Gradual Verification of Recursive Heap Data Structures

JENNA WISE, Carnegie Mellon University, USA
JOHANNES BADER, Facebook, USA
JONATHAN ALDRICH, Carnegie Mellon University, USA
ÉRIC TANTER, University of Chile, Chile
JOSHUA SUNSHINE, Carnegie Mellon University, USA

Static verification tools for recursive heap data structures impose significant annotation burden on developers. This is true even when verifying a simple function that inserts an element at the end of a linked list. Gradual verification was introduced to allow developers to deal with this burden incrementally, if at all. It draws from research on gradual typing to produce a verification system that supports imprecise specifications along a continuum. Our work extends the prior approach to gradual verification to support the specification and verification of programs that use basic recursive data structures like trees or lists. This paper outlines work in progress on this extension using examples, and highlights research challenges and proposes solutions to them.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages*;

Additional Key Words and Phrases: gradual typing, gradual verification, implicit dynamic frames, recursive predicates, separation logic

ACM Reference Format:
Jenna Wise, Johannes Bader, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual Verification of Recursive Heap Data Structures. 1, 1 (January 2020), 13 pages.

1 INTRODUCTION

Traditional static verification techniques often rely on user specifications of system components (method or function pre- and postconditions, loop invariants, etc.) expressed in some logic (e.g. Hoare logic [Hoare 1969]) to ensure a software system adheres to its specifications at run time. Techniques aimed at verifying the manipulation of recursive heap data structures (trees, linked-lists, graphs, etc.) are based on either implicit dynamic frames (IDF) [Smans et al. 2009] or separation logic [Reynolds 2002] extended with recursive abstract predicates [Parkinson and Bierman 2005] or higher order encodings. Without a resource logic and recursive predicates it is impossible to statically and modularly verify any interesting properties of programs containing such data structures (e.g. whether a list remains sorted before or after manipulation or whether a tree is a binary tree before or after manipulation). Unfortunately, tools implementing these techniques require a significant annotation effort to support inductive proofs of correctness. This is true even for a simple function that inserts an element at the end of a linked list.

Fortunately, Bader et al. [2018] introduced a sound verification approach that allows developers to deal with specification burdens incrementally, if at all. Bader *et al.* draw on research in *gradual typing* [Garcia et al. 2016; Siek and Taha 2007, 2006] to produce *gradual verification*, which extends a static verification system with support for *imprecise specifications* – a static specification joined with ?. The resulting static verification system warns only about inconsistencies between specifications and code; it does not produce warnings due to missing information in specifications. Instead, missing information is dynamically verified. Gradual verification adheres to *gradual guarantees* that ensure developers can choose their desired level of precision without artificial constraints imposed

2020. XXXX-XXXX/2020/1-ART
https://doi.org/

by the verification technology. These guarantees are inspired by the corresponding gradual typing properties formulated by Siek et al. [2015].

This paper builds on prior work [Bader et al. 2018] by exploring the applicability of gradual verification to programs manipulating recursive heap data structures. This exploration involves walking through multiple attempts at verifying linked list insertion with an extended gradual verification approach that is currently under development. Each attempt focuses on ways developers could reduce or ignore annotation burden. Unique challenges for gradual verification based on IDF and recursive abstract predicates arise from this process: 1) ? in imprecise specifications must represent accessibility predicates and predicate instances needed for static verification, 2) accessibility predicates and predicate instances must be dynamically verified, and 3) dynamically verifying specifications with accessibility predicates and predicate instances can incur significant runtime overhead. To overcome these challenges, we suggest: 1) adjusting Bader *et al.*'s concretization definition for gradual formulas to rely on an *iso-recursive* interpretation of predicate instances and require that concretizations of gradual formulas be *self-framed*, 2) verifying accessibility predicates by tracking and updating a set of heap locations at runtime and verifying predicate instances *equi-recursively*, and 3) using language design to avoid particular specifications that incur runtime overhead and exploring further optimizations in future work.

The rest of this paper is outlined as follows. The annotation burden induced by statically verifying linked list insertion is discussed in Section 2. Section 3 illustrates how this burden can be reduced or eliminated with gradual verification by using examples, and Section 4 discusses challenges and solutions to supporting such examples. Finally, Sections 5 and 6 further relate this paper to prior work and discuss future work, respectively.

2 THE BURDEN OF STATIC VERIFICATION

We will use the program in Example 2.1 to illustrate why burdensome specifications are often required when statically verifying programs containing recursive heap data structures. The program implements a linked list and two methods, `insertLast` and `insertLastHelper`, for inserting an element at the end of a list. Notice that `insertLastHelper` iteratively traverses a list for insertion.

Example 2.1.

```
class Node { int val; Node next; }

class List {
  Node head;

  void insertLast(int val)
  {
    if (this.head == null) {
      this.head = new Node(val,null);
    } else {
      insertLastHelper(val);
    }
  }

  void insertLastHelper(int val)
  {
    Node y = this.head;
    while (y.next != null)
      { y = y.next; }
    y.next = new Node(val,null);
  }
}
```

Verifying `insertLast` and `insertLastHelper` requires specifying the shape of the lists they manipulate. For example, we can prohibit cyclic lists – which might cause nontermination – by specifying that each heap location in the list is separate from the others. This can be achieved by using *implicit dynamic frames* (IDF) [Smans et al. 2009] extended with *recursive abstract predicates* [Parkinson and Bierman 2005]. IDF extends specifications with *accessibility predicates* (**acc**(`x.f`)) each denoting permission to access a heap location (`o.f` where `x` maps to the object `o` with field `f`).

It also employs the *separating conjunction* ∗ that forces accessibility predicates to refer to different heap locations. With these tools, IDF allows us to reason freely about the heap locations that we can be explicit about. For locations that we cannot or do not want to be explicit about (because they are statically unknown or because listing them would break data abstraction), we can use abstract predicates. *Abstract predicates* are boolean functions defined by a name, arguments, a definition (body), and a scope; and *recursive abstract predicates* are abstract predicates whose bodies are defined recursively. Then, the specifications we are looking for are:

```
predicate valid(List l) = acc(l.head) ∗ ListSeg(l.head,null) and
predicate ListSeg(Node from, Node to) = if (from == to) then true
            else acc(from.val) ∗ acc(from.next) ∗ ListSeg(from.next,to)
```

The *predicate instance* `valid(l)` can provide all the accessibility predicates for list `l` separated by the separating conjunction. `valid(l)`'s body directly provides **acc**(`l.head`) and relies on the `ListSeg(l.head,null)` predicate instance to provide the rest recursively until the list terminates.

To prohibit cyclic lists, we would like to only specify the preconditions of `insertLast` and `insertLastHelper` as `valid(this)`. Unfortunately, static verification tools require additional external (pre- and postconditions) and internal (loop invariants, fold and unfold statements, and lemmas) specifications to verify these methods. The full set of specifications are given in Figure 1, highlighted in grey (and inspired by Smans et al. [2009]). Note that the specifications of `insertLast` and `insertLastHelper` are minimal and yet the ratio of lines of specification code (LoSC) to lines of program code (LoPC) is 48:19 (253%). A functionally complete specification would ensure that after the method executes, the list has the same elements in the same order except that the new item is appended at the end. Of course, verifying this would require even more annotations.

Tools implementing IDF enforce that only one accessibility predicate is held across a call stack for each heap location dereferenced in the call stack. They also require heap locations dereferenced in source code to have corresponding accessibility predicates. Therefore during verification at method boundaries, accessibility predicates required by callees' preconditions are passed to the callee. Since callers may want to access corresponding heap locations after calls, callees' postconditions must give back received accessibility predicates. This forces `insertLast` and `insertLastHelper`'s postconditions to contain `valid(this)`. Additionally, both internal and external specifications must contain accessibility predicates for each heap location used within them, and such specifications are called *self-framed* formulas. For example, **acc**(`x.f`) ∗ `x.f = 2` is self-framed while `x.f = 2` is not.

Similarly to recursive typing, static verification tools must reason about predicate instances either equi-recursively or iso-recursively. The intuitive equi-recursive semantics treats predicate instances as their complete unrollings. Unfortunately, unrolling recursive predicate instances like `ListSeg(l,null)` completely often requires statically-unknown information, e.g. when the list `l` terminates. As a result, predicate instances are treated iso-recursively, i.e. as permissions to access their bodies. Since predicate instances are not equivalent to their bodies, `unfold` and `fold` statements have to be used to explicitly specify when a proof requires the predicate instance and when it requires the body. Figure 1 contains many examples of fold and unfold statements for `valid(this)` and `ListSeg` predicate instances illustrating the approach and its unwieldiness (VeriFast [Jacobs et al. 2011] suffers from this approach). Furthermore, an iso-recursive semantics forces tools to implement the `(unfolding [predicate instance] in [formula])` construct to allow predicate bodies to frame formulas. For example, `insertLastHelper`'s precondition `valid(this) ∗ unfolding valid(this) in this.head != null` uses the construct to allow `valid(this)` to frame `this.head != null`; conversely, `valid(this) ∗ this.head != null` is not self-framed with an iso-recursive semantics.

Fig. 1. The static specifications for Example 2.1.

```
class Node { int val; Node next; }

class List {
  Node head;

  predicate valid(List l) =
    acc(l.head) * ListSeg(l.head,null)

  predicate ListSeg(Node from, Node to) =
    if (from == to) then true else
      acc(from.val) * acc(from.next) *
        ListSeg(from.next,to)

  void insertLast(int val)
    requires valid(this)
    ensures valid(this)
  {
    unfold valid(this);
    if (this.head == null) {
      this.head = new Node(val,null);
      fold ListSeg(this.head.next,null);
      fold ListSeg(this.head,null);
      fold valid(this);
    } else {
      fold valid(this);
      insertLastHelper(val);
    }
  }

  void insertLastHelper(int val)
    requires valid(this) *
      unfolding valid(this) in
        this.head != null
    ensures valid(this)
  {
    unfold valid(this);
    Node y = this.head;
    fold ListSeg(this.head,y);
    unfold ListSeg(y,null);
    while (y.next != null)
      invariant y != null * acc(this.head) *
        ListSeg(this.head,y) *
        acc(y.val) * acc(y.next) *
        ListSeg(y.next,null);
    {
      Node x = y;
      y = y.next;
      fold ListSeg(x.next,y);
      fold ListSeg(x,y);
      unfold ListSeg(y,null);
      appendLemma(this.head, x, y);
    }

    y.next = new Node(val,null);
    fold ListSeg(y.next.next,null);
    fold ListSeg(y.next,null);
    fold ListSeg(y,null);
    appendLemma(this.head, y, null);
    fold valid(this);
  }

  void appendLemma(Node a, Node b, Node c)
    requires ListSeg(a,b) * ListSeg(b,c) *
      (if (c == null) then true
        else acc(c.next))
    ensures ListSeg(a,c) *
      (if (c == null) then true
        else acc(c.next))
  {
    if (a == b) {
    } else {
      unfold ListSeg(a,b);
      appendLemma(a.next, b, c);
      fold ListSeg(a,c);
    }
  }
}
```

To reason about loops tools must also rely on loop invariants, which are hard to develop. `insertLastHelper`'s loop invariant is quite complex, because it must provide accessibility predicates for each heap location accessed in the loop body, be self-framed, be preserved by the loop body, and provide enough information to prove that `valid(this)` is held at the end of `insertLastHelper`. The resulting loop invariant is `y != null * acc(this.head) *`

`ListSeg(this.head,y)` ∗ **acc**(`y.val`) ∗ **acc**(`y.next`) ∗ `ListSeg(y.next,null)`, which segments the accessibility predicates of a list into four parts using `ListSeg` predicate instances. `valid(this)` can be achieved from these segments if tools prove transitivity of `ListSeg`, i.e. `ListSeg(a,b)` ∗ `ListSeg(b,c)` ⇒ `ListSeg(a,c)`. Tools cannot prove inductive proofs like this themselves, so the proof is encoded in an `appendLemma` method.

3 GRADUAL VERIFICATION OF RECURSIVE HEAP DATA STRUCTURES IN ACTION

Section 2 demonstrates just how burdensome specifying recursive heap data structures for static verification can be. Gradual verification [Bader et al. 2018] is a promising approach that, if extended, could allow developers to deal with this burden incrementally. Of course, developers could also choose not to push all the way to a fully statically verified system.

Gradual verification applies Garcia et al. [2016]'s *Abstracting Gradual Typing* framework to a simple static verification system extended with *gradual formulas*. A gradual formula is either a static formula (a *precise formula*) or a static formula joined with ? (an *imprecise formula*), similarly to gradual refinement types [Lehmann and Tanter 2017]. This produces a static verification system that supports partially specified code via imprecise formulas. The key to this system is the concretization of gradual formulas to the set of static formulas that they represent. A static formula represents a set containing only itself and an imprecise formula represents a potentially infinite set of static formulas that are satisfiable and imply its static part. Then static verification operators, such as formula implication, are extended with this interpretation in mind ($\widetilde{\Rightarrow}$). For example, $x \geq 0 \wedge ? \widetilde{\Rightarrow} x \geq 50$ is true, because $x \geq 50$ is among the formulas represented by $x \geq 0 \wedge ?$, and obviously implies $x \geq 50$. On the other hand, $x \geq 0 \wedge ? \widetilde{\not\Rightarrow} x < 0$, because no formula represented by $x \geq 0 \wedge ?$ can imply $x < 0$, i.e. the static part is respected. This leads to a static verification system that can reject clearly invalid programs, and will optimistically accept interpretations of gradual formulas that may not be true at runtime. Soundness is preserved by dynamically verifying such interpretations ($x \geq 50$) at appropriate places, similarly to how casts enforce static assumptions in gradual typing. Ideally, gradual verification systems should also adhere to gradual properties similar to those for gradual typing [Siek et al. 2015]. The rest of this section uses examples to illustrate how gradual verification could reduce the specification burden of Example 2.1.

3.1 A simple attempt at static verification

The specifications provided in Example 3.1 below (highlighted in grey and yellow) heavily rely on imprecise formulas to avoid specifying the separation of heap locations in a list, as this can be difficult for developers who are new to IDF and recursive abstract predicates. In particular, the `valid` predicate is defined as the unknown formula ?. With this, as long as the `valid(this)` predicate instances are unfolded and folded on entry and exit to the methods and the loop invariant also relies on ?, `ListSeg` becomes unnecessary for verifying heap accesses. ? optimistically justifies any missing accessibility predicates statically, which are then verified dynamically. As a result, the lines of specification code needed to specify and verify `insertLast` and `insertLastHelper` has been significantly reduced compared to Figure 1 (from 48:19 (LoSC:LoPC), 253% to 13:19, 68%).

Example 3.1.

```
class Node { int val; Node next; }

class List {
  Node head;

  predicate valid(List l) = ?

  void insertLast(int val)
    requires valid(this)
    ensures valid(this)
  {
    unfold valid(this);
    if (this.head == null) {
      this.head = new Node(val,null);
      fold valid(this);
    } else {
      fold valid(this);
      insertLastHelper(val);
    }
  }

  void insertLastHelper(int val)
    requires valid(this) *
      unfolding valid(this) in
        this.head != null
    ensures valid(this)
  {
    unfold valid(this);
    Node y = this.head;
    while (y.next != null)
      invariant ? * y != null
      { y = y.next; }
    y.next = new Node(val,null);
    fold valid(this);
  }
}
```

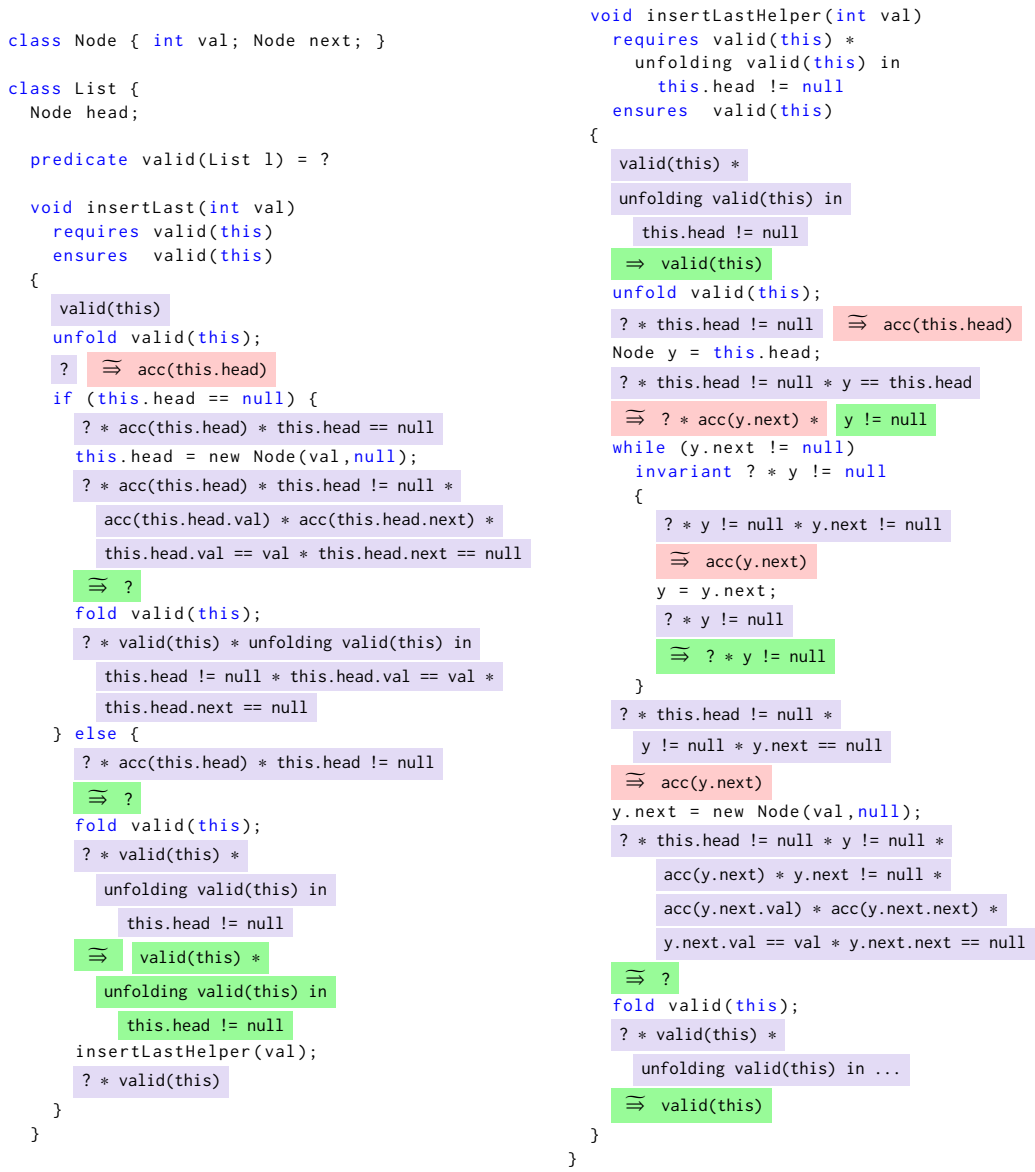
Figure 2 illustrates how gradual verification should operate to support this methodology. It contains conditions from applying a strongest postcondition rule to each statement, highlighted in purple. The strongest postcondition calculus essentially applies the rules to the static part of a gradual formula and carries through or appends ? to the resulting static formula when appropriate. However, special considerations are made for fold statements when the predicate instance being folded has an imprecise body, for method call statements when the corresponding method's precondition is imprecise, and for loops when their loop invariants are imprecise. Figure 2 also contains checks with verified information, highlighted in green and red. Green checks are verified statically and red checks are verified dynamically. Checks are generated to ensure heap accesses are valid, loop invariants are in fact loop invariants, predicate bodies or instances are available before corresponding fold or unfold statements, and method preconditions are available before corresponding calls. If static information is available to satisfy a check, then it is statically verified. If ? can provide missing information to satisfy a check that has not been statically verified, then it is dynamically verified. Otherwise, verification of the check fails. Notably, there are a significant number of red checks involving accessibility predicates. This is expected, as the developer has chosen not to specify that information.

Perhaps a developer would rather not specify fold and unfold statements in Example 3.1. Changing `insertLast` and `insertLastHelper`'s preconditions to ? or appending ? to their preconditions allows ? to justify the missing information needed for verification without folding and unfolding `valid(this)`. In fact, fold and unfold statements can be omitted as long as their omission does not cause static verification to fail. As illustrated, achieving this condition is easier with more imprecision.

3.2 An involved attempt at static verification

The specifications in Example 3.2 represent a reasonable attempt by a developer to specify and verify the separation of heap locations in lists for `insertLast` and `insertLastHelper`. The developer specifies `valid`'s body as **acc**(`l.head`) $*$ `ListAcc(l.head)` where `ListAcc`'s declaration is

Fig. 2. The gradual verification of Example 3.1.




```
predicate ListAcc(Node root) = if (root == null) then true else
                    acc(root.val) * acc(root.next) * ListAcc(root.next)
```

The developer uses the `valid` predicate to specify the methods, adding pre- and postconditions, `unfold` and `fold` statements for `valid` and `ListAcc`, and a loop invariant. But alas, she is not sure how to utilize her loop invariant `y != null * acc(y.val) * acc(y.next) * ListAcc(y.next)` and the program statements after the while loop to prove `valid(this)`. Instead of trying to figure this out, the developer lets the postcondition of `insertLastHelper` be `?` in the meantime. Ultimately,

Example 3.2's specifications are nice, because the developer can reason in terms of a list predicate rather than list segments and does not need a lemma. This reduces the lines of specification code needed to specify and verify `insertLast` and `insertLastHelper` compared to Figure 1 (from 48:19 (LoSC:LoPC), 253% to 25:19, 132%).

Example 3.2.

```
class Node { int val; Node next; }

class List {
  Node head;

  predicate valid(List l) =
    acc(l.head) * ListAcc(l.head)

  predicate ListAcc(Node root) =
    if (root == null) then true else
      acc(root.val) * acc(root.next)
        * ListAcc(root.next)

  void insertLast(int val)
    requires valid(this)
    ensures valid(this)
  {
    unfold valid(this);
    if (this.head == null) {
      this.head = new Node(val,null);
      fold ListAcc(this.head.next);
      fold ListAcc(this.head);
      fold valid(this);
    } else {
      fold valid(this);
      insertLastHelper(val);
    }

  }

  void insertLastHelper(int val)
    requires valid(this) *
      unfolding valid(this) in
        this.head != null
    ensures ?
  {
    unfold valid(this);
    Node y = this.head;
    unfold ListAcc(y);
    while (y.next != null)
      invariant y != null * acc(y.val) *
        acc(y.next) * ListAcc(y.next)
    {
      y = y.next;
      unfold ListAcc(y);
    }
    y.next = new Node(val,null);
    fold ListAcc(y.next.next);
    fold ListAcc(y.next);
    fold ListAcc(y);
  }
}
```

Figure 3 contains the gradual verification of Example 3.2. Static verification is now used everywhere except to prove that `valid(this)` holds after `insertLast` is called. Dynamic verification is used instead, because the postcondition of `insertLastHelper` was weakened (i.e. made less precise) from `valid(this)` to ? to allow verification without significant annotation burden.

4 CHALLENGES

This section discusses challenges with developing a gradual verification system from a static verification system supporting IDF and recursive abstract predicates. This section also presents proposed solutions to those challenges.

4.1 Static verification

All gradual verification systems must adhere to the static gradual guarantee [Bader et al. 2018; Siek et al. 2015], which states that reducing the precision of specifications never breaks static

Fig. 3. The gradual verification of Example 3.2.

```
class Node { int val; Node next; }

class List {
  Node head;

  predicate valid(List l) =
    acc(l.head) * ListAcc(l.head)

  predicate ListAcc(Node root) =
    if (root == null) then true else
      acc(root.val) * acc(root.next)
        * ListAcc(root.next)

  void insertLast(int val)
    requires valid(this)
    ensures  valid(this)
  {
    valid(this)  ⇒ valid(this)
    unfold valid(this);
    acc(this.head) * ListAcc(this.head)
    ⇒ acc(this.head)
    if (this.head == null) {
      acc(this.head) * ListAcc(this.head) *
        this.head == null
      this.head = new Node(val,null);
      acc(this.head) * this.head != null *
        acc(this.head.val) * acc(this.head.next) *
        this.head.next == null
      ⇒ if (this.head.next == null) then true
        else ...
      fold ListAcc(this.head.next);
      acc(this.head) * this.head != null *
        acc(this.head.val) * acc(this.head.next)
        * List(this.head.next)
      ⇒ acc(this.head.val) * acc(this.head.next)
        * List(this.head.next)
      fold ListAcc(this.head);
      acc(this.head) * List(this.head)
      ⇒ acc(this.head) * List(this.head)
      fold valid(this);
      valid(this)  ⇒ valid(this)
    } else {
      acc(this.head) * ListAcc(this.head) *
        this.head != null
      ⇒ acc(this.head) * ListAcc(this.head)
      fold valid(this);
      valid(this) *
        unfolding valid(this) in
          this.head != null
      ⇒ valid(this) *
        unfolding valid(this) in
          this.head != null
      insertLastHelper(val);
      ?  ⇏ valid(this)
    }
  }
```

verification. In Figures 2 and 3 (Secs. 3.1 and 3.2) static verification will fail if ? doesn't provide the missing information highlighted in red. Similarly, static verification will fail if specifications are not self-framed. In Figure 2, both user and tool generated specifications are not self-framed: ? * `y != null * y.next != null` and `valid(this) * unfolding valid(this) in this.head != null`. Note, `valid(this) * unfolding valid(this) in this.head != null` would be self-framed if the body of `valid(this)` contained **acc**(this.head), but it does not. Therefore, ? must provide missing accessibility predicates and predicate instances for verification and framing.

Solution. First, formulas, such as `valid(this) * unfolding valid(this) in this.head != null` where `valid(this)`'s body is imprecise, can be thought of as ? * `valid(this) * this.head != null`. Then, Bader et al. [2018]'s concretization definition for gradual formulas can be extended to allow ? to represent missing accessibility predicates and predicate instances. The new definition should rely on an iso-recursive interpretation of predicate instances and require that all precise formulas represented by gradual formulas be self-framed. Since equi-recursive styled verification operators and predicates require statically unknown information in implementation, an iso-recursive interpretation of predicate instances is chosen.

Fig. 3. The gradual verification of Example 3.2 (continued).

```
void insertLastHelper(int val)
  requires valid(this) *
    unfolding valid(this) in
      this.head != null
  ensures  ?
{
  valid(this) *
  unfolding valid(this) in this.head != null
  ⇒ valid(this)
  unfold valid(this);
  acc(this.head) * ListAcc(this.head) *
    this.head != null
  ⇒ acc(this.head)
  Node y = this.head;
  acc(this.head) * ListAcc(this.head) *
    this.head != null * y == this.head
  ⇒ ListAcc(y)
  unfold ListAcc(y);
  acc(this.head) * this.head != null *
    y == this.head * acc(y.val) *
    acc(y.next) * ListAcc(y.next)
  ⇒ acc(y.next)
  ⇒ loop invariant
  while (y.next != null)
    invariant y != null * acc(y.val) *
      acc(y.next) * ListAcc(y.next)
  {
    y != null * acc(y.val) * acc(y.next) *
      ListAcc(y.next) * y.next != null
    ⇒ acc(y.next)
    y = y.next;
    ListAcc(y) * y != null
    ⇒ ListAcc(y)
    unfold ListAcc(y);
    acc(y.val) * acc(y.next) *
      ListAcc(y.next) * y != null
    ⇒ y != null * acc(y.val) *
      acc(y.next) * ListAcc(y.next)
  }
  acc(this.head) * this.head != null *
    acc(y.val) * acc(y.next) * ListAcc(y.next) *
    y != null * y.next == null
  ⇒ acc(y.next)
  y.next = new Node(val,null);
  y != null * acc(y.val) * acc(y.next) *
    y.next != null * acc(y.next.val) *
    acc(y.next.next) * y.next.next == null
  ⇒ true
  fold ListAcc(y.next.next);
  y != null * acc(y.val) * acc(y.next) *
    y.next != null * acc(y.next.val) *
    acc(y.next.next) * ListAcc(y.next.next)
  ⇒ acc(y.next.val) * acc(y.next.next) *
    List(y.next.next)
  fold ListAcc(y.next);
  y != null * acc(y.val) * acc(y.next) *
    List(y.next)
  ⇒ acc(y.val) * acc(y.next) * ListAcc(y.next)
  fold ListAcc(y);
  y != null * ListAcc(y)
  ≅ ?
}
```

4.2 Dynamic verification

For gradual verification systems to be sound, any accessibility predicates or predicate instances supplied by ? during static verification must be dynamically verified. Furthermore, any runtime system extended with dynamic verification must adhere to the dynamic gradual guarantee, which states that reducing the precision of specifications does not change the observable behavior of the runtime system.

Solution. To facilitate verification of accessibility predicates, runtime systems can track and update a set of heap locations at every program point that indicate accessibility. Heap locations are added to this set when objects are created and removed from this set and passed to callees if they are required by preconditions at method boundaries. Preconditions that are imprecise or hide imprecision in predicate instances require all the heap locations of the caller's set, because ? may represent any accessibility predicate not already given statically which refers to one of those heap locations. This may not be ideal, so an extra specification construct (we call this the "hold" construct) can be employed before method calls to make explicit, using corresponding accessibility predicates, what heap locations should be kept by callers. When callees finish executing, accumulated heap locations are passed back to the caller. Then, if an accessibility predicate refers to a heap location in this set at a program point it is verified at that program point. Predicate instances are verified equi-recursively. During this process nested ?s may be ignored, if missing framing information is accounted for and verified.

An equi-recursive evaluation of predicate instances is the natural choice for runtime systems, since equi-recursive predicates mirror the structure of the program. In fact, an iso-recursive styled runtime system would break the dynamic guarantee when heuristics to infer fold and unfold statements fail. However, an equi-recursive styled dynamic verification system incurs runtime overhead. For example, runtime verification is exponential for certain formulas containing logical disjunction $\vee$ and accessibility predicates and for predicate instances whose definitions contain duplicate recursive predicate instances joined with $\wedge$. Worse even, is if predicate instances do not terminate.

Solution. In any of these cases, we suggest not producing such formulas for verification. In particular, disjunction can be replaced by conditionals containing only boolean expressions in their conditions. We are still working on termination detection measures in the face of imprecision. Beyond the aforementioned challenges, we have yet to explore runtime performance for our extended gradual verification system. This is important future work.

5 RELATED WORK

We have already compared our work to the most-closely related research, including work on the underlying logics [Parkinson and Bierman 2005; Reynolds 2002; Smans et al. 2009] and the theory of gradual typing and gradual verification [Bader et al. 2018; Garcia et al. 2016; Lehmann and Tanter 2017; Siek and Taha 2007, 2006; Siek et al. 2015]. Therefore, we will focus on providing a more detailed comparison of our approach to its predecessor and to other related work.

Bader et al. [2018]'s work on gradual verification outlines an IDF extension in an accompanying technical report. We take inspiration from this work for our own work and formal system, but every piece of the formal system in the technical report is extended or redesigned in non-trivial ways to support recursive predicates and programs with loops and conditionals. New definitions are also introduced. The decisions to support both an iso-recursive styled static verification system and an equi-recursive styled dynamic verification system, imprecise predicate bodies, and imprecision regarding fold and unfold statements motivate these extensions.

Other related work includes gradual type systems that include notions of ownership or linearity, as IDF can be viewed in these ways. In gradual typestate [Garcia et al. 2014; Wolff et al. 2011] permissions to objects are passed linearly from one function to another without duplication. Strong permissions that are lost due to imprecise specifications can be regained through a runtime check, so long as a conflicting permission does not exist. Sergey and Clarke [2012]'s gradual ownership approach allows developers to specify containment relationships between objects signifying ownership. An *owned* object cannot be accessed from outside its owner. These relationships are checked statically if possible and dynamically otherwise. Neither of these efforts benefited from the AGT

framework [Garcia et al. 2016], which led to principled design choices in our work. Also, it is unclear whether the gradual guarantees of Siek et al. [2015] hold in these proposals.

Additionally, Nguyen et al. [2008] leveraged static information to reduce the overhead of their runtime checking approach for separation logic. They do not try to report static verification failures, because their technique cannot not distinguish between failures due to inconsistent specifications and failures due to incomplete specifications. Also, their runtime checking approach forces developers to specify matching heap footprints in pre- and postconditions to avoid false negatives.

There is also related work focused on making static verification more usable. In particular, Furia and Meyer [2010] infer candidate loop invariants by using heuristics to weaken postconditions into invariants. Therefore, their approach cannot infer invariants not expressible as weakenings of postconditions. Gradual verification does not try to infer invariants directly, but rather infers missing information required for verification and not supplied by invariants in a principled way. Additionally, developers can use Dafny's [Leino 2010] **assume** and **assert** statements to debug specifications similar to how they debug programs with print statements [Lucio 2017]. Unlike gradual verification, this approach does not reduce specification burden and requires manual elicitation and verification of missing specifications needed for verification. Further, a significant number of tools (Smallfoot [Berdine et al. 2005], jStar [Distefano and Parkinson J 2008], Chalice [Leino et al. 2009]) rely on heuristics to infer fold and unfold statements for verification. Incorporating heuristics into gradual verification is likely complicated by imprecise specifications. However, it is a promising direction for future work.

6 CONCLUSION

Gradual verification provides a promising solution to the static verification burden imposed by tools supporting recursive heap data structures. However, unique challenges must be overcome for gradual verification to support IDF and recursive predicates: 1) ? in imprecise specifications must represent accessibility predicates and predicate instances needed for static verification, 2) accessibility predicates and predicate instances must be dynamically verified, and 3) dynamically verifying specifications with accessibility predicates and predicate instances can incur significant runtime overhead. Possible solutions to these challenges are: 1) adjusting Bader et al. [2018]'s concretization definition for gradual formulas to rely on an *iso-recursive* interpretation of predicate instances and require that concretizations of gradual formulas be *self-framed*, 2) verifying accessibility predicates by tracking and updating a set of heap locations at runtime and verifying predicate instances *equi-recursively*, and 3) using language design to avoid particular specifications that incur runtime overhead and exploring further optimizations in future work.

We have formalized a gradual verification system supporting IDF and recursive abstract predicates based on this work. It is sound and we are confident it adheres to the gradual guarantees (proofs of the gradual guarantees are still in progress). Going forward, a prototype will be developed and used as a means to evaluate the formal system and explore practical implementation challenges, such as runtime performance. It will also be used to explore gradual verification's impact on static verification education.

ACKNOWLEDGMENTS

This material is based upon work supported by a Facebook Testing and Verification research award and the National Science Foundation under Grant No. CCF-1901033 and Grant No. DGE1745016. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation or Facebook.

REFERENCES

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.

Carlo Alberto Furia and Bertrand Meyer. 2010. Inferring loop invariants using postconditions. In *Fields of logic and computation*. Springer, 277–300.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. 36, 4, Article 12 (Oct. 2014), 12:1–12:44 pages.

Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*. Springer, 41–55.

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. Paris, France, 775–788.

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370.

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.

Paqui Lucio. 2017. A Tutorial on Using Dafny to Construct Verified Software. *arXiv preprint arXiv:1701.04481* (2017).

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

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.

Ilya Sergey and Dave Clarke. 2012. Gradual Ownership Types. In *Proceedings of the 21st European Conference on Programming Languages and Systems (ESOP'12)*. Springer-Verlag, Berlin, Heidelberg, 579–599. https://doi.org/10.1007/978-3-642-28869-2_29

Jeremy Siek and Walid Taha. 2007. Gradual typing for objects. In *European Conference on Object-Oriented Programming*. Springer, 2–27.

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.

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. 2011. Gradual typestate. In *European Conference on Object-Oriented Programming*. Springer, 459–483.