicate bodies at folds (lines 47-48), asserted formulas, and postconditions at the end of method bodies (lines 53-54). Additionally, checks are also produced for accessibility predicates required by program statements (lines 14, 20-21, 27, 31-32, 37-38, and 42-43). Therefore, a consistent implication is generated for the unfold at line 9. The left-hand side of the implication (line 7) is the current condition, *i.e.* `insertLast`’s precondition. The right-hand side is `acyclic(list)` (line 8) from the unfold, which is statically discharged (as highlighted in green).

After the check, the predicate `acyclic(list)` is consumed by the unfold leaving  $? \&\& \text{list} \neq \text{NULL}$  to be joined with `acyclic(list)`’s body  $?$ . Joining  $?$ s results in a single  $?$ , so the condition on line 10 is  $? \&\& \text{list} \neq \text{NULL}$ . Then,  $y = \text{list}$  is added to  $? \&\& \text{list} \neq \text{NULL}$  on line 12 after the variable assignment on line 11.

Upon reaching the loop at line 16, Gradual C0 checks whether the loop invariant  $(? \&\& y \neq \text{NULL}$ , line 17) holds and whether access to heap locations in the loop guard are justified with an accessibility predicate (`acc(y->next)`). This is the right-hand side of the consistent implication (lines 14-15). The left-hand side is the current intermediate condition  $? \&\& \text{list} \neq \text{NULL} \&\& y = \text{list}$ . The static part of the left-hand side ( $\text{list} \neq \text{NULL} \&\& y = \text{list}$ ) implies  $y \neq \text{NULL}$  in the right-hand side, so Gradual C0 statically proves this fact (line 15). The accessibility predicate `acc(y->next)` is dynamically checked thanks to  $?$  (line 14).

To verify the loop body, Gradual C0 begins with an intermediate condition containing the loop invariant, loop guard, and accessibility predicates required by the loop guard (line 19). The variable assignment on line 22 accesses `y->next` in its right-hand side, so a consistent implication checks this access on lines 20-21. Since the loop start condition on line 19 contains `acc(y->next)`, the check is statically verified. Then, the previous value for  $y$ , which is reassigned, is versioned as  $y'$  in the loop start condition. This way  $y = y' \rightarrow \text{next}$  from the assignment can be added safely to the loop start condition. The condition resulting from this process is on lines 23-24, and becomes the left-hand side of the consistent implication on lines 25-27 at the end of the loop. This implication checks preservation of the loop invariant and access to heap locations in the loop guard (the right-hand side). The left-hand side can only statically prove  $y \neq \text{NULL}$  (highlighted in green), and so access to `y->next` in the loop guard is checked dynamically (highlighted in red).

Following the loop, Gradual C0 begins with the loop invariant, negated loop guard, and accessibility predicates required by the loop guard (lines 29-30). Then, this after loop condition contains `acc(y->next)`, which is used to statically verify access to `y->next` (lines 31-32) in the `alloc` statement on line 33. The `alloc` statement adds accessibility predicates for a new node’s fields to the after loop condition (lines 34-36). This is enough to statically prove access to `y->next->val` and `y->next->next` in the following two assignment statements (lines 39 and 44). The assignments, then,

add `y->next->val == val` and `y->next->next == NULL` to the propagated intermediate conditions respectively. The resulting intermediate condition is on lines 45-46.

When Gradual C0 reaches the fold on line 49, the body of the predicate must be true, hence the consistent implication on lines 47-48. Since `acyclic(list)`'s body is `?`, the implication succeeds trivially. Then, the body of `acyclic(list)` is replaced in the current condition with `acyclic(list)` itself to produce the next condition `? && acyclic(list)` on line 50. Since the body is `?`, all of the current condition is conservatively replaced, and `?` in the next condition represents any residual information from the replacement. Finally, `insertLast`'s postcondition `acyclic(\result)` is discharged statically on lines 53-54 by `? && acyclic(list) && list == \result`, which was created from `? && acyclic(list)` and returning `list` on line 51.

### 2.3 $\widetilde{\text{WLP}}$ vs Gradual C0

Now that we have gradually verified `insertLast` with both a naive  $\widetilde{\text{WLP}}$  approach and Gradual C0, issues with the naive approach are more evident. First, the naive approach produces a number of additional run-time checks compared to Gradual C0: `list != NULL` on line 10 and `acc(y->next)` after the loop (line 27). This issue can be fixed by simple modifications to the naive approach. For example, `acc(list->next)` is dynamically checked at the same program point as `list != NULL`. Having access to `list->next` implies `list != NULL`, so checking `acc(list->next)` only is sound. Further, it is sound for the left-hand side of the implication (line 26), where the run-time check for `acc(y->next)` is created, to include accessibility predicates required by the loop guard. Thus, `acc(y->next)` is instead statically discharged. The run-time check for `acc(y->next->next)` in the loop body (line 19) ensures access to heap locations in the loop guard are justified on every iteration of the loop, including the last (when the execution breaks out of the loop).

By propagating information forward, Gradual C0 can statically prove that `list` or `y` after the variable assignment on line 11 is non-null. Thus, the assertion can be proved statically without the more complicated reasoning enhancements required in the naive approach. Also, Gradual C0 ensures access to heap locations in the loop guard are justified for every loop iteration at the end of the loop body on line 27. Thus, it is clear that `acc(y->next)` must statically hold directly after the loop. Gradual C0 was designed and implemented with this in mind, unlike with the naive approach.

A more fundamental issue with the naive approach is that it eagerly moves dynamic checks as early as possible, which breaks the gradual guarantee as formalized by Wise et al. [2020]. Half of the gradual guarantee states that if a verified program takes a step at run time, then a less precise version of that program is guaranteed to take the same step. However, if the unfold on line 9 in `insertLast` is removed, then the condition `? && list != NULL && acc(list->next)` on line 12 becomes the right-hand side of the consistent implication at the start of the method (line 7) and a dynamic check will be inserted. If, at run time, the executing function does not have permission to `list->next`, it will fail at the beginning of the method, instead of taking several steps and then failing just before testing the loop condition, which is where permission to `list->next` is actually needed. The gradual guarantee could, of course, be relaxed to permit eager checking, but this could have undesirable effects on users—for example, the program might fail before producing some useful output instead of after doing so.

It turns out that this problem is inherent:  $\widetilde{\text{WLP}}$  moves checks earlier because doing so is the only way that checks can be eliminated via static reasoning, so any implementation based on  $\widetilde{\text{WLP}}$  must choose between breaking the gradual guarantee and inserting run-time checks that would be unnecessary. In our subjective experience, it is also more difficult to come up with precise algorithms for gradual verification working backwards compared to forwards; this is a more pragmatic reason to prefer the symbolic-execution approach.



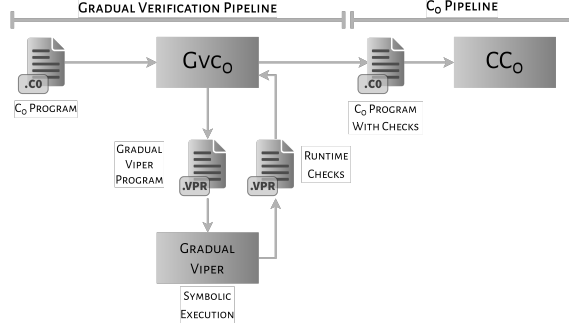


Fig. 4. System design of Gradual C0

Since Gradual C0 uses forward reasoning, it does not attempt to verify access to  $y \rightarrow \text{next}$  ( $\text{list} \rightarrow \text{next}$ ) until right before it is needed, *i.e.* right before the loop on line 14. For `insertLast` with the `unfold`,  $\text{acc}(y \rightarrow \text{next})$  is dynamically verified and  $y \neq \text{NULL}$  is statically verified on lines 14-15. Even if the `unfold` is removed, Gradual C0 still tries to verify both of these clauses at lines 14-15. Further, removing `unfold` does not change how `insertLast` executes at run time. Therefore, since  $\text{acc}(y \rightarrow \text{next})$  and  $y \neq \text{NULL}$  are true at lines 14-15 when the `unfold` is there, they are guaranteed to be true at the same program point when the `unfold` is missing. In general for Gradual C0, imprecision does not affect how a program executes and only weakens what is checked with consistent implications (in contrast, WLP may strengthen the right-hand side of an implication). Thus, any correctly verified program is guaranteed to take the same run-time steps even when made less precise. Gradual C0 easily adheres to the dynamic half of the gradual guarantee without weakening its formal statement.

In summary, by modeling the run-time execution of a program, Gradual C0 easily adheres to the gradual guarantee and avoids producing too many run-time checks compared to Wise et al. [2020]’s WLP approach. Additionally, modeling program execution allows Gradual C0 to produce more digestible error messages than WLP when dynamic checks fail. For example, an error message for  $\text{acc}(y \rightarrow \text{next})$  prior to the loop (line 14), which accesses  $y \rightarrow \text{next}$ , is easier to understand than one for  $\text{acc}(\text{list} \rightarrow \text{next})$  further up in the program.

### 3 GRADUAL C0

Gradual C0 is a working *gradual verifier* that is built on top of the Viper static verifier [Müller et al. 2016] and supports the C0 programming language [Arnold 2010]. The design of Gradual C0 is illustrated in Fig. 4.

Gradual C0 has two major sub-systems: 1) the gradual verification pipeline and 2) the C0 pipeline. The gradual verification pipeline is responsible for statically verifying C0 programs as in §2.2 and producing run-time checks for soundness. First, a C0 program is translated into a Gradual Viper program by Gradual C0’s frontend module, GVC0. Next, the Gradual Viper module uses a symbolic execution approach that handles imprecise formulas to statically verify the Gradual Viper program as in §2.2. Wherever imprecise formulas are strengthened in support of proofs, Gradual Viper creates run-time checks in its language to ensure soundness. Finally, GVC0 takes those run-time checks and produces a C0 program from them and the original C0 program. The C0 pipeline takes this C0 program and feeds it to the C0 compiler, CC0, which executes the program. Note, by designing Gradual Viper to operate on its own language (the Viper language modified for imprecise formulas), it can support multiple frontend languages, not just C0. We chose to build a C0

frontend first, because C0 is a simplified version of C—a well-known programming language—and C0 was designed with dynamic verification in mind.

For the rest of this section, we provide the implementation details behind Gradual Viper and GVC0’s design and illustrate the concepts via example. The rest of this section is organized as follows. §3.1 discusses how C0 programs are translated to Gradual Viper programs. Modifications made to the C0 and Viper languages to support verification are also discussed. Then, §3.2 details Gradual Viper’s symbolic execution approach and how it produces run-time checks. Finally, §3.3 focuses on how GVC0 turns run-time checks from Gradual Viper into C0 code for dynamic verification.

### 3.1 Translating C0 Source Code to Gradual Viper Source Code

The C0 language, with its minimal set of language features and its existing support for specifications, serves well as the source 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 that forbids casts, pointer arithmetic, and pointers to stack-allocated memory. 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 Fig. 5b, *i.e.* GVC0’s abstract syntax. GVC0 programs are made of struct and method declarations that largely follow C syntax. What differs from C is GVC0’s specification language. Methods may specify constraints on their input and output values as side-effect-free gradual formulas  $\tilde{\phi}$ , usually in `//@requires` or `//@ensures` clauses in the method header. Loops similarly contain loop invariants made of gradual formulas. Users of GVC0 may also specify abstract predicates with bodies that are gradual formulas. Such formulas  $\tilde{\phi}$  are imprecise formulas  $? \&\& \phi$  or complete boolean formulas  $\phi$  (Note, in this case,  $\phi$  must be self-framed as defined in IDF). A formula  $\phi$  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  $\phi$`  statements for convenience.

Now, GVC0 programs are converted to Gradual Viper programs before verification (Fig. 4). Thus, we also give the abstract syntax for Gradual Viper in Fig. 5c. The two languages are roughly 1-to-1, but there are some differences as highlighted in yellow (trivial) and blue (non-trivial) in Fig. 5. 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, `allocs`, and ternaries in arbitrary expressions, while Gradual Viper only allows method calls, `allocs`, and ternaries in corresponding program statements<sup>2</sup>. 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;  $e$  is translated into Gradual Viper syntax to verify its heap accesses, but  $e$  is not asserted (*i.e.*  $e$  is discarded).

Note that GVC0 does not support array and string values since gradually verifying any interesting properties about such constructs requires non-trivial 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.

<sup>2</sup>Note, ternaries correspond to if statements

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

## (a) Shared abstract syntax definitions

<p> <math>P ::= \overline{\text{struct}} \overline{\text{predicate}} \overline{\text{method}}</math>  <math>\text{struct} ::= \text{struct } S \{ \overline{T} \overline{f} \}</math>  <math>\text{predicate} ::= //@predicate \ p(\overline{T} \ x) = \tilde{\phi}</math>  <math>\text{method} ::= \tilde{T} \ m(\overline{T} \ x) \ \text{contract} \{ s \}</math>  <math>\text{contract} ::= //@requires \ \tilde{\phi}; //@ensures \ \tilde{\phi};</math>  <math>\text{invariant} ::= //@loop\_invariant \ \tilde{\phi}</math>  <math>T ::= \text{struct } S \mid \text{int} \mid \text{bool} \mid \text{char} \mid T^*</math>  <math>\tilde{T} ::= \text{void} \mid T</math>  <math>s ::= s; s \mid T \ x \mid T \ x = e \mid x = e</math>  <math>\quad \mid l = e \mid e \mid \text{assert}(e) \mid //@assert \ \phi</math>  <math>\quad \mid //@fold \ p(\tilde{e}) \mid //@unfold \ p(\tilde{e})</math>  <math>\quad \mid \text{if } (e) \{ s \} \text{ else } \{ s \}</math>  <math>\quad \mid \text{while } (e) \ \text{invariant} \{ s \}</math>  <math>\quad \mid \text{for } (s; e; s) \ \text{invariant} \{ s \}</math>  <math>e ::= v \mid x \mid op(\tilde{e}) \mid e \rightarrow f \mid *e \mid m(\tilde{e})</math>  <math>\quad \mid \text{alloc}(T) \mid e ? e : e</math>  <math>\tilde{e} ::= v \mid x \mid op(\tilde{e}) \mid \tilde{e} \rightarrow f \mid *e</math>  <math>l ::= x \rightarrow f \mid *x \mid l \rightarrow f \mid *l</math>  <math>x ::= \backslash \text{result} \mid id</math>  <math>v ::= n \mid c \mid \text{NULL} \mid \text{true} \mid \text{false}</math>  <math>\tilde{\phi} ::= ? \&amp;\&amp; \phi \mid \theta</math>  <math>\theta ::= \text{self-framed } \phi</math>  <math>\phi ::= \tilde{e} \mid p(\tilde{e}) \mid \text{acc}(l) \mid \phi \&amp;\&amp; \phi</math>  <math>\quad \mid \tilde{e} ? \phi : \phi</math> </p>	<p> <math>P ::= \overline{\text{field}} \overline{\text{predicate}} \overline{\text{method}}</math>  <math>\text{field} ::= \text{field } f : T</math>  <math>\text{predicate} ::= \text{predicate } p(\overline{x} : \overline{T}) \{ \tilde{\phi} \}</math>  <math>\text{method} ::= \text{method } m(\overline{x} : \overline{T}) \text{ returns } (\overline{y} : \overline{T})</math>  <math>\quad \text{contract} \{ s \}</math>  <math>\text{contract} ::= \text{requires } \tilde{\phi} \ \text{ensures } \tilde{\phi}</math>  <math>T ::= \text{Int} \mid \text{Bool} \mid \text{Ref}</math>  <math>s ::= s; s \mid \text{var } x : T \mid x := e</math>  <math>\quad \mid x.f := e \mid x := \text{new}(\tilde{f})</math>  <math>\quad \mid \overline{x} := m(\tilde{e}) \mid \text{assert } \phi</math>  <math>\quad \mid \text{fold } \text{acc}(p(\tilde{e})) \mid \text{unfold } \text{acc}(p(\tilde{e}))</math>  <math>\quad \mid \text{if } (e) \{ s \} \text{ else } \{ s \}</math>  <math>\quad \mid \text{while } (e) \ \text{invariant } \tilde{\phi} \{ s \}</math>  <math>e ::= v \mid x \mid op(\tilde{e}) \mid e.f</math>  <math>x ::= \text{result} \mid id</math>  <math>v ::= n \mid \text{null} \mid \text{true} \mid \text{false}</math>  <math>\tilde{\phi} ::= ? \&amp;\&amp; \phi \mid \theta</math>  <math>\theta ::= \text{self-framed } \phi</math>  <math>\phi ::= e \mid \text{acc}(p(\tilde{e})) \mid \text{acc}(e.f) \mid \phi \&amp;\&amp; \phi</math>  <math>\quad \mid e ? \phi : \phi</math> </p>
---	--

## (c) Gradual Viper abstract syntax

## (b) GVC0 abstract syntax

 Representation differs slightly in GVC0 vs. Gradual Viper	 Functionality in GVC0 that requires non-trivial translation to Gradual Viper
---	--

Fig. 5. Abstract syntax comparison for GVC0 and Gradual Viper

### 3.2 Gradual Viper: Symbolic Execution for Gradual Verification

In this section, we describe Gradual Viper’s symbolic execution based algorithm that supports the static verification of imprecise formulas. A Gradual Viper program is checked by examining each of its method and predicate definitions to ensure they are well-formed. The formal definitions are given in the supplementary material, §A.4. Intuitively, for each method, we start by defining symbolic values for the method arguments, and then create an initial symbolic state by calling the produce function on the method precondition. 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 set of run-time checks is built up, which (along with success or failure) is the ultimate result of gradual verification.

Our algorithm extends Viper’s [Müller et al. 2016] symbolic execution engine, and so Gradual Viper’s structure and design is heavily influenced by Müller et al. [2016]’s work. 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.

This section is organized as follows. Run-time checks and the collections that hold them are described in §3.2.1. We define symbolic states in §3.2.2 and preliminary definitions in §3.2.3. The 4 major functions of our algorithm are given in sections §3.2.4, §3.2.5, §3.2.6, and §3.2.7 for eval, produce, consume, and exec respectively.

**3.2.1 Run-time checks.** Run-time checks produced by Gradual Viper are collected into the  $\mathfrak{R}$  set. A run-time check is a 4-tuple  $(bcs_c, origin_c, location_c, \phi_c)$ , where  $bcs_c$  is a set of branch conditions,  $origin_c$  and  $location_c$  denote where the run-time check is required in the program, and  $\phi_c$  is what must be checked. In particular, a branch condition in  $bcs_c$  is also a tuple of  $(origin_e, location_e, e)$ , where  $origin_e$  and  $location_e$  define the program location at which Gradual Viper's execution branches on the condition  $e$ . A location is the AST element for a branch condition or a check in the program; in our formalism we write it as a formula  $\phi_l$ . 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 in the case where the condition is in the method being verified, or else 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:  $(\{(none, x > 2, \neg(x > 2))\}, z := m(y), acc(y.f), acc(y.f))$ . The check is for accessing  $y.f$ , and it is required for  $m$ 's precondition element  $acc(y.f)$  at the method call statement  $z := m(y)$ . The check is also only required when  $\neg(x > 2)$ , which must be evaluated at the program point where the AST element  $x > 2$  exists. Since  $\neg(x > 2)$ 's origin is none, it could have come 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  $(bcs_p, origin_p, rcs_p)$  where  $bcs_p$  is the set of branch conditions collected down the execution path  $p$ ,  $origin_p$  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  $\phi_l$  denoting the position of the check, and the check itself. It returns a new  $\mathcal{R}$  collection that is a copy of  $\mathcal{R}_{arg}$  with a run-time check added to  $\mathcal{R}_{arg}.rcs$ . If necessary, `addcheck` uses  $\mathcal{R}_{arg}.origin$  and substitution to ensure  $\phi_l$  and the check refer to the correct context. For example, let  $\phi_l$  and check  $\phi_c$  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 similarly takes in  $\mathcal{R}$ , a location for a branch condition, and the condition itself. It returns  $\mathcal{R}'$ , which is a copy of  $\mathcal{R}$  but with a branch condition added to  $\mathcal{R}.bcs$ . The `addbc` function also performs substitution similar to `addcheck`.

**3.2.2 Symbolic State.** We use  $\sigma \in \Sigma$  to denote a symbolic state, which represents an intermediate condition from §2. A symbolic state is a 6-tuple  $(isImprecise, h?, h, \gamma, \pi, \mathcal{R})$  consisting of a boolean `isImprecise`, a symbolic heap  $h?$ , another symbolic heap  $h$ , a symbolic store  $\gamma$ , a path condition  $\pi$ , and a collection  $\mathcal{R}$  (defined previously, §3.2.1). Since symbolic states represent intermediate conditions from §2, symbolic states may be imprecise. We therefore use the boolean `isImprecise` to record whether or not the state is imprecise.

A symbolic heap is a multiset of heap chunks that are currently accessible. A heap chunk may be for a field or predicate. 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  $\delta$ . We also refer to  $\delta$  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  $\delta$  is the snapshot of the

predicate. A predicate’s snapshot represents the values of the heap locations abstracted over by the predicate. The symbolic heap  $h_\gamma$  contains heap chunks that are currently accessible due to optimism in the symbolic execution. In contrast, the symbolic heap  $h$  contains heap chunks that are statically known to be accessible. Further,  $h$  maintains the invariant that its heap chunks are separated in memory, *i.e.*  $h$ ’s heap chunks can be joined successfully by the separating conjunction. The heap  $h_\gamma$  maintains no such invariant.

A symbolic store maps local variables to their symbolic values. A path condition (defined below) contains constraints on symbolic values that have been collected on the current verification path. As an example, the *empty symbolic state* is  $\sigma_0 = (\text{isImprecise} := \text{false}, h_\gamma := \emptyset, h := \emptyset, \gamma := \emptyset, \pi := \emptyset, \mathcal{R} := (\emptyset, \text{none}, \emptyset))$ .

**3.2.3 Preliminaries.** Like Viper [Müller et al. 2016], Gradual Viper’s symbolic execution algorithm consists of 4 major functions: *eval* (§3.2.4), *produce* (§3.2.5), *consume* (§3.2.6), and *exec* (§3.2.7). The functions evaluate expressions, produce (inhale) and consume (exhale) formulas, and execute program statements respectively. Following Viper’s lead, our 4 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 ( $\lambda \_ . \text{success}()$  or  $\lambda \_ . \text{failure}()$ ), indicating whether or not symbolic execution was successful. The rest of this section (§3.2) explains each of the 4 functions in more detail and how they come together to perform optimistic static verification. Highlighting is used to indicate deviations from Viper’s symbolic execution algorithm to handle imprecision in static verification: yellow highlighting deals with imprecision and blue highlighting deals with run-time check generation and collection.

But first, we introduce a few preliminary definitions here that will be helpful later. A path condition  $\pi$  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 the supplement Fig. 14. The *pc-all* function collects and returns all the constraints in  $\pi$ , *pc-add* adds a new constraint to  $\pi$ , and *pc-push* adds a new stack entry to  $\pi$ . 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 sub-parts. 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,  $\text{check}(\pi, t) = \text{pc-all}(\pi) \Rightarrow t$  queries the underlying SAT solver to see if the given constraint  $t$  is valid in a given path condition  $\pi$  (*i.e.*  $\pi$  proves or implies  $t$ ).

**3.2.4 Symbolic execution of expressions.** The symbolic execution of expressions by the *eval* function is defined in Fig. 6. 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  $op(\bar{e})$ , its arguments  $\bar{e}$  are each evaluated to their symbolic values  $\bar{t}$ . A symbolic value  $op'(\bar{t})$  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  $\text{add}(t_1, t_2)$  where  $e_1$  and  $e_2$  evaluate to  $t_1$  and  $t_2$  respectively.

```

eval( $\sigma$ ,  $t$ ,  $Q$ )      =  $Q(\sigma$ ,  $t$ )
eval( $\sigma$ ,  $x$ ,  $Q$ )      =  $Q(\sigma$ ,  $\sigma.y(x)$ )
eval( $\sigma_1$ ,  $op(\bar{e})$ ,  $Q$ ) = eval( $\sigma_1$ ,  $\bar{e}$ ,  $(\lambda \sigma_2, \bar{t} . Q(\sigma_2, op'(\bar{t})))$ )
eval( $\sigma_1$ ,  $e.f$ ,  $Q$ )    = eval( $\sigma_1$ ,  $e$ ,  $(\lambda \sigma_2, t .$ 
                        if  $(\exists f(r; \delta) \in \sigma_2.h . \text{check}(\sigma_2.\pi, r = t))$  then
                           $Q(\sigma_2, \delta)$ 
                        else if  $(\exists f(r; \delta) \in \sigma_2.h? . \text{check}(\sigma_2.\pi, r = t))$  then
                           $Q(\sigma_2, \delta)$ 
                        else if  $(\sigma_2.\text{isImprecise})$  then
                           $e_t := \text{translate}(\sigma_2, t)$ 
                           $\mathcal{R}' := \text{addcheck}(\sigma_2.\mathcal{R}, e.f, \text{acc}(e_t.f))$ 
                           $\delta := \text{fresh}$ 
                           $Q(\sigma_2\{h? := \sigma_2.h? \cup f(t; \delta), \mathcal{R} := \mathcal{R}'\}, \delta)$ 
                        else failure())

```

■ Handles imprecision

■ Handles run-time check generation and collection

Fig. 6. Rules for symbolically executing expressions

Finally, the most interesting rule is for fields  $e.f$ . The receiver  $e$  is first evaluated to  $t$  resulting in a new state  $\sigma_2$ . Then, eval looks for a heap chunk for  $t.f$  first in the current heap  $h$ .<sup>3</sup> If a chunk exists, then the heap read succeeds and  $\sigma_2$  and the chunk's snapshot  $\delta$  is returned to the continuation. If a chunk does not exist in  $h$ , then eval looks for a chunk in the optimistic heap  $h?$ . Similarly, the heap read succeeds if a chunk is found, and its snapshot is returned with  $\sigma_2$ . If a heap chunk for  $t.f$  is not found in either heap, then the heap read can still succeed when  $\sigma_2$  is imprecise. In this case,  $\sigma_2$ 's imprecision optimistically provides access to  $t.f$ . Therefore, a run-time check for  $\text{acc}(e_t.f)$  is created and added to  $\sigma_2$ 's set of run-time checks (as 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, `translate` (described in the supplementary material, Fig. 17) is called on  $t$  with the current state  $\sigma_2$  to compute  $e_t$ . Additionally, the AST element  $e.f$  is used to denote the check's location.

Afterwards, a fresh snapshot  $\delta$  is created for  $t.f$ 's value. Further, a heap chunk  $f(t; \delta)$  for  $t.f$  and  $\delta$  is created and added to  $\sigma_2$ 's optimistic heap before it is passed to the continuation along with  $\delta$ . 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 Gradual Viper produces. Finally, verification of the heap read for  $t.f$  fails when none of the aforementioned cases are true. Fig. 15 in the supplementary material defines a variant of eval, called eval-pc, that is similar but doesn't extend the optimistic heap; eval-pc is used in produce and consume.

**3.2.5 Symbolic production of formulas.** Produce (Fig. 7) 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.

<sup>3</sup>Note that heap lookup in eval also looks for heap chunks that are aliases (according to the path condition) to the chunk in question.



$\text{produce}(\sigma, \text{?}\&\&\phi, \delta, Q)$	$= \text{produce}(\sigma\{\text{isImprecise} := \text{true}\}, \phi, \text{second}(\delta), Q)$
$\text{produce}(\sigma_1, e, \delta, Q)$	$= \text{eval-pc}(\sigma_1, e, (\lambda \sigma_2, t. Q(\sigma_2\{\pi := \text{pc-add}(\sigma_2.\pi, \{t, \delta = \text{unit}\}\}, \mathcal{R} := \sigma_1.\mathcal{R}\})))$
$\text{produce}(\sigma_1, \text{acc}(p(\bar{e})), \delta, Q)$	$= \text{eval-pc}(\sigma_1, \bar{e}, (\lambda \sigma_2, \bar{t}. Q(\sigma_2\{h := \sigma_2.h \uplus p(\bar{t}; \delta), \mathcal{R} := \sigma_1.\mathcal{R}\})))$
$\text{produce}(\sigma_1, \text{acc}(e.f), \delta, Q)$	$= \text{eval-pc}(\sigma_1, e, (\lambda \sigma_2, t. \\ (h', \pi') := (\sigma_2.h \uplus f(t; \delta), \text{pc-add}(\sigma_2.\pi, \{t \neq \text{null}\})), \\ Q(\sigma_2\{h := h', \pi := \pi', \mathcal{R} := \sigma_1.\mathcal{R}\})))$
$\text{produce}(\sigma_1, \phi_1 \&\&\phi_2, \delta, Q)$	$= \text{produce}(\sigma_1, \phi_1, \text{first}(\delta), (\lambda \sigma_2. \text{produce}(\sigma_2, \phi_2, \text{second}(\delta), Q)))$
$\text{produce}(\sigma_1, e ? \phi_1 : \phi_2, \delta, Q) = \text{eval-pc}(\sigma_1, e, (\lambda \sigma_2, t. \\ \text{branch}(\sigma_2\{\mathcal{R} := \sigma_1.\mathcal{R}\}, e, t, \\ (\lambda \sigma_3. \text{produce}(\sigma_3, \phi_1, \delta, Q)), \\ (\lambda \sigma_3. \text{produce}(\sigma_3, \phi_2, \delta, Q))))$	

■ Handles imprecision
■ Handles run-time check generation and collection

Fig. 7. Rules for symbolically producing formulas

The field  $e.f$  in  $\text{acc}(e.f)$  first has its receiver  $e$  evaluated to a symbolic value  $t$ . Then, using the parameter  $\delta$  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,  $\text{acc}(e.f)$  implies  $e \neq \text{null}$  and so that fact is recorded in the path condition. When the separating conjunction  $\phi_1 \&\&\phi_2$  is produced,  $\phi_1$  is first produced into the symbolic state, followed by  $\phi_2$ . 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  $\phi_1$  is produced into the state under the assumption that  $t$  is true, and along the other path  $\phi_2$  is produced under the  $\neg t$  assumption. Both paths follow the continuation to the end of its execution, and a branch condition corresponding to the  $t$  assumption made is added to the symbolic state. Additionally, paths may be pruned when the path is infeasible (the assumption about  $t$  would contradict the current path conditions). The definition of branch and other details are in the supplementary material §A.1. Note, as highlighted in blue, produce does not add any run-time checks to the symbolic state.

**3.2.6 Symbolic consumption of formulas.** The goals of consume are 3-fold: 1) given a symbolic state  $\sigma$  and formula  $\tilde{\phi}$ , check whether  $\tilde{\phi}$  is established by  $\sigma$ , i.e.  $\tilde{\phi}_\sigma \widetilde{\Rightarrow} \tilde{\phi}$  where  $\tilde{\phi}_\sigma$  is the formula which represents the state  $\sigma$ , 2) produce and collect run-time checks that are minimally sufficient for  $\sigma$  to establish  $\tilde{\phi}$  soundly, i.e. the red and green highlighting in Fig. 3, and 3) remove accessibility predicates and predicates that are asserted in  $\tilde{\phi}$  from  $\sigma$ . The rules for consume are described in great detail in supplement §A.2. We give an abstract description here. For the first and second goals, heap chunks representing accessibility predicates and predicates in  $\tilde{\phi}$  are looked up in the heap  $h$  and optimistic heap  $h_?$  from  $\sigma$ . When  $\sigma$  is precise, the heap chunks must be in  $h$  or verification fails. If  $\sigma$  is imprecise, then the heap chunks are always justified either by the heaps or imprecision. Run-time checks for accessibility predicates or predicates that are verified by imprecision are collected in  $\sigma.\mathcal{R}$ . Clauses in  $\tilde{\phi}$  containing logical expressions are first evaluated to a symbolic value  $t$ , and then  $t$  is checked against  $\sigma$ 's path condition  $\pi$ . If  $\sigma$  is precise, then  $\text{pc-all}(\pi) \Rightarrow t$  must hold (i.e. the constraints in  $\pi$  prove  $t$ ) or verification fails. In contrast, when  $\sigma$  is imprecise,  $\bigwedge \text{pc-all}(\pi) \wedge t$  must hold (i.e.  $t$  does not contradict constraints in  $\pi$ ) 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  $\pi$ . Finally, fields used in  $\tilde{\phi}$  must have corresponding heap chunks in  $h$  when  $\sigma$  and  $\tilde{\phi}$  are precise;

$$\begin{aligned}
\text{exec}(\sigma_1, x.f := e, Q) &= \text{eval}(\sigma_1, e, (\lambda \sigma_2, t. \text{consume}(\sigma_2, \text{acc}(x.f), (\lambda \sigma_3, \_ . \\
&\quad \text{produce}(\sigma_3, \text{acc}(x.f) \ \&\& \ x.f = t, \text{pair}(\text{fresh}, \text{unit}), Q)))))) \\
\text{exec}(\sigma_1, \bar{z} := m(\bar{e}), Q) &= \text{eval}(\sigma_1, \bar{e}, (\lambda \sigma_2, \bar{t}. \\
&\quad \mathcal{R}' := \sigma_2.\mathcal{R}\{\text{origin} := (\sigma_2, \bar{z} := m(\bar{e}), \bar{t})\} \\
&\quad \text{consume}(\sigma_2\{\mathcal{R} := \mathcal{R}'\}, \text{meth}_{\text{pre}}[\overline{\text{meth}_{\text{args}} \mapsto t}], (\lambda \sigma_3, \delta. \\
&\quad \sigma_4 := \sigma_3\{\gamma := \text{havoc}(\sigma_3.\gamma, \bar{z})\} \\
&\quad \text{produce}(\sigma_4, \text{meth}_{\text{post}}[\overline{\text{meth}_{\text{args}} \mapsto t}][\overline{\text{meth}_{\text{ret}} \mapsto z}], \text{fresh}, \\
&\quad (\lambda \sigma_5. Q(\sigma_5\{\mathcal{R} := \sigma_5.\mathcal{R}\{\text{origin} := \text{none}\}\}))))))
\end{aligned}$$

Handles imprecision
   Handles run-time check generation and collection

Fig. 8. Select rules for symbolically executing program statements

otherwise when  $\sigma$  is imprecise or  $\tilde{\phi}$  is imprecise, field access is always justified either by the heaps or by imprecision. A run-time check containing an accessibility predicate for the field is added to  $\sigma.\mathcal{R}$  when imprecision is relied on.

The third goal of `consume` is to remove heap chunks  $\overline{hc_i}$  representing accessibility predicates and predicates in  $\tilde{\phi}$  from  $\sigma$ , and in particular, from heaps  $h$  and  $h_\gamma$ . When  $\sigma$  and  $\tilde{\phi}$  are both precise, the heap chunks in  $\overline{hc_i}$  are each removed from  $h$  ( $h_\gamma$  is empty here). If  $\tilde{\phi}$  is imprecise, then all heap chunks in both heaps are removed as they may be in  $\overline{hc_i}$  or  $\tilde{\phi}$  may represent them with imprecision. Finally, when  $\sigma$  is imprecise and  $\tilde{\phi}$  is precise, any heap chunks in  $h$  or  $h_\gamma$  that overlap with or may potentially overlap with (thanks to  $\sigma$ 's imprecision) heap chunks in  $\overline{hc_i}$  are removed.

**3.2.7 Symbolic execution of statements.** Select rules for `exec`, which symbolically executes program statements, are given in Fig. 8. Here we provide an intuition; the full set of rules are listed in the supplementary material §A.3. The `exec` function takes a symbolic state  $\sigma$ , program statement  $stmt$ , and continuation  $Q$ . Then, `exec` symbolically executes  $stmt$  using  $\sigma$  to produce a potentially modified state  $\sigma'$ , which is passed to the continuation. That is, `exec` is the main verification function: it produces the intermediate conditions in Fig. 3 (§2.2) and calls `eval` and `consume` to produce consistent implications  $\Rightarrow$  (Fig. 3 §2.2) where necessary.

Symbolic execution of field assignments first evaluates the right-hand side expression  $e$  to the symbolic value  $t$  with the current state  $\sigma_1$  and `eval`. Any field reads in  $e$  are either directly or optimistically verified using  $\sigma_1$ . Then, the resulting state  $\sigma_2$  must establish write access to  $x.f$  in `consume`, i.e.  $\sigma_2 \Rightarrow \text{acc}(x.f)$ . The call to `consume` also removes the field chunk for  $\text{acc}(x.f)$  from  $\sigma_2$  (if it is in there) resulting in  $\sigma_3$ . Therefore, the call to `produce` can safely add a fresh field chunk for  $\text{acc}(x.f)$  alongside  $x.f = t$  to  $\sigma_3$  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 method call rule evaluates the arguments  $\bar{e}$  to symbolic values  $\bar{t}$ , consumes the method precondition (substituting arguments with  $\bar{t}$ ) and 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`. A more in-depth explanation is in the supplementary material, along with the other `exec` rules.

```

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. 9. Original insertLastWrapper program with run-time checks from Gradual Viper

```

1  Node* insertLastWrapper(Node* l, int val,
2      OwnedFields* _ownedFields)
3  {
4      bool _cond_1 = l == NULL;
5      if (l == NULL) {
6          l = alloc(struct Node);
7          l->id = addStructAcc(_ownedFields, 2);
8          l->val = val; l->next = NULL;
9      } else {
10         if (!_cond_1) { acyclic(l, _ownedFields); }
11         OwnedFields* _tempFields =
12             initOwnedFields(_ownedFields->instCntr);
13         sep_acyclic(l, _tempFields);
14         l = insertLast(l, val, _ownedFields);
15     }
16     if (_cond_1) { acyclic(l, _ownedFields); }
17     OwnedFields* _tempFields1 =
18         initOwnedFields(_ownedFields->instCntr);
19     sep_acyclic(l, _tempFields1);
20     return l;
21 }

```

■ Run-time checks from GVC0

Fig. 10. GVC0 generated insertLastWrapper program with run-time checks

### 3.3 Dynamic Verification: Translating 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. GVC0 is responsible for creating an executable C0 program from the run-time checks in  $\mathfrak{R}$  and the original C0 program. The resulting C0 program is then passed to the C0 compiler to be executed at run time. For example, consider the C0 program in Fig. 9 that implements a method for inserting a new node at the end of a list, called insertLastWrapper. Note that when insertLastWrapper is passed a non-empty list it calls insertLast from Fig. 1 to perform insertion (line 17). Further, insertLastWrapper is 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. Note, the acyclic predicate is fully specified (is precise) (lines 1-4). Fig. 9 also contains run-time checks generated by Gradual Viper for insertLastWrapper, as highlighted in blue. The first check on lines 15-16 ensures the list l sent to insertLast (line 17) is acyclic. This check is only required when l is non-empty (non-null). The second check on lines 20-21 ensures the list returned from insertLastWrapper is acyclic. This check is only required when insertLastWrapper’s parameter l is empty (null). Clearly, these checks are not in a form that is executable by the C0 compiler; therefore, GVC0 takes the program and checks in Fig. 9 and returns the program in Fig. 10 that is executable. That is, GVC0 translates 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, lines 1-4) from Gradual Viper into C0 source code. We discuss the aforementioned translations via example throughout the rest of this section: §3.3.1, §3.3.2, and §3.3.3 respectively. While not needed in the insertLastWrapper example, GVC0 also translates checks of simple logical expressions into C0 assertions: e.g. assert(y >= 0);.

**3.3.1 Translating branch conditions.** Run-time checks contain branch conditions that denote the execution path a check is required on. For example, in Fig. 9 `acyclic(\result)` should only be checked at lines 20-21 when `l == NULL`, as indicated with the branch condition `(none, l == NULL, l == NULL)`. Therefore, GVC0 first translates the condition `l == NULL` into C0 code. The origin and location pair `(none, l == NULL)` tells GVC0 that `l == NULL` must be evaluated at the program point in `insertLastWrapper` that contains the `l == NULL` AST element. As a result, in Fig. 10 a boolean variable `_cond_1` is introduced on line 4 to hold the value of `l == NULL`. The condition variable `_cond_1` is then used in the C0 run-time check for `acyclic(\result)` later in the program (line 16). Further, to reduce run-time overhead, `_cond_1` is also used in the check for `acyclic(l)` on line 10, which relies on the same branch point `(none, l == NULL)`. While not demonstrated here, GVC0 supports short-circuit evaluation of conditions on the same execution path.

**3.3.2 Translating predicates.** Now that GVC0 has translated the branch conditions in Fig. 9 into condition 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_1) {acyclic(l, _ownedFields);}` on line 10 in Fig. 10. 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 14. The branch condition becomes the if statement with condition `!_cond_1` (§3.3.1), and `acyclic(l)` is turned into the method call `acyclic(l, _ownedFields)`. The `acyclic` method implements `acyclic`'s predicate body as C0 code: it asserts true for empty lists and recursively verifies accessibility predicates (using `_ownedFields`) for nodes in non-empty lists. For efficiency, separation of list nodes is encoded separately on lines 11-13. We discuss the dynamic verification of accessibility predicates and the separating conjunction in C0 code next (§3.3.3). Finally, note that a similar C0 check is created for `acyclic(\result)` on lines 16-19.

**3.3.3 Translating 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. An owned field is a tuple  $(id, field)$  where  $id$  is an integer identifier for a struct instance and  $field$  is the integer index for a field in the struct. Then, the `OwnedFields` struct instance called `_ownedFields` contains currently owned fields. Also, all struct definitions in a program are modified to contain an `_id` field. When a new struct instance is created—such as allocating a new node on line 6 in Fig. 10—the `_id` field is initialized with the value of a global integer `instCntr` that uniquely identifies the instance. The call to library method `addStructAcc` on line 7 performs this functionality and then increments `instCntr`. It also adds all fields in the struct instance (e.g. `l->val: (l->_id, 0)`, `l->next: (l->_id, 1)`, and `l->_id: (l->_id, 2)`) to `_ownedFields`.

For methods with an imprecise<sup>4</sup> pre- or postcondition, like `insertLastWrapper`, a parameter that initializes `_ownedFields` is added to the method's declaration (line 2, Fig. 10). When a method's precondition is imprecise, then any caller will pass all of its owned fields to the method, as on line 14 for the call to `insertLast`. After execution, the callee method returns all of its owned fields to the caller. When a method's precondition is precise, then any caller only passes its owned fields specified by the precondition to the method. 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 are returned. Finally, in precisely specified methods (no external—pre- and postconditions—or internal—loop invariants, unfolds, folds, etc.—specifications contain or introduce imprecision), GVC0 does not implement any `_ownedFields` tracking.

<sup>4</sup>Note, here, a formula is also considered imprecise if it contains predicates that when completely unrolled expose ?—i.e. we use an equi-recursive interpretation of predicates to define imprecision here.

Once `_ownedFields` tracking is implemented, GVC0 uses `_ownedFields` 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 `_ownedFields`. For example, a check for `acc(1->val)` looks like `assertAcc(_ownedFields, 1->_id, 0)`; in C0 code, where 0 is the index for `val` in the `Node` struct. Wherever GVC0 must check separation of heap locations,—such as for the nodes in list `l` at lines 10-13—GVC0 creates an auxiliary data structure `_tempFields` of type `OwnedFields` for checking separation. We check that heap cells are disjoint by adding them one at a time to `_tempFields`; if a heap cell has already been added, the separation check fails. The library method `initOwnedFields` creates the auxiliary data structure and GVC0 generates a `sep_X` method for each predicate `X` to actually perform the separation check. When we are done, `_tempFields` is discarded, as its purpose was only to check separation. Similar checks are performed for the `acyclic(\result)` check on lines 16-19.

#### 4 EMPIRICAL EVALUATION: IS GRADUAL VIPER PAY-AS-YOU-GO?

Early theoretical work on gradual typing [Siek and Taha 2006] hinted at the possibility of obtaining a pay-as-you-go cost model. However, practical implementations demonstrate that this is not easy to achieve due to many issues including, but not limited to: higher-order wrappers and their allocation, and space efficiency [Herman et al. 2010; Takikawa et al. 2016]. Theoretical work on gradual verification [Bader et al. 2018] aims at a similar pay-as-you-go cost model, but Wise et al. [2020] identify challenges with achieving it. In particular, Wise et al. [2020] show that when a loop invariant is specified before the loop body, gradual verification will introduce run-time checks, and thus, additional run-time overhead to ensure the loop invariant is preserved by the loop. That is, specifying a loop invariant has increased rather than decreased run-time overhead. Fortunately, the user only pays for what is specified and nothing more—a different form of pay-as-you-go, which we call *pay-only-for-specs*. We refer to the ideal pay-as-you-go cost model—that says adding more specifications decreases run time cost correspondingly—as *benefit-as-you-go*. In Wise et al. [2020]’s example, specifying more of the loop body eventually results in the benefit-as-you-go trend. Through our first-ever implementation of a gradual verifier, we can explore Wise et al. [2020]’s speculations about pay-as-you-go trends in gradual verification.

We explore the performance characteristics of intermediate, imprecise specifications by adapting Takikawa et al. [2016]’s work to gradual verification. In particular, we observe how adding or removing individual atomic formulas and `?` within a specification impacts the degree of static and dynamic verification and, as a result, the run-time overhead of the program. Additionally, we compare the run-time performance of gradual verification, and in particular Gradual C0, to a fully dynamic approach. The aforementioned ideas are captured in the following research questions:

- RQ1:** Does gradual verification result in less run-time overhead than a fully dynamic approach?
- RQ2:** As more elements are added to partial specifications, can more verification conditions be eliminated statically?
- RQ3:** Are there particular types of specification elements that lead to significant changes in run-time overhead?

##### 4.1 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 complete specification where, at each step, an *element* is added to the current, partial specification. An element is an atomic conjunct (excluding boolean primitives) in

Example	Unverified Complexity	# Specs	Contents of Complete Spec					
			Fold	Unfold	Pre.	Post.	Pred. Body	Loop Inv.
Binary Search Tree	$O(n \log(n))$	3344	43	23	0/20/20/23	0/22/3/23	6/6/7/4	0/0/4/2
Linked List	$O(n)$	1728	17	10	8/6/15/5	4/5/6/5	4/3/4/3	4/2/5/2
Composite	$O(n \log(n))$	2577	28	15	0/10/2/12	0/11/1/12	32/9/17/3	0/3/2/3
AVL	$O(n \log(n))$	3056	25	14	3/4/5/9	3/6/9/9	25/8/21/3	1/1/2/1

Table 1. For each example, the complexity of the test program without verification, the number of partial specifications sampled, and the distribution of specification elements for the complete specification. Element counts are formatted as “Accessibility Predicate/Predicate Instance/Boolean Expression/Imprecision”

any type of method contract, assertion, or loop invariant. We form a lattice of partial specifications by varying elements in the complete specification. We also similarly vary the presence of ? in formulas that are complete—contain the same elements as their counterparts in the complete 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 complete 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 ones supported by the gradual guarantee.

To illustrate the aforementioned approach, consider the following loop invariant:

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

There are only two elements present: the sortedSeg predicate instance and the boolean expression `curr->val <= val`; `true` is removed. Then, the lattice generated for a program containing this invariant has five unique specifications where four of them contain a combination of the two elements joined with ? and the fifth one is the complete invariant minus `true` and ?.

## 4.2 Data Structures

To apply this methodology, we implemented and fully specified four recursive heap data structures with Gradual C0: binary search tree, sorted linked list, composite tree, and AVL tree. 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  $\omega$ . 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.

*Binary Search Tree (BST).* The implementation of our binary search tree is typical; each node contains a value and pointers to left and right nodes. We statically specify memory safety and preservation of the binary search tree 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  $\omega$  and sequentially adds and removes a set of  $\omega$  values in the range  $[0, 2\omega]$ . Note that values are removed in the same order they were added.

*Linked List.* We implemented a list with insertion similar to the what is given in Fig. 1 and described in §2. Insertion is statically specified for memory safety and preservation of list sortedness. Its test program creates a new list and inserts  $\omega$  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 memory safety. Its test program starts with



Example	$\omega$	% $\Delta t$ , GV vs. DV				% Steps GV<DV for Paths DV<GV				% Paths GV <DV
		Mean	St. Dev.	Max	Min	Mean	St. Dev.	Max	Min.	
Binary Search Tree	4	-34.4	20.9	19.2	-78.1	97.8	1.7	100	93.3	6.25
	8	-59.7	25.0	14.5	-96.6	98.9	1.4	100	95.2	18.75
	16	-75.2	22.0	14.1	-99.4	99.0	1.0	100	96.7	25.0
Linked List	8	-10.7	14.3	29.5	-57.3	72.9	17.5	99.1	42.6	0.0
	16	-28.4	28.1	32.1	-90.7	72.2	20.4	99.1	38.0	0.0
	32	-46.6	41.2	24.9	-99.0	75.4	18.6	100	39.8	6.25
AVL	4	-2.5	11.8	87.4	-42.6	68.3	15.3	98.4	39.8	0.0
	8	-21.6	17.1	69.9	-79.5	93.5	5.5	100	83.8	6.25
	16	-49.7	22.6	27.0	-97.0	97.7	3.0	100	91.6	31.25

Table 2. Summary statistics for the performance of each example over 16 paths at selected workloads ( $\omega$ ), comparing gradual verification (GV) against dynamic verification (DV). The grouped column “% in  $\Delta t$ , 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.

a root node and builds a tree of size  $\omega$  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 our AVL tree with insertion is standard except the height of the left and right subtrees is stored in each node rather than 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 or ghost variables, which Gradual C0 does not support. In addition to specifying the AVL balanced property for insertion, we also specify memory safety. The AVL test program starts with a root node and builds a tree of size  $\omega$  by inserting randomly valued nodes into the tree using balanced insertion.

Notably, it was our experience that the incrementalness 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’s conventional wisdom that one of the benefits of static checking is that you get feedback early, when it is easier to correct mistakes. Here, gradual checking has a similar benefit over static checking: we found an error earlier than we would have otherwise, presumably saving time.

### 4.3 Experimental Setup

Our approach diverges from Takikawa et al. [2016]’s work by sampling a subset of partial specifications in a lattice rather than all of them. With upwards of 100 elements in the specifications for each data structure, it is combinatorially infeasible to fully explore every partial specification. Instead, we sample 16 unique paths through the lattice from randomized orderings of specification elements. Every step is executed with three workloads chosen arbitrarily to ensure observable

differences in timing. Each timing measurement is the median of 10 iterations, and programs were executed on four physical Intel Core i5-4250U 1.3GHz Cores with 16 GB of RAM.

We introduce two baseline configurations to compare Gradual C0 against. The *dynamic* configuration transforms every specification into a run-time check and inserts accessibility predicate checks for field dereferences—this emulates a dynamic verifier. The *framing* configuration only performs the accessibility predicate checks; and therefore, represents the minimal dynamic checks that must be performed in a language that checks ownership.<sup>5</sup> To illustrate our sampling and baseline methods, consider again the invariant in 4.1. A maximum of three run-time checks may be inserted; an accessibility check for `curr->val`, an equi-recursive unrolling of `sortedSeg`, and an assertion that `curr->val <= val`. All of these checks are inserted for the *dynamic* configuration, but in the *framing* configuration, only the accessibility check for `curr->val` is inserted.

#### 4.4 Evaluation

**RQ1: Run-time Gains Over Fully Dynamic Approach.** 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, on average Gradual C0 reduces run-time overhead by 2-75% (Table 2, Column 3) compared to dynamic verification. In fact, the speed-ups are more substantial as  $\omega$  increases: -75%, -47%, and -50% at the largest  $\omega$  values compared to -34%, -11%, and -2% at the lowest. 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 14-87%. Fortunately, for lattice paths that produced these poor-performing specifications, gradual verification still outperforms dynamic verification (on average) for 72-99% (Table 2, Column 7) of all steps. Further, these outliers are partially due to the bookkeeping we insert to track conditionals, which is unoptimized and can be improved.

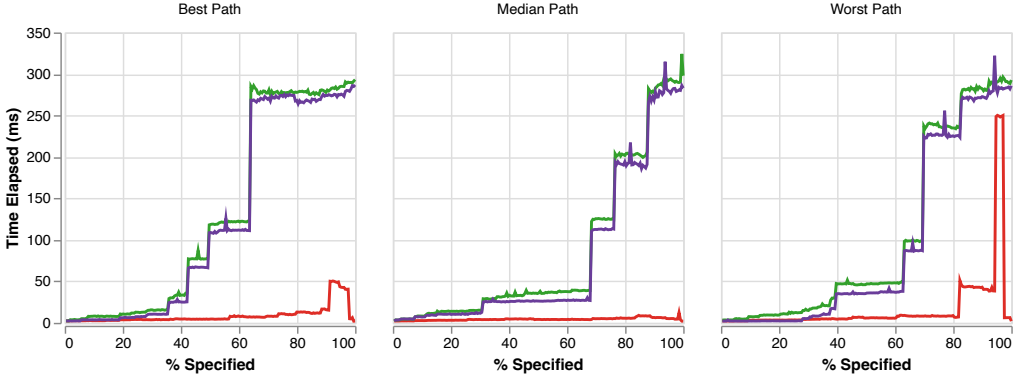
Fig. 11 displays the run-time cost (area under the red curve) of the best (minimum time elapsed), median, and worst (maximum time elapsed) performing lattice paths per benchmark for Gradual C0. In particular, the time elapsed for each step in a path is averaged into one statistic representing the overall performance of the path. Then, that number is used to determine which paths should be plotted and are the best, median, or worst paths for Gradual C0. For comparison, the run-time performances of the fully dynamic (in green) and framing only (in purple) approaches for a chosen path are also plotted alongside the results for Gradual C0. In all the plots Gradual C0 significantly outperforms dynamic verification—the red lines are always well below the green ones. However, the best path plot for AVL breaks this trend briefly at around 49% static completion—the red line briefly spikes above the green line—and contains an example of the outliers discussed previously.

Therefore, the answer to **RQ1** is *yes*. Gradual C0 and thus gradual verification outperforms dynamic verification for each of our examples.

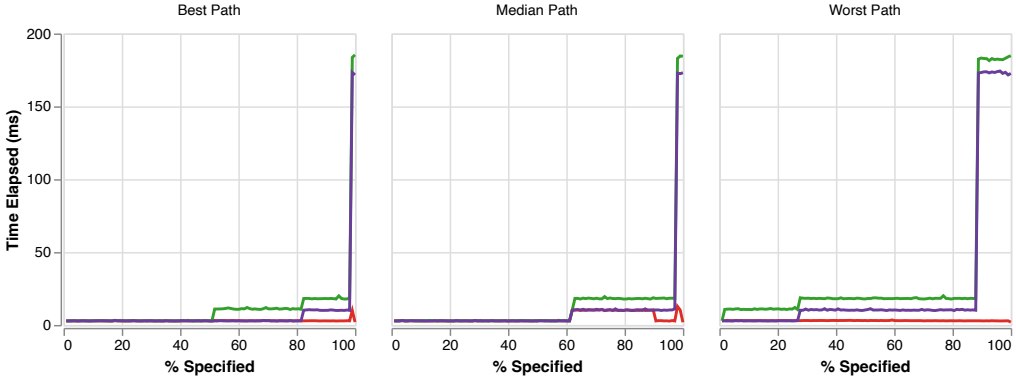
**RQ2: Pay-as-you-go Trends.** Fig. 12 shows how the total number of verification conditions (proof obligations) changes as more of each benchmark is specified (green curve). The figure also similarly shows the number of verification conditions that are statically verified as each benchmark is specified (purple curve). From the green curve, we see that even when there are no specifications, there are verification conditions, e.g. before fields are accessed, the object reference must be non-null and the program must own the field. 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)—pay-only-for-specs—but also, more of these verification conditions are discharged statically and do not have to be checked dynamically (purple curve)—benefit-as-you-go.

<sup>5</sup>These framing checks could fail, for example, if some function lower in the stack owns data that is accessed by the currently executing function.

**Binary Search Tree ( $\omega = 16$ )**



**Linked List ( $\omega = 32$ )**



**AVL ( $\omega = 16$ )**

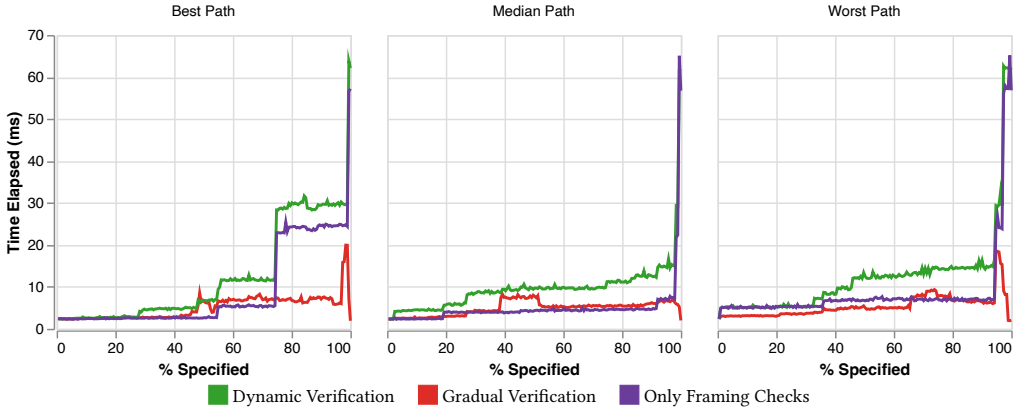


Fig. 11. The worst (maximum), median, and best (minimum) paths in terms of mean run-time observed at single step. Each graph shows a single path with performance results for the indicated example.

### Mean Verification Conditions

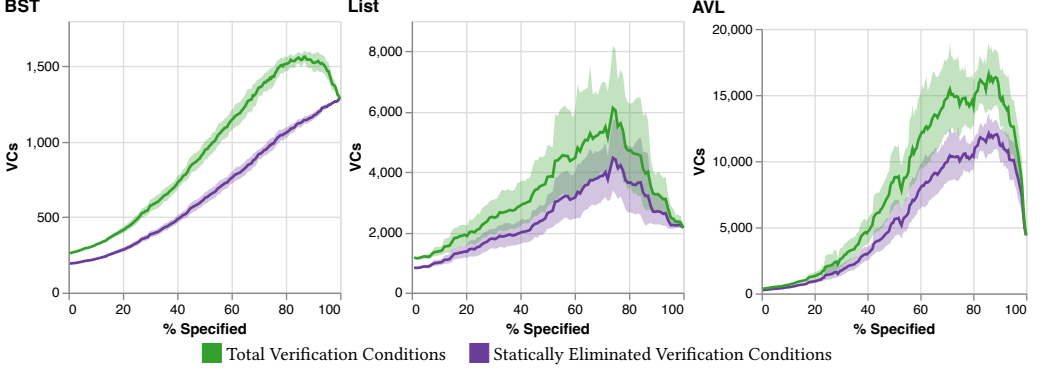


Fig. 12. For each example, the average quantity of verification conditions and the subset that were eliminated statically at each level of specification completeness across all paths sampled. Shading indicates the standard deviation.

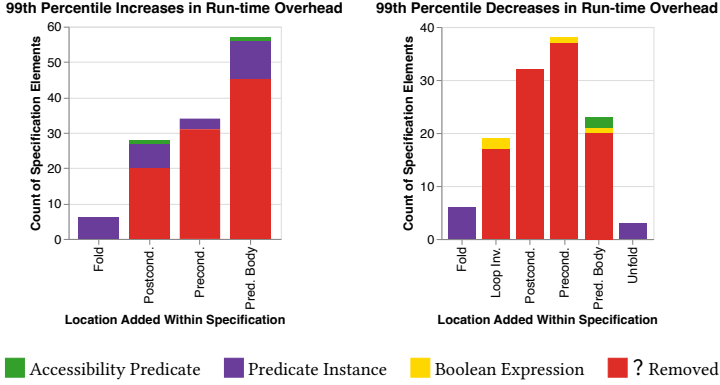


Fig. 13. The quantity of specification elements, grouped by type and location, that caused the highest ( $P_{99\%}$ ) increases and decreases in time elapsed out of every path sampled

Towards the end (right-side of the plots), the two curves converge until they meet when all the verification conditions are discharged statically—benefit-as-you-go. As a result, the answer to **RQ2** 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 execution paths with new static information.

Additionally, the plots in Fig. 11 display similar pay-as-you-go trends as those in Fig. 12. The green lines show that as more properties are specified, the cost of run-time verification increases for a fixed workload, aka. pay-only-for-specs. With Gradual C0, some of these properties can be checked statically; therefore, the run-time cost of gradual verification, shown in red, is generally lower than the cost of pure run-time verification. Further, the cost of gradual verification (the red line) tends to increase at first, because there are some properties that are specified but are not statically verifiable and checking these properties has a cost. The cost is always lower than a full dynamic approach, and once a critical mass of specifications have been written the total cost decreases—benefit-as-you-go. Although it is not shown in the plots, the cost of running the fully

specified version (conceptually, at the right-hand-side of each graph) is the same as the cost of running a raw C version of the benchmark.

Notably, all three lines (green, purple, and red) in all plots exhibit shelf-like jumps as more of a benchmark is specified. This is caused by the owned fields passing strategy employed at method boundaries (introduced in §3.3.3) to verify memory safety at run time. To respect precondition abstractions, only owned fields specified by a caller’s precondition are passed to the caller when the precondition is precise. Similarly, when a caller’s postcondition is precise, then only the owned fields specified by the postcondition are passed back to the callee. Computing owned fields from precise contracts is costly; and even more-so for contracts containing recursive predicates like in our benchmarks. Further, our benchmarks call such methods frequently during execution. As a result, run-time performance jumps up significantly at each path step where one of the aforementioned methods gets a precise pre or postcondition from ? removal. Fortunately, looking at the red lines Gradual C0 is not as heavily affected by this phenomena as the other verifiers—the jumps are less frequent and less costly. Additionally, the costly jumps do not occur before 80% specified where developers are likely to stop during the specification process. However, production gradual verifiers may choose to employ more optimal permission passing strategies.

*RQ3: Sources of Overhead.* Fig. 13 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 sub-figure, and ones that decrease run-time overhead significantly (top 1%) are counted in the right sub-figure. The count for accessibility predicates is colored in green, predicates in purple, boolean expressions in yellow, and ? removal in red. Clearly, removing ? from one step to another has the most significant impact on performance either way, whether it be increasing or decreasing run-time cost. When ? is removed from preconditions, postconditions, and predicate bodies, run-time cost significantly increases. This corresponds with the shelf-like jumps in Fig. 11 caused by our owned fields passing strategy. Removal of ? in the aforementioned locations leads to precise pre- and postconditions that trigger the use of our 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 Fig. 11. As such, the answer to **RQ3** is *yes*.

**4.4.1 Threats to Validity.** While the test programs we used are of sufficient complexity and result in interesting empirical trends, they do not generalize to 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. Only a small subset of the billions of possible verification paths were sampled due to computation constraints, and we did not use a formal criteria to choose our workload values. Finally, our partial specification generation strategy (§4.1) could be improved to further approximate those supported by the gradual guarantee. As such, while our results may indicate significant performance improvements by Gradual C0 over dynamic verification and interesting pay-as-you-go trends, our results would be stronger if the aforementioned weaknesses are addressed.

## 5 RELATED WORK

Much of the closely related work, particularly previous work on gradual verification [Bader et al. 2018; Wise et al. 2020], gradual typing [Herman et al. 2010; Siek and Taha 2006; Siek et al. 2015; Takikawa et al. 2016], and formal verification [Müller et al. 2016; Parkinson and Bierman 2005;

Reynolds 2002; Smans et al. 2009], has already been discussed throughout the paper. We thoroughly detail the differences between our work and Wise et al. [2020]’s work (the most closely related work to ours) in §1 and §2.

Another strongly related work is by Nguyen et al. [2014], which verifies dynamic contracts statically where possible and dynamically where necessary by utilizing symbolic execution. Unlike in Gradual C0, there is no notion of contract precision in Nguyen et al. [2014]’s work. Their approach uses 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, Nguyen et al. [2014]’s work is scoped to dynamic and functional languages, while our work focuses more on imperative languages. We also build in memory safety as a default, while Nguyen et al. [2014] does not.

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]. These systems focus on the needs of functional programs, 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 slow downs for structural types. Feltey et al. [2018]’s work reduces 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 §4 are promising, we may be able to draw from the extensive body of work in gradual typing to achieve further performance gains.

Finally, work in formal verification contains approaches that try to reduce the specification burden of users—a goal of Gradual C0. Furia and Meyer [2010]’s approach infers loop invariants with heuristics that weaken postconditions into invariants. When Furia and Meyer [2010]’s approach fails, verification also fails as invariants are missing. Similarly, several tools (Smallfoot [Berdine et al. 2005], jStar [Distefano and Parkinson J 2008], and Chalice [Leino et al. 2009]) use heuristics to infer fold and unfold statements for verification. In contrast, Gradual C0 will not fail solely because invariants, folds, or unfolds are missing. However, Gradual C0 may benefit from similar heuristic approaches by leveraging additional static information to reduce run-time overhead.

## 6 CONCLUSION

Gradual verification is a promising approach to supporting incrementality in proofs of programs. 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. Our paper overcomes several limitations of prior work to develop an implementable algorithm. The experimental results show that our approach can reduce overhead dramatically compared to purely dynamic checking and exhibits pay-as-you-go trends speculated in prior work. While more work remains to extend gradual verification (and Gradual C0) to the expressiveness of existing non-gradual verifiers, we believe the technology shows promise to make verification more adoptable in software development practice.

## REFERENCES

- 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 *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 25–46.
- Josh Berdine, Cristiano Calcagno, and Peter W O’hearn. 2005. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*. Springer, 115–137.



- Dino Distefano and Matthew J Parkinson J. 2008. jStar: Towards practical verification for Java. *ACM Sigplan Notices* 43, 10 (2008), 213–226.
- Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019. Approximate Normalization for Gradual Dependent Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 88 (jul 2019), 30 pages. <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. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 133 (oct 2018), 27 pages. <https://doi.org/10.1145/3276503>
- Carlo Alberto Furia and Bertrand Meyer. 2010. Inferring loop invariants using postconditions. In *Fields of logic and computation*. Springer, 277–300.
- David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. 23, 2 (June 2010), 167–189.
- Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. 2017. Closing the gap—the formally verified optimizing compiler CompCert. In *SSS'17: Safety-critical Systems Symposium 2017*. CreateSpace, 163–180.
- Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 207–220.
- Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. *SIGPLAN Not.* 52, 1 (jan 2017), 775–788. <https://doi.org/10.1145/3093333.3009856>
- K Rustan M Leino, Peter Müller, and Jan Smans. 2009. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*. Springer, 195–222.
- Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. <http://xavierleroy.org/publi/comp-cert-CACM.pdf>
- Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 56 (oct 2017), 30 pages. <https://doi.org/10.1145/3133880>
- P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (LNCS, Vol. 9583)*, B. Jobstmann and K. R. M. Leino (Eds.). Springer-Verlag, 41–62.
- 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 (Gothenburg, Sweden) (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 139–152. <https://doi.org/10.1145/2628136.2628156>
- Matthew Parkinson and Gavin Bierman. 2005. Separation logic and abstraction. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 247–258.
- John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*. IEEE, 55–74.
- Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. In *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 *LIPICs-Leibniz International Proceedings in Informatics*, Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit dynamic frames: Combining dynamic frames and separation logic. In *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 *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? *SIGPLAN Not.* 51, 1 (jan 2016), 456–468. <https://doi.org/10.1145/2914770.2837630>
- 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.

## A SUPPLEMENTARY MATERIAL FOR GRADUAL C0: SYMBOLIC EXECUTION FOR EFFICIENT GRADUAL VERIFICATION

```

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

```

Fig. 14. Path condition helper functions

```

eval-pc( $\sigma$ ,  $t$ ,  $Q$ )      =  $Q(\sigma$ ,  $t$ )
eval-pc( $\sigma$ ,  $x$ ,  $Q$ )     =  $Q(\sigma$ ,  $\sigma.Y(x)$ )
eval-pc( $\sigma_1$ ,  $op(\bar{e})$ ,  $Q$ ) = eval-pc( $\sigma_1$ ,  $\bar{e}$ , ( $\lambda$   $\sigma_2$ ,  $\bar{t}$  .  $Q(\sigma_2$ ,  $op'(\bar{t})$ )))
eval-pc( $\sigma_1$ ,  $e.f$ ,  $Q$ )  = eval-pc( $\sigma_1$ ,  $e$ , ( $\lambda$   $\sigma_2$ ,  $t$  .
    if ( $\exists f(r; \delta) \in \sigma_2.h$  . check( $\sigma_2.\pi$ ,  $r = t$ )) then
         $Q(\sigma_2$ ,  $\delta$ )
    else if ( $\exists f(r; \delta) \in \sigma_2.h?$  . check( $\sigma_2.\pi$ ,  $r = t$ )) then
         $Q(\sigma_2$ ,  $\delta$ )
    else if ( $\sigma_2.isImprecise$ ) then
         $e_t := \text{translate}(\sigma_2, t)$ 
         $\mathcal{R}' := \text{addcheck}(\sigma_2.\mathcal{R}, e.f, \text{acc}(e_t.f))$ 
         $Q(\sigma_2\{ \mathcal{R} := \mathcal{R}' \}, \text{fresh})$ 
    else failure())

```

Fig. 15. Rules for symbolically executing expressions without modifying the optimistic heap

---

### Algorithm 1 Generating minimal checks

---

```

1: function Diff( $\phi$ )
2:    $conjuncts \leftarrow CNF(\phi)$ 
3:    $\phi' \leftarrow \emptyset$ 
4:   for  $c \leftarrow conjuncts$  do
5:     if !check( $c$ ) then
6:        $\phi' \leftarrow \phi' + c$ 
7:     end if
8:   end for
9:   return  $\phi'$ 
10: end function

```

---

Fig. 16. Algorithm for computing the diff between two symbolic values

The `DIFF` function finds a minimal run-time check from an optimistically asserted formula containing statically known information. It accomplishes this by first performing a standard

**Algorithm 2** Variable resolution procedure

---

```

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

```

---

Fig. 17. TRANSLATE's procedure for resolving variables

transformation to conjunctive normal form (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.

The TRANSLATE 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 body for an (un)fold

statement. In these cases, an old store attached to the current symbolic state as described in 3.2.7 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 Fig. 17.

### A.1 Symbolic production of formulas

The rules for produce are given in Fig. 7. Essentially, `produce` takes a formula and snapshot  $\delta$  (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  $? \&\& \phi$  has its static part  $\phi$  produced into the current state  $\sigma$  alongside `second`( $\delta$ ). Note the snapshot  $\delta$  for an imprecise formula looks like  $(unit, second(\delta))$  where *unit* is the snapshot for  $?$  and `second`( $\delta$ ) is the snapshot for  $\phi$ . An imprecise formula also turns  $\sigma$  imprecise to produce the unknown information represented by  $?$  into  $\sigma$ . For example, if the state is represented by the formula  $\theta$ , then this rule results in  $? \&\& \theta \&\& \phi$ . A symbolic value  $t$  is produced into the path condition of the current state  $\sigma$ . Also, the snapshot  $\delta$  for  $t$  must be *unit*, so this fact is also stored in  $\sigma$ 's path condition. Then,  $\sigma$  is passed to  $Q$ .

The `produce` rule for expression  $e$ , first evaluates  $e$  to its symbolic value  $t$  using `eval-pc`. Then,  $t$  is produced into the path condition of the current state  $\sigma_2$  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-pc`. 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  $? \&\& true \&\& e$ . As highlighted in blue, the run-time checks generated during the evaluation of  $e$  are dropped before continuing execution. The goal of `produce` is not to assert information in the state, but rather add information to the state. Further, it is sound to ignore these run-time checks, reducing check overhead.

The rules for producing field and predicate accessibility predicates into the state  $\sigma_1$  operate in a very similar manner. Thus, we will focus on the rule for fields only. The field  $e.f$  in `acc`( $e.f$ ) first has its receiver  $e$  evaluated to  $t$  by `eval-pc`, resulting in  $\sigma_2$ . Then, using the parameter  $\delta$  a fresh heap chunk  $f(t; \delta)$  is created and added to  $\sigma_2$ 's heap  $h$ , which represents `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, `acc`( $e.f$ ) implies  $e \neq null$  and so that fact is recorded in  $\sigma_2$ 's path condition as  $t \neq null$ . Again, as highlighted in blue, run-time checks from `eval-pc` are not recorded.

When the separating conjunction  $\phi_1 \&\& \phi_2$  is produced,  $\phi_1$  is first produced and then afterwards  $\phi_2$  is produced into the resulting symbolic state. Note that the snapshot  $\delta$  is split between the two formulas using `first`( $\delta$ ) and `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  $\phi_1$  is produced into the state, and along the other path only the false branch  $\phi_2$  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 Fig. 18 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

```

branch( $\sigma$ ,  $e$ ,  $t$ ,  $Q_t$ ,  $Q_{\neg t}$ ) =
  ( $\pi_T$ ,  $\mathcal{R}_T$ ) := (pc-push( $\sigma.\pi$ , fresh,  $t$ ), addbc( $\sigma.\mathcal{R}$ ,  $e$ ,  $e$ ))
  ( $\pi_F$ ,  $\mathcal{R}_F$ ) := (pc-push( $\sigma.\pi$ , fresh,  $\neg t$ ), addbc( $\sigma.\mathcal{R}$ ,  $e$ ,  $\neg e$ ))
  if ( $\sigma.isImprecise$ ) then
     $res_T$  := (if  $\neg$ check( $\sigma.\pi$ ,  $\neg t$ ) then  $Q_t(\sigma\{\pi := \pi_T, \mathcal{R} := \mathcal{R}_T\})$  else failure())
     $res_F$  := (if  $\neg$ check( $\sigma.\pi$ ,  $t$ ) then  $Q_{\neg t}(\sigma\{\pi := \pi_F, \mathcal{R} := \mathcal{R}_F\})$  else failure())
    if (( $res_T \wedge \neg res_F$ )  $\vee$  ( $\neg res_T \wedge res_F$ )) then
       $\mathcal{R}'$  := addcheck( $\sigma.\mathcal{R}$ ,  $e$ , (if ( $res_T$ ) then  $e$  else  $\neg e$ ))
       $\mathfrak{R}$  :=  $\mathfrak{R} \cup \mathcal{R}'.rcs.last$ 
       $res_T \vee res_F$ 
    else
      (if  $\neg$ check( $\sigma.\pi$ ,  $\neg t$ ) then  $Q_t(\sigma\{\pi := \pi_T, \mathcal{R} := \mathcal{R}_T\})$  else success())  $\wedge$ 
      (if  $\neg$ check( $\sigma.\pi$ ,  $t$ ) then  $Q_{\neg t}(\sigma\{\pi := \pi_F, \mathcal{R} := \mathcal{R}_F\})$  else success())

```

Handles imprecision
Handles run-time check generation and collection

Fig. 18. branch function definition

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  $\sigma$  (the current state) is precise. However, when  $\sigma$  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.2 Symbolic consumption of formulas

```

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

```

Handles imprecision
Handles run-time check generation and collection

Fig. 19. Rules for symbolically consuming formulas (1/3)

```

consume'(σ, f?, h?, h, (e, t), Q) = res,  $\bar{t}$  := assert(σ.isImprecise, σ.π, t)
 $\mathcal{R}' := \text{addcheck}(\sigma.\mathcal{R}, e, \text{translate}(\sigma, \bar{t}))$ 
res ∧ Q(σ{  $\mathcal{R} := \mathcal{R}'$  }, h?, h, unit)

consume'(σ1, f?, h?, h, e, Q) = eval-pc(σ1{ isImprecise := f? }, e, (λ σ2, t .
consume'(σ2{ isImprecise := σ1.isImprecise }, f?, h?, h, (e, t), Q)))

consume'(σ1, f?, h?, h, acc(p( $\bar{e}$ )), Q) = eval-pc(σ1{ isImprecise := f? },  $\bar{e}$ , (λ σ2,  $\bar{t}$  .
σ3 := σ2{ isImprecise := σ1.isImprecise }
(h1, δ1, b1) := heap-rem-pred(σ3.isImprecise, h, σ3.π, p( $\bar{t}$ ))
if (σ3.isImprecise) then
  (h'?, δ2, b2) := heap-rem-pred(σ3.isImprecise, h?, σ3.π, p( $\bar{t}$ ))
  if (b1 = b2 = false) then
     $\mathcal{R}' := \text{addcheck}(\sigma_3.\mathcal{R}, \text{acc}(p(\bar{e})), \text{acc}(p(\bar{e})))$ 
  else  $\mathcal{R}' := \sigma_3.\mathcal{R}$ 
  Q(σ3{  $\mathcal{R} := \mathcal{R}'$  }, h'?, h1, (if (b1) then δ1 else δ2))
else if (b1) then Q(σ3, σ3.h?, h1, δ1)
else failure()))

consume'(σ1, f?, h?, h, acc(e.f), Q) = eval-pc(σ1{ isImprecise := f? }, e, (λ σ2, t .
σ3 := σ2{ isImprecise := σ1.isImprecise }
res,  $\bar{t}$  := assert(σ3.isImprecise, σ3.π, t ≠ null)
res ∧ (
 $\mathcal{R}' := \text{addcheck}(\sigma_3.\mathcal{R}, \text{acc}(e.f), \text{translate}(\sigma_3, \bar{t}))$ 
(h1, δ1, b1) := heap-rem-acc(σ3.isImprecise, h, σ3.π, f(t))
if (σ3.isImprecise) then
  (h'?, δ2, b2) := heap-rem-acc(σ3.isImprecise, h?, σ3.π, f(t))
  if (b1 = b2 = false) then
     $\mathcal{R}'' := \text{addcheck}(\mathcal{R}', \text{acc}(e.f), \text{acc}(\text{translate}(\sigma_3, t).f))$ 
  else  $\mathcal{R}'' := \mathcal{R}'$ 
  Q(σ3{  $\mathcal{R} := \mathcal{R}''$  }, h'?, h1, (if (b1) then δ1 else δ2))
else if (b1) then Q(σ3{  $\mathcal{R} := \mathcal{R}'$  }, σ3.h?, h1, δ1)
else failure()))

```

■ Handles imprecision   ■ Handles run-time check generation and collection

Fig. 19. Rules for symbolically consuming formulas (2/3)

The goals of consume are 3-fold: 1) given a symbolic state  $\sigma$  and formula  $\tilde{\phi}$  check whether  $\tilde{\phi}$  is established by  $\sigma$ , i.e.  $\tilde{\phi}_\sigma \widetilde{\Rightarrow} \tilde{\phi}$  where  $\tilde{\phi}_\sigma$  is the formula which represents the state  $\sigma$ , 2) produce and collect run-time checks that are minimally sufficient for  $\sigma$  to establish  $\tilde{\phi}$  soundly, i.e. the red and green highlighting in Fig. 3, and 3) remove accessibility predicates and predicates that are asserted



$$\begin{aligned}
\text{consume}'(\sigma_1, \boxed{f_?}, \boxed{h_?}, h, \phi_1 \&\& \phi_2, Q) &= \text{consume}'(\sigma_1, \boxed{f_?}, \boxed{h_?}, h, \phi_1, (\lambda \sigma_2, \boxed{h'_?}, h', \delta_1 . \\
&\quad \text{consume}'(\sigma_2, \boxed{f_?}, \boxed{h'_?}, h', \phi_2, (\lambda \sigma_3, \boxed{h''_?}, h'', \delta_2 . \\
&\quad \quad Q(\sigma_3, \boxed{h''_?}, h'', \text{pair}(\delta_1, \delta_2)))))) \\
\text{consume}'(\sigma_1, \boxed{f_?}, \boxed{h_?}, h, e ? \phi_1 : \phi_2, Q) &= \text{eval-pc}(\sigma_1 \{ \text{isImprecise} := \boxed{f_?} \}, e, (\lambda \sigma_2, t . \\
&\quad \boxed{\sigma_3 := \sigma_2 \{ \text{isImprecise} := \sigma_1.\text{isImprecise} \}} \\
&\quad \text{branch}(\sigma_3, e, t, \\
&\quad \quad (\lambda \sigma_4 . \text{consume}'(\sigma_4, \boxed{f_?}, \boxed{h_?}, h, \phi_1, Q)), \\
&\quad \quad (\lambda \sigma_4 . \text{consume}'(\sigma_4, \boxed{f_?}, \boxed{h_?}, h, \phi_2, Q))))))
\end{aligned}$$

■ Handles imprecision
■ Handles run-time check generation and collection

Fig. 19. Rules for symbolically consuming formulas (3/3)

in  $\tilde{\phi}$  from  $\sigma$ . Note that  $\widetilde{\Rightarrow}$  is the consistent implication described in §2 and formally defined by Wise et al. [2020]. The rules for consume are given in Fig. 19.

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  $x \neq \text{null}$ . Therefore, such information is added to the path condition  $\sigma_1.\pi$  during consolidation. 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 [Müller et al. 2016], without repeating it here.

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

Rules for  $\text{consume}'$  can also be found in Fig. 19. Cases for expressions  $e$ , the separating conjunction  $\phi_1 \&\& \phi_2$ , 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,  $\phi_1$  is consumed first, then afterward  $\phi_2$  is consumed. The rule for logical conditionals evaluates the condition  $e$  to a symbolic value, and then uses the branch function to consume  $\phi_1$  and  $\phi_2$  along different execution paths. The case for  $\text{acc}(p(\bar{e}))$  is also very similar to the case for  $\text{acc}(e.f)$  that we discuss later in this section.

When a symbolic value  $t$  is consumed, the current state  $\sigma$  must establish  $t$ , i.e.  $\sigma \widetilde{\Rightarrow} t$ , or verification fails. The assert function (defined in Fig. 21) implements this functionality. In particular, assert returns `success()` when  $\pi$  can statically prove  $t$  or when  $\sigma$  is imprecise and  $t$  does not contradict

constraints in  $\pi$ —here,  $t$  is optimistically assumed to be true. Otherwise, `assert` returns `failure()`. When `assert` succeeds, it also returns a set of symbolic values  $\bar{t}$  that are residuals of  $t$  that cannot be proved statically by  $\pi$ . If  $t$  is proven entirely statically, then `assert` returns `true`. A run-time check is created for the residuals  $\bar{t}$  and is added to  $\sigma$  to be passed to the continuation  $Q$ . Note that `translate` is used to create an expression from  $\bar{t}$  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  $\text{acc}(e.f)$ , first evaluates the receiver  $e$  to  $t$  using `eval-pc`, the current state  $\sigma_1$ , and the parameter  $f_?$ . The parameter  $f_?$  is the boolean flag mentioned previously. Assigning  $f_?$  to  $\sigma_1.\text{isImprecise}$  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  $\sigma_3$ , and `assert` is used to ensure the receiver  $t$  is non-null. If  $t \neq \text{null}$  is optimistically true, a run-time check for  $t \neq \text{null}$  at location  $\text{acc}(e.f)$  is created and added to  $\sigma_3.\mathcal{R}$ . Next, `heap-rem-acc` is used to remove the heap chunks from heap  $h$  that overlap with or may potentially overlap with  $\text{acc}(e.f)$  in memory. The `heap-rem-acc` function is formally defined alongside a similar function for predicates (`heap-rem-pred`) in the Fig. 20. If a field chunk is not statically proven to be disjoint from  $\text{acc}(e.f)$ , then it is removed. Further, since predicates are opaque, Gradual Viper cannot tell whether or not their predicate bodies overlap with  $\text{acc}(e.f)$ . Therefore, predicate chunks are almost always considered to potentially overlap with  $\text{acc}(e.f)$ . 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  $\text{acc}(e.f)$  has a corresponding heap chunk in  $h$ . If so, its snapshot  $\delta_1$  is returned and  $b_1$  is assigned `true`. Otherwise, a fresh snapshot is returned with `false`. If the current state  $\sigma_3$  is imprecise, then heap chunks are similarly removed from  $h_?$  and  $\text{acc}(e.f)$  is checked for existence in  $h_?$ . If a field chunk for  $\text{acc}(e.f)$  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  $\text{acc}(e.f)$ 's snapshot. Without imprecision, `consume'` will fail when a field chunk for  $\text{acc}(e.f)$  is not found in  $h$ .

```

heap-rem-pred(isImprecise, h,  $\pi$ ,  $p(\bar{t})$ ) = if  $\exists (p(\bar{r}; \delta) \in h \cdot \text{check}(\pi, \bigwedge \bar{t} = \bar{r}))$  then
    ( $h \setminus \{p(\bar{r}; \delta)\}$ ,  $\delta$ , true)
else ( $\emptyset$ , fresh, false)

heap-rem-acc(isImprecise, h,  $\pi$ ,  $f(t)$ ) =  $h' := \text{foldl}(h, \emptyset, (\lambda f_{src}(\bar{r}; \delta), h_{dst} \cdot$ 
    if  $(\neg(|\bar{r}| = 1) \parallel \neg(f = f_{src}) \parallel \neg\text{check}(\text{isImprecise}, \pi, t = r))$  then
         $h_{dst} \cup f_{src}(\bar{r}; \delta)$ 
    else  $h_{dst}))$ 
    if  $\exists f(r; \delta) \in h \cdot \text{check}(\pi, t = r)$  then
        ( $h'$ ,  $\delta$ , true)
    else
         $h' := \text{foldl}(h', \emptyset, (\lambda f_{src}(\bar{r}; \delta), h_{dst} \cdot$ 
            if  $(f_{src}(\bar{r}; \delta)$  is a field chunk) then
                 $h_{dst} \cup f_{src}(\bar{r}; \delta)$ 
            else  $h_{dst}))$ 
        ( $h'$ , fresh, false)

```

Fig. 20. Heap remove function definitions

$$\begin{aligned}
& \text{check}(\pi, t) = \text{pc-all}(\pi) \Rightarrow t \\
& \text{check}(\text{isImprecise}, \pi, t) = \begin{cases} \text{true, true} & \text{if } \text{check}(\pi, t) \\ \text{true, diff}(\text{pc-all}(\pi), t) & \text{if } (\text{isImprecise} \wedge (\bigwedge \text{pc-all}(\pi) \wedge t)_{\text{SAT}}) \\ \text{false, } \emptyset & \text{otherwise} \end{cases} \\
& \text{assert}(\text{isImprecise}, \pi, t) = \begin{cases} \text{success}(), \bar{t} & \text{if } (b = \text{true}) \text{ where } b, \bar{t} := \text{check}(\text{isImprecise}, \pi, t) \\ \text{failure}(), \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

Handles imprecision
  Handles run-time check generation and collection

Fig. 21. Check and assert function definitions

### A.3 Symbolic execution of statements

The exec rules 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  $\sigma_1$  and eval. Any field reads in  $e$  are either directly or optimistically verified using  $\sigma_1$ . Then, the resulting state  $\sigma_2$  must establish write access to  $x.f$  in consume, i.e.  $\sigma_2 \approx \text{acc}(x.f)$ . The call to consume also removes the field chunk for  $\text{acc}(x.f)$  from  $\sigma_2$  (if it is in there) resulting in  $\sigma_3$ . Therefore, the call to produce can safely add a fresh field chunk for  $\text{acc}(x.f)$  alongside  $x.f = t$  to  $\sigma_3$  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  $\bar{e}$  to symbolic values  $\bar{t}$ , asserts the method's precondition  $\text{meth}_{pre}$  holds in the current state, consumes the heap chunks in the precondition, and produces the method's postcondition  $\text{meth}_{post}$  into the continuation. Run-time checks are also collected where necessary (under the hood) and passed to the continuation. Note that the origin field of  $\mathcal{R}$  is set to  $\bar{z} := m(\bar{e})$  before consuming  $\text{meth}_{pre}$  and reset to none after producing  $\text{meth}_{post}$ . Setting the origin indicates that run-time checks or branch conditions for  $\text{meth}_{pre}$  or  $\text{meth}_{post}$  should be attached to the method call statement rather than where they are declared. The origin arguments  $\sigma_2$  and  $\bar{t}$  are used to reverse the substitution  $[\text{meth}_{args} \mapsto t]$  in run-time checks and branch conditions for  $\text{meth}_{pre}$  and  $\text{meth}_{post}$ . The rule for (un)folding predicates operates the same as for method calls where  $\text{meth}_{pre}$  is the predicate body (predicate instance) and  $\text{meth}_{post}$  is the predicate instance (predicate body). The origin is set to fold  $\text{acc}(p(\bar{e}))$  and unfold  $\text{acc}(p(\bar{e}))$  respectively.

In contrast,  $\phi$  in assert  $\phi$  maintains a none origin field, because  $\phi$ 's use and declaration align at the same program location assert  $\phi$ . The assert rule relies on consume to assert  $\phi$  holds in the current state  $\sigma_1$ . If the consume succeeds, the state  $\sigma_1$  is passed to the continuation nearly unmodified. Path condition constraints from  $\phi$  hold in  $\sigma_1$  either directly or optimistically. Therefore, these constraints are added to  $\sigma_1$  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  $\phi$  is

$$\begin{aligned}
& \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, \text{var } x : T, Q) &= Q(\sigma\{y := \text{havoc}(\sigma.y, x)\}) \\
& \text{exec}(\sigma_1, x := e, Q) &= \text{eval}(\sigma_1, e, (\lambda \sigma_2, t. Q(\sigma_2\{y := \sigma_2.y[x \mapsto t]\})) \\
& \text{exec}(\sigma_1, x.f := e, Q) &= \text{eval}(\sigma_1, e, (\lambda \sigma_2, t. \text{consume}(\sigma_2, \text{acc}(x.f), (\lambda \sigma_3, \_ . \\
& & \quad \text{produce}(\sigma_3, \text{acc}(x.f) \&\& x.f = t, \text{pair}(\text{fresh}, \text{unit}), Q)))) \\
& \text{exec}(\sigma, x := \text{new}(\bar{f}), Q) &= \text{produce}(\sigma\{y := \text{havoc}(\sigma.y, x)\}, \overline{\text{acc}(x.f)}, \text{fresh}, Q) \\
& \text{exec}(\sigma_1, \bar{z} := m(\bar{e}), Q) &= \text{eval}(\sigma_1, \bar{e}, (\lambda \sigma_2, \bar{t}. \\
& & \quad \mathcal{R}' := \sigma_2.\mathcal{R}\{\text{origin} := (\sigma_2, \bar{z} := m(\bar{e}), \bar{t})\} \\
& & \quad \text{consume}(\sigma_2\{\mathcal{R} := \mathcal{R}'\}, \text{meth}_{pre}[\overline{\text{meth}_{args} \mapsto t}], (\lambda \sigma_3, \delta. \\
& & \quad \sigma_4 := \sigma_3\{y := \text{havoc}(\sigma_3.y, \bar{z})\} \\
& & \quad \text{produce}(\sigma_4, \text{meth}_{post}[\overline{\text{meth}_{args} \mapsto t}][\overline{\text{meth}_{ret} \mapsto z}], \text{fresh}, \\
& & \quad (\lambda \sigma_5. Q(\sigma_5\{\mathcal{R} := \sigma_5.\mathcal{R}\{\text{origin} := \text{none}\}\})))) \\
& \text{exec}(\sigma_1, \text{assert } \phi, Q) &= \text{consume}(\sigma_1, \phi, (\lambda \sigma_2, \delta. \\
& & \quad \text{well-formed}(\sigma_2, ? \&\& \phi, \delta, (\lambda \sigma_3. \\
& & \quad Q(\sigma_1\{\pi := \sigma_3.\pi, \mathcal{R} := \sigma_3.\mathcal{R}\})))) \\
& \text{exec}(\sigma_1, \text{fold } \text{acc}(p(\bar{e})), Q) &= \text{eval}(\sigma_1, \bar{e}, (\lambda \sigma_2, \bar{t}. \\
& & \quad \mathcal{R}' := \sigma_2.\mathcal{R}\{\text{origin} := (\sigma_2, \text{fold } \text{acc}(p(\bar{e})), \bar{t})\} \\
& & \quad \text{consume}(\sigma_2\{\mathcal{R} := \mathcal{R}'\}, \text{pred}_{body}[\overline{\text{pred}_{args} \mapsto t}], (\lambda \sigma_3, \delta. \\
& & \quad \text{produce}(\sigma_3\{\mathcal{R} := \sigma_3.\mathcal{R}\{\text{origin} := \text{none}\}\}, \text{acc}(p(\bar{t})), \delta, \\
& & \quad Q)))) \\
& \text{exec}(\sigma_1, \text{unfold } \text{acc}(p(\bar{e})), Q) &= \text{eval}(\sigma_1, \bar{e}, (\lambda \sigma_2, \bar{t}. \\
& & \quad \mathcal{R}' := \sigma_2.\mathcal{R}\{\text{origin} := (\sigma_2, \text{unfold } \text{acc}(p(\bar{e})), \bar{t})\} \\
& & \quad \text{consume}(\sigma_2\{\mathcal{R} := \mathcal{R}'\}, \text{acc}(p(\bar{t})), (\lambda \sigma_3, \delta. \\
& & \quad \text{produce}(\sigma_3, \text{pred}_{body}[\overline{\text{pred}_{args} \mapsto t}], \delta, (\lambda \sigma_4. \\
& & \quad Q(\sigma_4\{\mathcal{R} := \sigma_4.\mathcal{R}\{\text{origin} := \text{none}\}\})))) \\
& \text{exec}(\sigma_1, \text{if } (e) \{ \text{stmt}_1 \} \text{ else } \{ \text{stmt}_2 \}, Q) &= \text{eval}(\sigma_1, e, (\lambda \sigma_2, t. \\
& & \quad \text{branch}(\sigma_2, e, t, (\lambda \sigma_3. \text{exec}(\sigma_3, \text{stmt}_1, Q)), (\lambda \sigma_3. \text{exec}(\sigma_3, \text{stmt}_2, Q))))
\end{aligned}$$

■ Handles imprecision

■ Handles run-time check generation and collection

Fig. 22. Rules for symbolically executing program statements (1/2)

checked for well-formedness here (Fig. 23). 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,  $\phi$  does not need to be self-framed, and so it is joined with ? in the call to well-formed. ? verifies access to all of  $\phi$ '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 §A.1.

$\text{exec}(\sigma_1, \text{while } (e) \text{ invariant } \tilde{\phi} \{ \text{stmt} \}, Q) = \gamma_2 := \text{havoc}(\sigma_1.\gamma, \bar{x})$   
 $\text{resbody} := \text{well-formed}(\sigma_1 \{ \text{isImprecise} := \text{false}, h_\gamma := \emptyset, h := \emptyset, \gamma := \gamma_2, \\ \mathcal{R} := \sigma_1.\mathcal{R} \{ \text{origin} := (\sigma_1, \text{while } (e) \text{ invariant } \tilde{\phi} \{ \text{stmt} \}, \text{beginning}) \}, \\ \tilde{\phi} \&\& e, \text{fresh}, \\ (\lambda \sigma_3 \{ \mathcal{R} := \sigma_3.\mathcal{R} \{ \text{origin} := \text{none} \} \} . \text{exec}(\sigma_3, \text{stmt}, (\lambda \sigma_4 . \\ \text{consume}(\sigma_4 \{ \mathcal{R} := \sigma_4.\mathcal{R} \{ \text{origin} := (\sigma_4, \text{while } (e) \text{ invariant } \tilde{\phi} \{ \text{stmt} \}, \text{end}) \}, \\ \tilde{\phi}, (\lambda \sigma_5, \_ . \mathfrak{R} := \mathfrak{R} \cup \sigma_5.\mathcal{R}.\text{rcs} \\ \text{success}()))))) \\ \text{resafter} := \text{consume}(\sigma_1 \{ \mathcal{R} := \sigma_1.\mathcal{R} \{ \text{origin} := (\sigma_4, \text{while } (e) \text{ invariant } \tilde{\phi} \{ \text{stmt} \}, \text{before}) \}, \tilde{\phi}, \\ (\lambda \sigma_2, \_ . \text{produce}(\sigma_2 \{ \gamma := \gamma_2, \sigma_2.\mathcal{R} \{ \text{origin} := (\sigma_2, \text{while } (e) \text{ invariant } \tilde{\phi} \{ \text{stmt} \}, \text{after}) \}, \\ \tilde{\phi} \&\&!e, \text{fresh}, Q))) \\ \text{if } (\sigma_1.\text{isImprecise}) \text{ then} \\ \quad \text{if } (\neg \text{resbody} \wedge \text{resafter}) \text{ then} \\ \quad \quad \mathcal{R}' := \text{addcheck}(\sigma_1.\mathcal{R}, \text{while } (e) \text{ invariant } \tilde{\phi} \{ \text{stmt} \}, \text{before}, e) \\ \quad \quad \mathfrak{R} := \mathfrak{R} \cup \mathcal{R}'.\text{rcs}.\text{last} \\ \quad \quad (\neg \text{resbody} \vee \text{resafter}) \wedge (\text{resbody} \vee \text{resafter}) \\ \text{else} \\ \quad \text{resbody} \wedge \text{resafter} \\ \text{where } \bar{x} \text{ are variables modified by the loop body}$

■	Handles imprecision	■	Handles run-time check generation and collection
---	---------------------	---	--

Fig. 22. Rules for symbolically executing program statements (2/2)

$$\begin{aligned}
 \text{well-formed}(\sigma_1, \tilde{\phi}, \delta, Q) = & \text{produce}(\sigma_1, \tilde{\phi}, \delta, (\lambda \sigma_2 . \\
 & \text{produce}(\sigma_1 \{ \pi := \sigma_2.\pi \}, \tilde{\phi}, \delta, Q)))
 \end{aligned}$$

Fig. 23. Well-formed formula function definition

#### A.4 Valid program

A Gradual Viper program is valid if all of its method and predicate declarations are verified successfully as defined in Fig. 24. In particular, a method  $m$ 's declaration is verified first by checking well-formedness of  $m$ 's precondition  $\text{meth}_{\text{pre}}$  and postcondition  $\text{meth}_{\text{post}}$  using the empty state  $\sigma_0$  (well-formedness is described in §A.3). Note, fresh symbolic values are created and added to  $\sigma_0$  for  $m$ 's argument variables  $\bar{x}$  and return variables  $\bar{y}$ . If  $\text{meth}_{\text{pre}}$  and  $\text{meth}_{\text{post}}$  are well-formed, then the body of  $m$  ( $\text{meth}_{\text{body}}$ ) is symbolically executed (§A.3) starting with the symbolic state  $\sigma_1$  containing  $\text{meth}_{\text{pre}}$ . Recall, well-formed additionally produces the formula that is being checked into the symbolic state. The symbolic state  $\sigma_2$  is produced after the symbolic execution of  $\text{meth}_{\text{body}}$ . Then,  $\text{meth}_{\text{post}}$  is checked for validity against  $\sigma_2$ , i.e.  $\sigma_2$  must establish  $\text{meth}_{\text{post}}$  (§A.2). If  $\text{meth}_{\text{post}}$  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  $\text{pred}_{\text{body}}$  is well-formed. As before, fresh symbolic values are created for  $p$ 's argument variables  $\bar{x}$ .

$$\begin{aligned}
\text{verify}(\text{method } m(\overline{x:T}) \text{ returns } (\overline{y:T})) &= \text{well-formed}(\sigma_0\{\gamma := \sigma_0.\gamma[\overline{x \mapsto \text{fresh}}][\overline{y \mapsto \text{fresh}}]\}, \text{meth}_{pre}, \text{fresh}, (\lambda \sigma_1 . \\
&\quad \text{well-formed}(\sigma_1\{\text{isImprecise} := \text{false}, h_? := \emptyset, h := \emptyset\}, \text{meth}_{post}, \\
&\quad \text{fresh}, (\lambda _ . \text{success()})) \\
&\quad \wedge \\
&\quad \text{exec}(\sigma_1, \text{meth}_{body}, (\lambda \sigma_2 . \\
&\quad \quad \text{consume}(\sigma_2, \text{meth}_{post}, (\lambda \sigma_3, _ . \\
&\quad \quad \quad \mathfrak{R} := \mathfrak{R} \cup \sigma_3.\mathcal{R}.\text{rccs} ; \text{success()})))))) \\
\text{verify}(\text{predicate } p(\overline{x:T})) &= \text{well-formed}(\sigma_0\{\gamma := \sigma_0.\gamma[\overline{x \mapsto \text{fresh}}]\}, \text{pred}_{body}, \text{fresh}, (\lambda _ . \text{success()}))
\end{aligned}$$

■ Handles imprecision
■ Handles run-time check generation and collection

Fig. 24. Rules defining a valid Gradual Viper program

Note, no run-time checks are added to  $\mathfrak{R}$  here, because well-formedness checks do not produce any run-time checks.