Gradual Verification of Recursive Heap Data Structures

JENNA WISE, Carnegie Mellon University, USA
JOHANNES BADER, Jane Street, USA
CAMERON WONG, Jane Street, USA
JONATHAN ALDRICH, Carnegie Mellon University, USA
ÉRIC TANTER, Computer Science Department (DCC), University of Chile, Chile
JOSHUA SUNSHINE, Carnegie Mellon University, USA

Current static verification techniques do not provide good support for incrementality, making it difficult for developers to focus on specifying and verifying the properties and components that are most important. Dynamic verification approaches support incrementality, but cannot provide static guarantees. To bridge this gap, prior work proposed gradual verification, which supports incrementality by allowing every assertion to be complete, partial, or omitted, and provides sound verification that smoothly scales from dynamic to static checking. The prior approach to gradual verification, however, was limited to programs without recursive data structures. This paper extends gradual verification to programs that manipulate recursive, mutable data structures on the heap. We address several technical challenges, such as semantically connecting iso- and equi-recursive interpretations of abstract predicates, and supporting gradual verification of heap ownership. This work thus lays the foundation for future tools that work on realistic programs and support verification within an engineering process in which cost-benefit trade-offs can be made.

228

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

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

ACM Reference Format:
Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual Verification of Recursive Heap Data Structures. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 228 (November 2020), 39 pages. https://doi.org/10.1145/3428296

1 INTRODUCTION

Hoare proposed a logic for static verification where developers specify method pre- and postconditions [Hoare 1969]. Over time, this work has been extended to support more interesting programs.

*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. 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, Facebook, ANID, or the Millennium Science Initiative.

Authors' addresses: Jenna Wise, Carnegie Mellon University, USA, jlwise@andrew.cmu.edu; Johannes Bader, Jane Street, USA, johannes-bader@hotmail.de; Cameron Wong, Jane Street, USA, cam@camdar.io; Jonathan Aldrich, Carnegie Mellon University, USA, jonathan.aldrich@cs.cmu.edu; Éric Tanter, Computer Science Department (DCC), University of Chile, Chile, etanter@dcc.uchile.cl; Joshua Sunshine, Carnegie Mellon University, USA, sunshine@cs.cmu.edu.

2475-1421/2020/11-ART228
https://doi.org/10.1145/3428296

Most notably, Reynolds [2002] introduced *separation logic* to support modular verification of programs that manipulate heap data structures. As an extension to separation logic, Parkinson and Bierman [2005] proposed *recursive abstract predicates*, enabling the verification of recursive heap data structures such as graphs, trees, or linked-lists. Shortly after, *implicit dynamic frames* (IDF) was proposed by Smans et al. [2009] as an alternative to separation logic that allows developers to specify heap ownership separately from heap contents.

Unfortunately, these techniques require developers to provide enough specifications to form a complete inductive proof. Consequently, even in very simple programs, a specification that is inductively verifiable may be twice the length of merely specifying the properties the programmer cares about (§3.2). To address this issue, Bader et al. [2018] proposed *gradual verification*, which builds on prior research on gradual typing [Siek and Taha 2007, 2006; Siek et al. 2015], in particular the Abstracting Gradual Typing methodology [Garcia et al. 2016]. Bader et al. [2018] extend a simple Hoare logic static verifier with partial, *imprecise specifications*. Statically, the gradual verifier can optimistically assume any (non-contradictory) strengthening of an imprecise specification. To ensure soundness, dynamic checks are added when partial specifications are optimistically strengthened. Bader *et al.*'s approach smoothly supports the spectrum between static and dynamic verification, as formalized similarly to the refined criteria for gradual typing [Siek et al. 2015].

While promising, the prior work on gradual verification does not support the specification of recursive heap data structures, and thus cannot verify realistic programs. In this paper, we address this limitation by presenting the design, formalization, and meta-theory of a sound gradual verifier for programs that manipulate recursive heap data structures. Our approach follows Bader *et al.*'s methodology, but starts from a static verifier with IDF and recursive abstract predicates. This more sophisticated setting requires us to address the following technical challenges:

- Imprecise specifications may be strengthened not just with boolean assertions about arithmetic expressions, but also with both *abstract predicates* and *accessibility predicates*, which denote ownership of heap locations. Our strengthening definition also includes *self-framing*, a well-formedness condition required by IDF [Smans et al. 2009].
- Both accessibility predicates and abstract predicates must potentially be verified dynamically. Our system verifies accessibility predicates at runtime by tracking and updating a set of owned heap locations. We verify recursive abstract predicates by executing them as recursive boolean functions. This runtime semantics corresponds to an equi-recursive interpretation of abstract predicates, contrasting with the iso-recursive interpretation used in static verifiers [Summers and Drossopoulou 2013]; our theory ensures that these interpretations are consistent.

We show that the resulting gradual verifier is sound, that it is a *conservative extension* of the static verifier—meaning that both coincide on programs with fully-precise specifications—and that it adheres to the *gradual guarantee*. This guarantee, originally formulated for gradual type systems [Siek et al. 2015], captures the intuition that relaxing specifications should not introduce new (static or dynamic) verification errors.

The rest of this paper is outlined as follows. The annotation burden induced by statically verifying linked list insertion is discussed in §2. Section 3 illustrates how this burden can be reduced or eliminated with gradual verification by using examples, and §4 discusses challenges and solutions to supporting such examples. In §5 we formally present a statically verified language supporting a propositional specification logic extended with IDF and recursive heap data structures, before gradualizing the static semantics of this language in §6 and dynamic semantics in §7. §8 discusses the properties of the resulting gradual verifier. Finally, §9 and §10 further relate this paper to prior work and discuss future work, respectively. The appendix of this paper contains full gradual verification examples and supplementary definitions (*e.g.* complete semantics of both the static

```
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);
  }
}
```

Fig. 1. Linked list with insertion

and gradual verifier introduced in this work). Proofs of all propositions and lemmas are given in Appendix §A.4.

2 THE BURDEN OF STATIC VERIFICATION

With static verification tools, ensuring that a component satisfies a given property requires more than specifying the said property: many additional specifications are needed for tools to be able to discharge proof obligations statically. In this section, we show that this additional specification burden can be significant, even for a very simple example.

The program in Figure 1 implements a linked list and two methods for inserting an element at the end of a list. Notice that `insertLastHelper` iteratively traverses a list for insertion, and that both methods diverge if given cyclic lists. Therefore, it is useful to ensure these methods only receive (and produce) acyclic lists. Let us look at how to achieve this with a static verifier.

One way to specify that a list is acyclic is to use the following *abstract predicates* [Parkinson and Bierman 2005], which are essentially pure boolean functions:

```
predicate acyclic(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)
```

Notice that `listSeg` is a *recursive* abstract predicate. Additionally, `acyclic` and `listSeg`'s bodies rely on *accessibility predicates* of the form **acc**(`x.f`) and on the *separating conjunction* $*$, from implicit dynamic frames (IDF) [Smans et al. 2009]. A program can only access a particular heap location if the corresponding accessibility predicate is provided. For example, **acc**(`l.head`) gives permission to access the heap location `o.head` if `l` is bound to the object `o`. The separating conjunction forces accessibility predicates to refer to different heap locations. `listSeg` recursively generates accessibility predicates for every node in a list segment. The accessibility predicates are joined with the separating conjunction. Therefore, the recursive *predicate instance* `acyclic(l)` states that all the heap locations in list `l` are distinct, *i.e.* `l` is acyclic.

A developer can expect to simply specify that both `insertLast` and `insertLastHelper` have `acyclic(this)` as pre- and postconditions. However, they may be disappointed by the many additional specifications required to statically discharge the proof obligations these specifications introduce: loop invariants, fold and unfold statements, and lemmas, as shown in Figure 2, and inspired by Smans et al. [2009]. Although this example is very simple, there is far more specification code (44 lines) than program code (19 lines). Furthermore, this 44:19 ratio only highlights part of the problem: many specifications are far more complex than the program itself, as explained next.

The specification `unfold acyclic(this)` at line 18 expands the abstract predicate `acyclic(this)` into its body. This unfolding exposes the accessibility predicate **acc**(`this.head`), which gives permission to access the heap location of `this.head`. Dually, `fold acyclic(this)` repacks `acyclic(this)`'s body. Figure 2 explicitly uses unfold and fold statements to control the availability of predicate

```
1  class Node { int val; Node next; }
2
3  class List {
4    Node head;
5
6    predicate acyclic(List l) =
7      acc(l.head) * listSeg(l.head,null)
8
9    predicate listSeg(Node from, Node to) =
10     if (from == to) then true else
11       acc(from.val) * acc(from.next) *
12         listSeg(from.next,to)
13
14   void insertLast(int val)
15     requires acyclic(this)
16     ensures acyclic(this)
17   {
18     unfold acyclic(this);
19     if (this.head == null) {
20       this.head = new Node(val,null);
21       fold listSeg(this.head.next,null);
22       fold listSeg(this.head,null);
23       fold acyclic(this);
24     } else {
25       fold acyclic(this);
26       insertLastHelper(val);
27     }
28   }
29
30   void insertLastHelper(int val)
31     requires acyclic(this) *
32       unfolding acyclic(this) in
33         this.head != null
34     ensures acyclic(this)
35   {
36     unfold acyclic(this);
37     Node y = this.head;
38     fold listSeg(this.head,y);
39     unfold listSeg(y,null);
40     while (y.next != null)
41       invariant y != null * acc(this.head) *
42         listSeg(this.head,y) *
43         acc(y.val) * acc(y.next) *
44         listSeg(y.next,null);
45     {
46       Node x = y;
47       y = y.next;
48       unfold listSeg(y,null);
49       fold listSeg(x.next,y);
50       fold listSeg(x,y);
51       appendLemma(this.head, x, y);
52     }
53
54     y.next = new Node(val,null);
55     fold listSeg(y.next.next,null);
56     fold listSeg(y.next,null);
57     fold listSeg(y,null);
58     appendLemma(this.head, y, null);
59     fold acyclic(this);
60   }
61
62   void appendLemma(Node a, Node b, Node c)
63     requires listSeg(a,b) * listSeg(b,c)
64     ensures listSeg(a,c)
65   {
66     if (a == b) {
67     } else {
68       unfold listSeg(a,b);
69       appendLemma(a.next, b, c);
70       fold listSeg(a,c);
71     }
72   }
73 }
```

Static specification ☐ Program code

Fig. 2. Specifying and proving acyclicity for linked list insertion

information. Each predicate instance is an opaque permission to access its body, *i.e.* predicates are *iso-recursive* [Summers and Drossopoulou 2013]. Some *dynamic* verifiers reason about predicate instances *equi-recursively*, *i.e.* treat a predicate instance equal to its complete unfolding. However, completely unfolding recursive predicates often requires statically unknown information, such as the length of the list in our example. Therefore, *static* verifiers reason about predicate instances iso-recursively.

The while loop invariant at lines 41–44 segments a list into three parts using `listSeg`: from the head to the current node (`listSeg(this.head,y)`), the current node (**acc**(`y.val`) ∗ **acc**(`y.next`)), and the rest of the list (`listSeg(y.next,null)`). The loop body accesses `y.next`, so the loop invariant must expose **acc**(`y.next`). After a new node holding the inserted element is added to the list at line 54, we must show that the `acyclic` predicate holds for the new list. The loop invariant also supports this goal. To build up the `acyclic` predicate we must first construct a `listSeg` predicate from the beginning of the list to the new end of the list. We do this by starting with an empty list segment (line 55) and incrementally extending it with the newly added element (line 56) and the previous end of the list (line 57). This gives us a `listSeg` predicate from the current node to the new end of the list. We then append the `listSeg` predicate from the head of the list to the current node (loop invariant) to the `listSeg` predicate from the current node to the new end of the list (line 57). To achieve this, we need to prove that `listSeg` is transitive. Unfortunately, static tools usually cannot automatically discharge such inductive proofs, so we encode the proof in the `appendLemma` method at 58. Note that such additional proof efforts are part of the barriers to the adoption of static verification, which would be important to get rid of. Finally, we combine the accessibility predicate to the head of the list (loop invariant) with our `listSeg` predicate to reconstruct the `acyclic` predicate (lines 7 and 59).

As the above description makes clear, static verification tools can impose a significant specification burden on developers even for simple programs. Constructing loop invariants and (un)folding predicates can be considerably more complex than program code. Simply ensuring that `insertLast` and `insertLastHelper` receive and produce acyclic lists requires far more specification code than program code. Of course, verifying more properties, for example that some insertion preserves ordering, would require substantially more specification and verification effort.

3 GRADUAL VERIFICATION OF RECURSIVE HEAP DATA STRUCTURES IN ACTION

We now demonstrate how developers can use gradual verification to choose which obligations they want to meet statically and leave the rest to be dynamically checked. They can then incrementally address each proof obligation statically until they reach fully static verification, or stop at any point along the way. As a result, the complexity of verification can be managed in small increments. In the rest of this section, we show different partial specifications of list insertion (§3.1-§3.2), as well as list search (§3.3). These examples illustrate the smooth scaling from dynamic to static checking enabled by gradual verification.

3.1 Gradually Verifying List Insertion: Take 1

Figure 3 presents a possible gradual specification of acyclicity of list insertion. In addition to fully precise formulas (in gray), the specification includes *imprecise formulas* [Lehmann and Tanter 2017] (in yellow), which contain the *unknown formula* ? in addition to a static part (`true` if omitted).

Here, the developer chooses to completely ignore accessibility predicates, which would be required for full static verification (§2), and only focuses on a partial specification. First, the `acyclic` predicate is kept unknown by using ? as its body (line 4). Second, only the simple part of the loop invariant—*i.e.* the current node of the list is not null—is statically specified, thanks to the imprecise formula `? * y != null` (line 27). Intuitively, this formula means that only `y != null` is enforced and guaranteed statically, but that other properties can be optimistically assumed. Note that the partial specification explicitly deals with (un)folding the `acyclic` predicate; unfolding `acyclic` implies bringing its imprecision (*i.e.* optimism) in the verification, while folding `acyclic` simply satisfies the declared pre- and postconditions. In general, the only interesting properties that can be verified with this gradual specification are whether `y != null` is preserved by the loop and whether heap accesses are justified with accessibility predicates. We discuss this in more detail.

```
class Node { int val; Node next; }
class List {
  Node head;
  predicate acyclic(List l) = ?
  void insertLast(int val)
    requires acyclic(this)
    ensures acyclic(this)
  {
    unfold acyclic(this);
    if (this.head == null) {
      this.head = new Node(val,null);
      fold acyclic(this);
    } else {
      fold acyclic(this);
      insertLastHelper(val);
    }
  }
```

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

Imprecise specification | Precise specification

Fig. 3. A possible gradual specification of `insertLast` and `insertLastHelper` from Figure 1

```
void insertLastHelper(int val)
  requires acyclic(this) *
    unfolding acyclic(this) in
      this.head != null
  ensures acyclic(this)
{
  acyclic(this) * unfolding acyclic(this) in
    this.head != null ⇛
  ? * acyclic(this)
  ? * acyclic(this)
  unfold acyclic(this);
  ? ⇛ ? * acc(this.head) * this.head != null *
    acc(this.head.next)
  ? * acc(this.head) * this.head != null *
    acc(this.head.next)
  Node y = this.head;
  ? * y != null * acc(y.next)
```

```
  while (y.next != null)
    invariant ? * y != null
    {
    ? * y != null * y.next != null * acc(y.next)
      ⇛ ? * acc(y.next.next)
          * acc(y.next) * y.next != null
      ? * acc(y.next.next) * acc(y.next) *
      y.next != null
      y = y.next;
      ? * y != null * acc(y.next)
    }
  ? * y != null * y.next == null ⇛
  ? * acc(y.next)
  ? * acc(y.next)
  y.next = new Node(val,null);
  ?
  fold acyclic(this);
  acyclic(this)
}
```

Intermediate condition produced by $\widetilde{WLP}$ | Dynamically checked right side of ⇛ | Left side of ⇛ | Statically checked right side of ⇛

Fig. 4. The gradual verification of insertLastHelper from Figure 3

Figure 4 demonstrates how to gradually verify `insertLastHelper` from Figure 3. The formulas shown in method bodies (highlighted in purple) are the result of applying *gradual weakest liberal precondition* rules $\widetilde{WLP}$ (defined in §6.5) to each program statement.

$\widetilde{WLP}$ proceeds from the end of a method body to the beginning, starting with the postcondition on the last line. Then, for each program statement $\widetilde{WLP}$ calculates a new intermediate condition that is minimally sufficient to verify the new program statement and the prior intermediate condition. Since `?` is the body of the `acyclic` abstract predicate, $\widetilde{\text{WLP}}$ calculates that `?` is minimally sufficient for lines 34 and 35. Assigning to `y.next` on line 32 requires an accessibility predicate, so $\widetilde{\text{WLP}}$ joins `acc(y.next)` to `?` on line 31.

When $\widetilde{\text{WLP}}$ cannot soundly propagate a condition backwards, a consistent implication ($\widetilde{\Rightarrow}$) check is performed. These implications are necessary under five conditions: at the beginning of a method, at the beginning of a loop body, at the end of a loop with an imprecise invariant, after unfolding an abstract predicate with an imprecise body, and after a method call with an imprecise postcondition. At line 29 the imprecise loop invariant is joined with the negation of the loop guard. The right-hand side of $\widetilde{\Rightarrow}$ is always the next intermediate condition. Since line 29 is not sufficient to statically entail the intermediate condition on 30, but may optimistically do so considering imprecision, it is *optimistically discharged* and therefore highlighted in red. An optimistically-discharged obligation gives rise to a *dynamic check* when running the program. Note that if the left-hand side of a consistent implication cannot possibly imply the right side (*e.g.* as in `? * x == null` $\widetilde{\Rightarrow}$ `x != null`), then the program is *statically rejected.*

The last condition in a loop body is always the loop invariant joined with accessibility predicates needed to evaluate the loop guard. Line 27 contains the loop invariant and an accessibility predicate for `y.next`. When encountering a variable assignment, like the one on line 26, $\widetilde{\text{WLP}}$ substitutes the right-hand side of the assignment (`y.next`) for the left-hand side (`y`) to generate the intermediate condition above the assignment (lines 24 and 25). In addition, accessibility predicates are added for the right-hand side of the assignment (`acc(y.next)`).

As mentioned earlier, a consistent implication is checked at the beginning of a loop body: the left-hand side (line 21) is the loop invariant, the loop guard, and any accessibility predicates necessary for the guard. The right-hand side, as usual, is the next intermediate condition. Observe that here, some of the conditions to prove are *definitely* implied—via standard implication—by the static part of the left-hand side: they can therefore be *discharged statically*, which is highlighted in green (line 23). The others are optimistically discharged, as before.

The condition on line 17 includes the loop invariant and an accessibility predicate to the loop guard. The condition on lines 14 and 15 follows the same pattern as the assignment discussed earlier. The unfold statement generates the consistent implication on lines 12 and 13. The left side is the body of the unfolded abstract predicate, in this case `?`. Since `?` provides no static information, the entire right-hand side is optimistically discharged.

The condition on line 10 includes the abstract predicate that is unfolded on line 11. This is joined to `?` because the body of `acyclic` is an imprecise formula and $\widetilde{\text{WLP}}$ maintains any residual conditions beyond those needed for the unfolding. Finally, the left-hand side of the $\widetilde{\Rightarrow}$ at the beginning of the method is the method precondition (lines 7 and 8). Since `acyclic(this)` is definitely implied, the right-hand side is fully discharged statically.

The complete gradual verification of Figure 3, including the `insertLast` method, is in §A.1.

3.2 Gradually Verifying List Insertion: Take 2

In Figure 5, we show another, more precise gradual specification of acyclicity for `insertLast` and `insertLastHelper`. The specifications highlighted in gray contain precise formulas, and the ones highlighted in yellow contain imprecise formulas. The darker gray specifications are additional specifications introduced by the developer as an increment over the ones in Figure 3. Here, the developer chooses to fully specify `acyclic`'s body on lines 4 and 5 as **acc**(`l.head`) `* listSeg(l.head, null)`. With these predicates, the developer fully specifies `insertLast` for static verification and adds more complete specifications to `insertLastHelper`. The developer uses `listSeg` to write a loop

```
1  class Node { int val; Node next; }
2  class List {
3    Node head;
4    predicate acyclic(List l) =
5      acc(l.head) * listSeg(l.head, null)
6
7    predicate listSeg(Node from, Node to) =
8      if (from == to) then true else
9        acc(from.val) * acc(from.next)
10         * listSeg(from.next, to)
11
12   void insertLast(int val)
13     requires acyclic(this)
14     ensures acyclic(this)
15   {
16     unfold acyclic(this);
17     if (this.head == null) {
18       this.head = new Node(val,null);
19       fold listSeg(this.head.next, null);
20       fold listSeg(this.head, null);
21       fold acyclic(this);
22     } else {
23       fold acyclic(this);
24       insertLastHelper(val);
25     }
26   }
27   void insertLastHelper(int val)
28     requires acyclic(this) *
29       unfolding acyclic(this) in
30         this.head != null
31     ensures ?
32   {
33     unfold acyclic(this);
34     Node y = this.head;
35     unfold listSeg(y, null);
36     while (y.next != null)
37       invariant y != null * acc(y.val) *
38         acc(y.next) * listSeg(y.next, null)
39     {
40       y = y.next;
41       unfold listSeg(y, null);
42     }
43     y.next = new Node(val,null);
44   }
45 }
```

Imprecise specification | New precise specification (increment over Fig. 3)
Precise specification (from Fig. 3)

Fig. 5. Another possible gradual specification of `insertLast` and `insertLastHelper` from Figure 1

invariant, which exposes **acc**(`y.next`) for statically verifying accesses to `y.next` in the loop body and on line 43. However, the developer does not want to build up specifications to statically prove that the new list after insertion is acyclic. They therefore leave the postcondition of `insertLastHelper` unknown. Observe that, in contrast to Figure 2, the programmer does not need to build up a `listSeg` predicate from the previous end of the list to the new one, state and prove a separate lemma about list concatenation, and state a more complex loop invariant. Instead, the gradual verifier ensures at runtime that the new list after insertion is acyclic. This is a major benefit of gradual verification, which can dispense the verification effort from working around certain limitations of static reasoning tools. The detailed verification of Figure 5 with $\widetilde{\mathrm{WLP}}$ is in §A.1.

3.3 Gradually Verifying List Search

Let us now consider another helpful method for linked lists, `findMax`, which finds and returns the maximal value of the list. The program in Figure 6 contains an iterative implementation of `findMax`. We discuss how a developer uses gradual verification to ensure that `findMax` indeed returns the maximal value of a list; they incrementally build up specifications as illustrated in Figure 7. In doing so, we show how developers can incrementally address proof obligations of interest and explore the cost-benefit tradeoffs between static reasoning effort and runtime overhead.

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

class List {
  Node head;

  int findMax()
  {
    int max = this.head.val;
    Node curr = this.head.next;
    while (curr != null) {
      if (curr.val > max) {
        max = curr.val;
        curr = curr.next;
      } else {
        curr = curr.next;
      }
    }
    result = max;
  }
}
```

Fig. 6. Linked list that iteratively finds and returns its maximal value

The developer begins the first increment (highlighted red, lines 6–20, 23–25, 34) by specifying two properties: whether a value is an upper bound of a list (`upperBound`, lines 6, 7; `upperBoundSeg`, lines 9–12) and whether a value is contained in a list (`contains`, lines 14, 15; `containsSeg`, lines 17–20). The `upperBound` and `contains` predicates are used in `findMax`'s postcondition to ensure that it returns the maximal element (lines 24, 25). The predicates are imprecise to enable heap accesses to `l.head`, `from.val`, and `from.next` without statically-acquired accessibility predicates. The developer specifies that `findMax` not execute on empty lists in its precondition (`this.head != null`). In this first increment, the precondition (line 23) is otherwise imprecise and the loop invariant (line 34) is completely imprecise. As a result, the gradual verifier optimistically assumes—and dynamically checks—accessibility predicates to heap accesses in `findMax`. The invariant also allows the verifier to check `upperBound(this, result)` and `contains(this, result)` at runtime.

To move towards a strengthened version of `findMax`, the developer adds the specifications highlighted in yellow in Figure 7 (lines 30, 31, 35, 55). The developer folds `upperBound(this, result)` on line 55 to show that `findMax` returns an upper bound of the list. The `upperBound` predicate is constructed from an `upperBoundSeg` predicate for the whole list and `result`. To achieve this `upperBoundSeg` predicate, the developer determines that the loop invariant (lines 34, 35) should contain `upperBoundSeg(this.head, curr, max)`. In other words, the loop should produce a value `max` that is the upper bound of the list from its head to the current node at every iteration. Then, when the loop terminates, `max` (`result`) will be an upper bound of the whole list (`upperBoundSeg(this.head, null, max)`). The additional folds before the loop, on lines 30, 31, are used to build up the `upperBoundSeg` for the first loop iteration.

As before, both accessibility predicates and `contains(this, result)` are dynamically verified. However, the verifier now statically establishes that `upperBound(this, result)` holds, at an unfortunate cost. The loop invariant (lines 34, 35) must be preserved for every iteration of the loop, but the developer has only constructed a proof for the first iteration (lines 30, 31). As a result, imprecision introduced by the invariant is used to prove that the invariant holds for the remaining iterations. That is, the invariant is dynamically checked—the list is traversed from its head to the current node—at every iteration beyond the first!

Appalled by this dynamic checking overhead, the developer decides to construct the missing static proofs. The resulting specifications are highlighted in green in Figure 7 (lines 48–50, 58–70). Since the loop's `else` case (lines 46–51) does not modify `max`, the developer focuses their effort here. Their goal is to show that `max` is an upper bound of the list from its head to the next traversed node (line 47). To achieve this, an empty `upperBoundSeg` starting and ending on the next node (line 48) is extended with the previous (current) node (line 49). This creates an `upperBoundSeg` predicate from the current node to the next node. The extension is justified by the negation of the if condition `curr.val ≤ max` (line 38). Then, the developer achieves their proof goal for the `else` case by appending (lines 50, 58–70) the `upperBoundSeg` predicate from the head of the list to the

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

class List {
  Node head;

  predicate upperBound(List l, int bound) =
      ? ∗ upperBoundSeg(l.head, null, bound)

  predicate upperBoundSeg(Node from, Node to, int bound)
      = ? ∗ if (from == to) then true else
        from.val ≤ bound ∗
        upperBoundSeg(from.next, to, bound)

  predicate contains(List l, int val) =
      ? ∗ containsSeg(l.head, null, val)

  predicate containsSeg(Node from, Node to, int val) =
      ? ∗ if (from == to) then false else
        if (from.val == val) then true else
          containsSeg(from.next, to, val)

int findMax()
  requires ? ∗ this.head != null
  ensures upperBound(this, result) ∗
    contains(this, result)
  {
    int max = this.head.val;
    Node curr = this.head.next;

    fold upperBoundSeg(this.head.next, curr, max);
    fold upperBoundSeg(this.head, curr, max);

    while (curr != null)
    invariant ?
      ∗ upperBoundSeg(this.head, curr, max)
    {
      Node x = curr;
      if (curr.val > max) {
        int oldMax = max;
        max = curr.val;
        curr = curr.next;
        fold upperBoundSeg(x.next, curr, max);
        fold upperBoundSeg(x, curr, max);
        upperBoundLemma(this.head, x, oldMax, max);
        appendLemma(this.head, x, curr, max);
      } else {
        curr = curr.next;
        fold upperBoundSeg(x.next, curr, max);
        fold upperBoundSeg(x, curr, max);
        appendLemma(this.head, x, curr, max);
      }
    }

    result = max;
    fold upperBound(this, result);
  }

void appendLemma(Node a, Node b,
                 Node c, int val)
  requires upperBoundSeg(a, b, val) ∗
    upperBoundSeg(b, c, val)
  ensures upperBoundSeg(a, c, val)
{
  if (a == b) {
  } else {
    unfold upperBoundSeg(a, b, val);
    appendLemma(a.next, b, c, val);
    fold upperBoundSeg(a, c, val);
  }
}

void upperBoundLemma(Node a, Node b,
                     int oldVal, int newVal)
  requires oldVal ≤ newVal ∗
    upperBoundSeg(a, b, oldVal)
  ensures upperBoundSeg(a, b, newVal)
{
  if (a == b) {
    fold upperBoundSeg(a, b, newVal);
  } else {
    unfold upperBoundSeg(a, b, oldVal);
    appendLemma(a.next, b, oldVal, newVal);
    fold upperBoundSeg(a, b, newVal);
  }
}

}
```

1^{st} increment (most imprecise of the 4)	3^{rd} increment
2^{nd} increment	4^{th} increment (most precise of the 4)

Fig. 7. Incrementally more precise ways to gradually verify `findMax` from Figure 6

current node (loop invariant, lines 34, 35) to the `upperBoundSeg` predicate from the current node to the next node.

Now, the gradual verifier can statically prove that the loop invariant is always preserved by the `else` branch. However, the verifier still dynamically checks the invariant on each loop iteration executing the `then` branch. The other dynamic checks for accessibility predicates and the `contains` predicate also still remain.

Finally, the developer generates specifications for the `then` branch, highlighted in blue in Figure 7 (lines 42–45, 72–85). As in the `else` case, the developer's goal is to show that `max` is an upper bound of the list from its head to the next traversed node (line 41). Here, however, `max` is assigned the current node's value (line 40). The assignment justfies the build up of the `upperBoundSeg` predicate from the current node to the next node (lines 42, 43). But, unlike in the `else` case, the loop invariant's `upperBoundSeg` predicate applies to an old `max` value rather than the current (new) one. The old value happens to be less than the current one (`then` condition, line 38), so the current `max` is an upper bound of the list from its head to the current node. The developer proves this fact with `upperBoundLemma` (lines 44, 72–85). Finally, as before, the developer uses `appendLemma` (lines 45, 58–70) to achieve the proof goal for the `then` case. This last increment allows the gradual verifier to prove that `findMax` returns an upper bound of the list completely statically. Only accessibility predicates for heap accesses and `contains(this, result)` are dynamically checked. The developer can stop here, or work further on either proving `contains(this, result)` or specifying accessibility predicates.

By using gradual verification on `findMax`, the developer is able to manage the complexity of meeting proofs obligations incrementally. The developer could have stopped at any of the aforementioned increments and be certain, in the absence of runtime checking errors, that the program returns the greatest element of the list and accesses only owned heap locations. Gradual verification enables the exploration of cost-benefit tradeoffs between static reasoning effort and runtime overhead.

4 CHALLENGES OF RECURSIVE HEAP DATA STRUCTURES

While the basic principles of gradual program verification have already been laid out by Bader et al. [2018], their work only accounts for pre- and postconditions that include basic logical and arithmetic formulas. The contribution of this work is to scale these basic principles to deal with realistic programming scenarios that involve recursive heap data structures.

This section explains the challenges involved in accounting for implicit dynamic frames (IDF) [Smans et al. 2009] and recursive abstract predicates [Parkinson and Bierman 2005] in the context of gradual program verification. We also informally outline our novel solutions to these challenges, which will be formally developed in the following sections.

4.1 Gradual Verification of Heap Ownership

Adapting the Abstracting Gradual Typing approach [Garcia et al. 2016] to the verification setting gives meaning to imprecise formulas such as $x > 10 \land ?$ by considering all the *logically consistent strengthenings* of such formulas [Bader et al. 2018; Lehmann and Tanter 2017]. For instance, $x > 10 \land ?$ *consistently implies* $x > 20$, but not $x < 0$. In the latter case, the formula $x < 0$ contradicts the static part of the imprecise formula $x > 10$. In the former case, if we definitely know that $x > 10$, then it might optimistically be the case that $x > 20$ as well. Of course, in order to preserve soundness, optimistically assuming $x > 20$ when one only definitely knows that $x > 10$ requires a runtime check to corroborate that the value bound to x at runtime is indeed greater than 20.

As we have seen in prior sections, when dealing with heap data structures, the logic—IDF in our case—includes more than arithmetic: we need to be able to talk about heap separation and ownership of heap cells. How are we to extend the interpretation of imprecise formulas in such a setting, and how can we soundly track optimistic assumptions?

Imprecise Heap Formulas. When using IDF in a static verifier, one must make sure that formulas are self-framed. For instance, `this.head != null` is not self-framed, because it does not explicitly mention the accessibility predicate needed to evaluate the formula. The formula **acc**(`this.head`) ∗ `this.head != null` is self-framed. We want to ensure that programmers can smoothly strengthen specifications, and one logical kind of strengthening is adding accessibility predicates that were previously missing. Accordingly, in our design imprecise formulas must optimistically allow ? to stand in for accessibility predicates that are necessary for framing. Furthermore, this is true whether the imprecise formula appears directly in an assertion or indirectly in the definition of an abstract predicate. Indeed, in IDF, framing can sometimes come from an abstract predicate. For instance, `acyclic(this) * unfolding acyclic(this) in this.head != null` is self-framed if the body of `acyclic(l)` includes **acc**(`l.head`). Thus, our semantics for imprecise formulas must allow ? to denote not only for predicates such as `acyclic(this)`, but also any unfoldings of them that are necessary to frame the static part of the formula. These semantic choices support different scenarios described in the previous section.

Runtime Checking of Ownership. For a gradual verifier to be sound, optimistic assumptions made statically due to imprecision must be safeguarded dynamically through runtime checks. Extending gradual verification to IDF by allowing imprecision to account for missing accessibility predicates means that we need to keep track of ownership in the runtime system. In particular, we design a runtime that tracks and updates a set of heap locations at every program point, indicating current ownership. Heap locations are added to this set when objects are created. Each time a field is accessed, the set of owned locations is looked up: if the corresponding permission is found, the check succeeds, otherwise a runtime error is raised.

At a call site, if an owned heap location is required by the precondition of the callee, then it is removed from the owned locations of the caller. When the callee finishes executing, all callee owned heap locations are passed to the caller.

The challenge here is how to deal with imprecise preconditions, either directly or via an imprecise abstract predicate. In order to maximize the ability for the callee to execute properly, an imprecise precondition has to require *all* the owned heap locations of the caller. Indeed, said imprecision might potentially denote any location owned by the caller, not already passed statically, and effectively required in the callee. Not transferring its ownership means the callee might error out at runtime.

4.2 Gradual Verification of Recursive Predicates

Recursive predicates can be dealt with in two different manners in program verification [Summers and Drossopoulou 2013]: either *iso-recursively*—in which case to be able to exploit a predicate instance, one needs to explicitly unfold it, and vice versa, to explicitly fold it back to establish it—or *equi-recursively*—in which case a predicate is deemed identical to its unfolding, which need not be specified explicitly. These two approaches have complementary strengths, which, we argue, are particularly relevant when apprehending gradual verification. The iso-recursive approach is critical for making static reasoning manageable for tools (and for humans who must deal with the error messages reported by these tools) because it breaks reasoning into small steps. In contrast, the equi-recursive approach is much more convenient in a dynamic setting, where the runtime system can automatically unfold predicates as needed, and so the user does not have to write explicit folds and unfolds.

In this work, we propose a novel design that achieves the benefits of both approaches. Statically, the gradual verifier treats recursive predicate instances *iso-recursively*: programmers can specify

$$
\begin{array}{rcll}
x, y, z & \in & \mathit{VAR} & \textit{(variables)} \\
v & \in & \mathit{VAL} & \textit{(values)} \\
e & \in & \mathit{EXPR} & \textit{(expressions)} \\
s & \in & \mathit{STMT} & \textit{(statements)} \\
o & \in & \mathit{LOC} & \textit{(object Ids)} \\
\mathtt{P} & ::= & \overline{\mathtt{cls}}\ s & \\
\mathtt{cls} & ::= & \mathtt{class}\ C\ \{\ \overline{\mathtt{field}}\ \overline{\mathtt{pred}}\ \overline{\mathtt{method}}\ \} & \\
\mathtt{field} & ::= & T\ f; & \\
\mathtt{pred} & ::= & \mathsf{predicate}\ p(\overline{T\ x}) = \theta & \\
T & ::= & \mathsf{int} \mid \mathsf{bool} \mid C \mid \top & \\
\mathtt{method} & ::= & T\ m(\overline{T\ x})\ \mathtt{contract}\ \{s\} & \\
\mathtt{contract} & ::= & \mathsf{requires}\ \theta\ \mathsf{ensures}\ \theta & \\
\oplus & ::= & + \mid - \mid * \mid \backslash & \\
\odot & ::= & \neq \mid = \mid < \mid > \mid \leq \mid \geq &
\end{array}
$$

$$
\begin{array}{rcll}
f & \in & \mathit{FIELDNAME} & \textit{(field names)} \\
m & \in & \mathit{METHODNAME} & \textit{(method names)} \\
C & \in & \mathit{CLASSNAME} & \textit{(class names)} \\
p & \in & \mathit{PREDNAME} & \textit{(predicate names)} \\
s & ::= & \mathsf{skip} \mid s\ ;\ s \mid T\ x \mid x := e \mid x.f := y & \\
 & & \mid \mathsf{if}\ (e)\ \{\ s\ \}\ \mathsf{else}\ \{\ s\ \} & \\
 & & \mid \mathsf{while}\ (e)\ \mathsf{inv}\ \theta\ \{\ s\ \} \mid x := \mathsf{new}\ C & \\
 & & \mid y := z.m(\overline{x}) \mid \mathsf{assert}\ \phi \mid \mathsf{fold}\ p(\overline{e}) & \\
 & & \mid \mathsf{unfold}\ p(\overline{e}) & \\
e & ::= & v \mid x \mid e \oplus e \mid e \odot e \mid e.f & \\
x & ::= & \mathsf{result} \mid id \mid \mathbf{old}(id) \mid \mathsf{this} & \\
v & ::= & n \mid o \mid \mathsf{null} \mid \mathsf{true} \mid \mathsf{false} & \\
\phi & ::= & \mathsf{true} \mid \mathsf{false} \mid e \odot e \mid p(\overline{e}) \mid \mathbf{acc}(e.f) & \\
 & & \mid \mathsf{if}\ e\ \mathsf{then}\ \phi\ \mathsf{else}\ \phi & \\
 & & \mid \mathsf{unfolding}\ p(\overline{e})\ \mathsf{in}\ \phi \mid \phi \land \phi \mid \phi * \phi & \\
\theta & ::= & \text{self-framed}\ \phi &
\end{array}
$$

Fig. 8. $\mathrm{SVL_{RP}}$: Syntax

folds and unfolds in the precise parts of their pre- and postconditions, as well as in program statements, just as they would with mainstream static verifiers. By exploiting syntax, verification becomes simply algorithmic for tools to implement, and visually clear for humans to keep track of the underlying activity of the verifier.

In contrast, dynamically, predicate instances are checked equi-recursively. An equi-recursive evaluation of predicate instances is the natural choice for dynamic checking, as the runtime system can simply execute the predicate as if it were a function. Crucially, an equi-recursive approach to program evaluation allows users to leave out fold and unfold statements, which one can expect to be the default for partially (or un-)verified code. Seen dually, adopting an iso-recursive runtime approach while allowing programmers to omit (un)folding statements would mean trying to automatically infer when to actually perform (un)folding. Known approaches to this are heuristic, meaning that some well-behaved code could be conservatively rejected when made imprecise enough. This would result in a violation of the dynamic gradual guarantee [Siek et al. 2015], whose motto is that *losing precision is harmless*.

Therefore we argue that combining iso- and equi-recursive treatments of recursive predicates is *required* in order to achieve a proper gradual verifier: statically, the iso-recursive approach ensures algorithmic checking, and dynamically, the equi-recursive approach allows imprecise code to run smoothly.

5 $\mathbf{SVL_{RP}}$

Following the AGT methodology [Garcia et al. 2016], gradual verification is built on top of static verification. Therefore, we first formally present a statically verified language supporting a propositional specification logic extended with implicit dynamic frames (IDF) and recursive abstract predicates. This language, called $\mathrm{SVL_{RP}}$, is directly inspired by Summers and Drossopoulou [2013]. Readers familiar with static verification might want to read through this section anyway, because it sets up notations and key concepts used in the gradualization (§6).

5.1 Syntax & Static Semantics

The complete syntax of $\mathrm{SVL_{RP}}$ can be found in Figure 8. Programs consist of classes and statements. Classes contain publicly accessible fields, predicates, and methods. Statements include the empty statement, sequences, variable declarations, variable and field assignments, conditionals, while loops, object allocations, method calls, assertions, as well as fold and unfold statements. Expressions can appear in specifications, and therefore cannot modify the heap. They consist of literal values (integers, objects, `null`, and booleans), variables, arithmetic expressions, comparisons, and field accesses. Methods have contracts consisting of pre- and postconditions, which are formulas represented by ϕ. Formulas join boolean values, comparisons, predicate instances, accessibility predicates, conditionals, and unfoldings via the non-separating conjunction $\wedge$ or the separating conjunction $*$. Note that θ refers to a *self-framed* formula [Smans et al. 2009], formally defined in §5.2.4. We require pre- and postconditions, predicate definitions, and loop invariants to be self-framed.

Looking ahead to gradual verification, we would like formulas to be efficiently evaluable at runtime—and in the presence of accessibility predicates, efficient evaluation requires knowing which branch of a disjunction to evaluate. Therefore, we include a conditional `if` construct in formulas instead of disjunction $\vee$.

As the focus of this work is not on typing, we only consider well-formed and well-typed programs, which is standard and not formalized here. Additionally, variables are declared and initialized before use, and class, predicate, and method names are unique. Finally, contracts should only contain variables that are in scope: a precondition can only contain the method's parameters $\overline{x_i}$ and `this` and a postcondition can only contain the special variable `result`, `this`, and dummy variables $\overline{old(x_i)}$.

5.2 Formula Semantics

In this section, we give meaning to formulas in SVL_{RP}. We also give related definitions for formula satisfiability, implication, footprint computation, and framing. The semantics and related definitions are inspired by Summers and Drossopoulou [2013] and Bader et al. [2018].

5.2.1 Equi-Recursive Evaluation. Evaluating the truth of formulas requires a heap H, a variable environment $\rho \in \textsc{Env}$, and a dynamic footprint $\pi \in \textsc{DynFprint} = \mathcal{P}(\textsc{Loc} \times \textsc{FieldName})$. A heap H is a partial function from heap locations to a value mapping of object fields, *i.e.* $\textsc{Heap} = \textsc{Loc} \rightharpoonup (\textsc{FieldName} \rightharpoonup \textsc{Val})$. Additionally, we introduce a big-step evaluation relation for expressions $H, \rho \vdash e \Downarrow v$, which is standard (and defined in Appendix Fig. 18). An expression e is evaluated according to $H, \rho \vdash e \Downarrow v$ yielding value v. The heap H is used to look up fields and the local variable environment ρ to look up variables.

Then, formula evaluation $\cdot \vDash_E \cdot \subseteq \textsc{Mem} \times \textsc{Formula}$ determines the truth of a formula using heap H, variable environment ρ, dynamic footprint π, and an equi-recursive interpretation of predicate instances. Select rules for formula evaluation are given in Figure 9 (complete rules are in Appendix Fig. 19). EvAcc checks whether access demanded by a formula is provided by the dynamic footprint, *e.g.* **acc**(`l.head`) where `l` points to `o` is true when $\langle$`o`, `head`$\rangle \in \pi$. EvSepOp checks two separated subformulas against disjoint partitions of the dynamic footprint. This ensures that access to the same field is not granted twice; for instance, this ensures that **acc**($\texttt{l}_1$`.head`) $*$ **acc**($\texttt{l}_2$`.head`) references two distinct fields. In contrast, the rule for $\wedge$ (EvAndOp) checks both operands against the full dynamic footprint; therefore, **acc**($\texttt{l}_1$`.head`) $\wedge$ **acc**($\texttt{l}_2$`.head`) may reference the same fields. Further, EvPred checks the complete unrolling of a predicate instance using the function body_μ : $\textsc{PredName} \to \textsc{Expr}^* \to \textsc{SfrmFormula}$. Given a predicate name and arguments, this function returns the predicate's definition (from the ambient program[1]) after parameter substitution. We make sure that every argument e_i produces a value, only in order to line up with the isorecursive semantics. But we do not need to substitute the values into $\text{body}_\mu(p)(e_1, ..., e_n)$, because it already has the e_i's within it after parameter substitution. Finally, the rule for an unfolding (EvUnfolding) ignores the predicate unfolding, because it is an iso-recursive only construct. For example, `unfolding`

[1]Many relations we define are implicitly parameterized over the ambient program.

$$\frac{H, \rho \vdash e \Downarrow o \qquad H, \rho \vdash e.f \Downarrow v \qquad \langle o, f \rangle \in \pi}{\langle H, \rho, \pi \rangle \vDash_E \mathbf{acc}(\mathtt{e.f})} \text{EvAcc} \quad \frac{\langle H, \rho, \pi_1 \rangle \vDash_E \phi_1 \qquad \langle H, \rho, \pi_2 \rangle \vDash_E \phi_2}{\langle H, \rho, \pi_1 \uplus \pi_2 \rangle \vDash_E \phi_1 * \phi_2} \text{EvSepOp}$$

$$\frac{H, \rho \vdash e_1 \Downarrow v_1 \qquad \ldots \qquad H, \rho \vdash e_n \Downarrow v_n \qquad \langle H, \rho, \pi \rangle \vDash_E \mathrm{body}_\mu(p)(e_1, \ldots, e_n)}{\langle H, \rho, \pi \rangle \vDash_E p(e_1, \ldots, e_n)} \text{EvPred}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \phi}{\langle H, \rho, \pi \rangle \vDash_E \mathtt{unfolding}\ p(e_1, \ldots, e_n)\ \mathtt{in}\ \phi} \text{EvUnfolding}$$

Fig. 9. $\mathrm{SVL}_{\mathrm{RP}}$: Formula evaluation (select rules)

$$\frac{H, \rho \vdash e_1 \Downarrow v_1 \qquad \ldots \qquad H, \rho \vdash e_n \Downarrow v_n \qquad \langle p, v_1, \ldots, v_n \rangle \in \Pi}{\langle H, \rho, \Pi \rangle \vDash_I p(e_1, \ldots, e_n)} \text{EvPred}$$

Fig. 10. $\mathrm{SVL}_{\mathrm{RP}}$: Iso-recursive formula evaluation (select rule)

`acyclic(l) in l.head != null` is true exactly when `l.head != null` is true. Also, all the construct does is provide access to more heap locations in the predicate. The other rules are as expected.

5.2.2 Iso-Recursive Evaluation. We also introduce an iso-recursive formula evaluation semantics, used in static verification (§2). This semantics differs from its equi-recursive counterpart in §5.2.1 on the EvPred rule. Figure 10 presents the iso-recursive version of EvPred. It treats predicate instances as opaque permissions by checking whether a predicate instance demanded by a formula is justified by a dynamic permission set $\Pi \in \textsc{Permissions} = \mathcal{P}((\textsc{Loc} \times \textsc{FieldName}) \cup (\textsc{PredName} \times \textsc{Val}^*))$. Compared to a dynamic footprint, a dynamic permission set can contain dynamic predicate instances in addition to heap locations. For example, `acyclic(l)` where `l` points to `o` is true when $\langle \mathtt{acyclic}, \mathtt{o} \rangle \in \Pi$. Other than EvPred, the iso-recursive semantics is defined by replacing π in Figures 9 and 19 with Π.

5.2.3 Formula Satisfiability and Implication. Similar to SVL [Bader et al. 2018], formal definitions for formula satisfiability and implication rely on sets of H, ρ, and Π tuples that make formulas true. Definition 5.1 presents a function that produces these sets from formulas. Definitions 5.2 and 5.3 rely on Definition 5.1 to formally state formula satisfiability and implication respectively. Note that these definitions are iso-recursive in order to be implementable in static verification tools (§2).

Definition 5.1 (Denotational Formula Semantics). $[\![\cdot]\!] : \textsc{Formula} \rightarrow \mathcal{P}(\textsc{Heap} \times \textsc{Env} \times \textsc{Permissions})$
$[\![\phi]\!] \overset{\text{def}}{=} \{ \langle H, \rho, \Pi \rangle \in \textsc{Heap} \times \textsc{Env} \times \textsc{Permissions} \mid \langle H, \rho, \Pi \rangle \vDash_I \phi \}$

Definition 5.2 (Formula Satisfiability). A formula ϕ is satisfiable if and only if $[\![\phi]\!] \neq \emptyset$. Let $\textsc{SatFormula} \subset \textsc{Formula}$ be the set of satisfiable formulas. *Ex.* $\mathbf{acc}(\mathtt{l}_1\mathtt{.head}) * \mathbf{acc}(\mathtt{l}_2\mathtt{.head})$ is satisfiable since $\mathtt{l}_1$ may not equal $\mathtt{l}_2$. In contrast, $\mathbf{acc}(\mathtt{l}_1\mathtt{.head}) * \mathbf{acc}(\mathtt{l}_2\mathtt{.head}) * \mathtt{l}_1 = \mathtt{l}_2$ is unsatisfiable.

Definition 5.3 (Formula Implication). $\phi_1 \Longrightarrow \phi_2$ if and only if $[\![\phi_1]\!] \subseteq [\![\phi_2]\!]$.
Ex. $\mathtt{l.head.val} = 6 \Rightarrow \mathtt{l.head.val} \geq 5$, and $\mathtt{l.head.val} = 6 \not\Rightarrow \mathbf{acc}(\mathtt{l.head.val}) * \mathtt{l.head.val} \geq 5$ since $\mathbf{acc}(\mathtt{l.head.val})$ is missing on the left-hand side.

5.2.4 Footprints and Framing. A statically-verified language supporting IDF requires formal definitions for the footprint and framing of a formula. These definitions are also iso-recursive.

The *footprint* of a formula ϕ, denoted $\lfloor \phi \rfloor_{H,\rho}$, is simply the minimum set of permissions Π required to satisfy ϕ given a heap H and variable environment ρ:

$$\lfloor \phi \rfloor_{H,\rho} = \min \{ \Pi \in \textsc{Permissions} \mid \langle H, \rho, \Pi \rangle \vDash_I \phi \}$$

$$\frac{\langle H,\rho,\Pi\rangle \vDash_I \mathbf{acc}(\mathtt{e.f}) \qquad \langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} e}{\langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} e.f}\ \textsc{FrmField} \qquad \frac{\langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} e}{\langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} \mathbf{acc}(\mathtt{e.f})}\ \textsc{FrmAcc}$$

$$\frac{\langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} \phi_1 \qquad \langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} \phi_2}{\langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} \phi_1 * \phi_2}\ \textsc{FrmSepOp}$$

$$\frac{\langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} e_1 \qquad \ldots \qquad \langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} e_n}{\langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} p(e_1,\ldots,e_n)}\ \textsc{FrmPred}$$

$$\frac{\begin{array}{c}\langle H,\rho,\Pi\rangle \vDash_I p(e_1,\ldots,e_n) \qquad \langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} e_1 \qquad \ldots \qquad \langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} e_n \\ \langle H,\rho,\Pi'\rangle \vdash_{\mathtt{frmI}} \phi \qquad \Pi' = \Pi \cup \lfloor \mathrm{body}_\mu(p)(e_1,\ldots,e_n)\rfloor_{H,\rho}\end{array}}{\langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} \mathtt{unfolding}\ p(e_1,\ldots,e_n)\ \mathtt{in}\ \phi}\ \textsc{FrmUnfolding}$$

Fig. 11. $\mathrm{SVL_{RP}}$: Framing (select rules)

The footprint is defined (*i.e.* there exists a unique minimal set of permission Π) for formulas satisfiable under H and ρ. It can be more directly implemented by simply evaluating ϕ using H and ρ, granting and recording precisely the permissions required. The footprint of `l.head != null` is empty, while the footprint of **acc**(`l.head`) $*$ `l.head != null` is $\{\langle \mathtt{o}, \mathtt{head}\rangle\}$ when `l` points to `o`.

A formula is said to be *framed* by permissions Π iff it only mentions fields and unfolds predicates in Π. We give select inference rules for formula framing in Figure 11 and give the rest in Appendix Figure 20. Note that FrmUnfolding allows one unrolling of a predicate to frame a part of a formula. Now, formula ϕ is called *self-framed* (we write $\vdash_{\mathtt{frm}} \phi$) if for all H, ρ, Π $\langle H,\rho,\Pi\rangle \vDash_I \phi$ implies $\langle H,\rho,\Pi\rangle \vdash_{\mathtt{frmI}} \phi$. We define the set of self-framed formulas $\textsc{SfrmFormula} \stackrel{\text{def}}{=} \{\ \phi \in \textsc{Formula} \mid\ \vdash_{\mathtt{frm}} \phi\ \}$. `l.head != null` is not self-framed, because it can evaluate to `true` even when Π does not contain **acc**(`l.head`). On the other hand, **acc**(`l.head`) $*$ `l.head != null` is self-framed, because it does not evaluate to `true` unless Π contains **acc**(`l.head`). Similarly, `unfolding acyclic(l) in l.head != null` is not self-framed while `acyclic(l)` $*$ `unfolding acyclic(l) in l.head != null` is for `acyclic(l)` with body **acc**(`l.head`). We write θ to denote self-framed formulas.

5.2.5 Relating Permission Sets and Footprints. By using the *footprint* definition in §5.2.4, we can formally relate dynamic permission sets to dynamic footprints via the partial function $\langle\!\langle \cdot \rangle\!\rangle_H$ of type $\textsc{Permissions} \times \textsc{Heap} \rightharpoonup \textsc{DynFprint}$:

$$\langle\!\langle\, \Pi\, \rangle\!\rangle_H = \{\langle o, f\rangle \mid \langle o, f\rangle \in \Pi\} \cup \langle\!\langle\, \Pi'\, \rangle\!\rangle_H \qquad \mathtt{where}\ \Pi' = \cup_{\langle p, v_1,\ldots,v_n\rangle \in \Pi} \lfloor \mathrm{body}_\mu(p)(v_1,\ldots,v_n)\rfloor_{H,[]}$$

This function completely unrolls the predicate instances in a dynamic permission set gathering owned heap locations on the way. For example, given $\langle \mathtt{acyclic}, \mathtt{o}\rangle$, with `acyclic` defined precisely as in Figure 2, this function returns all the heap locations ($\{\langle \mathtt{o}, \mathtt{head}\rangle,\ \langle \mathtt{o.head}, \mathtt{val}\rangle, \langle \mathtt{o.head}, \mathtt{next}\rangle,\ \ldots\}$) in the list `o`. Note that $\langle\!\langle \cdot \rangle\!\rangle_H$ is only defined when predicates can be finitely unrolled.

5.3 Static Verification

Static verification relies on a *weakest liberal precondition calculus* [Dijkstra 1975] to generate verification conditions. We now present this WLP calculus, which is defined iso-recursively.

5.3.1 WLP Calculus. Select rules for a weakest liberal precondition function $\mathrm{WLP}(s,\theta)$ of type $\textsc{Stmt} \times (\textsc{SatFormula} \cap \textsc{SfrmFormula}) \rightharpoonup (\textsc{SatFormula} \cap \textsc{SfrmFormula})$ are given in Figure 12 (all rules are in Appendix Fig. 22). Note, we explicitly restrict the domain and codomain of the WLP function to contain only satisfiable and self-framed formulas. These restrictions are often ensured

$$
\begin{aligned}
\mathsf{WLP}(x := e, \theta) &= \max_{\Rightarrow} \{\ \theta' \mid \theta' \Rightarrow \theta[e/x] \ \wedge\ \theta' \Rightarrow \mathrm{acc}(e)\ \} \\
\mathsf{WLP}(x.f := y, \theta) &= \mathbf{acc}(\mathtt{x.f}) \ \wedge\ \theta[y/x.f] \\
\mathsf{WLP}(y := z.m(\overline{x}), \theta) &= \max_{\Rightarrow} \{\ \theta' \mid\ y \notin \mathsf{FV}(\theta') \ \wedge \\
&\qquad \exists \theta_f\ .\ \theta' \Rightarrow (z \neq \mathtt{null}) * \mathsf{mpre}(m)[z/\mathtt{this}, \overline{x/\mathtt{mparam}(m)}] * \theta_f \ \wedge \\
&\qquad \theta_f * \mathsf{mpost}(m)[z/\mathtt{this}, \overline{x/\mathbf{old}(\mathtt{mparam}(m))}, y/\mathtt{result}] \Rightarrow \theta\ \}
\end{aligned}
$$

Fig. 12. SVL_{RP}: Weakest liberal precondition calculus (select rules)

in Figure 12 by finding a maximum self-framed and satisfiable formula with respect to implication (the weakest formula).

The statement-specific rules for WLP are standard, save for specific care related to field accesses, accessibility predicates, and predicate instances. Rules for variable and field assignment, conditionals, and while loops produce accessibility predicates for field accesses in the program statement, *e.g.* the WLP for `y := l.head` must contain **`acc`**`(l.head)`. Some rules rely on the function $\mathrm{acc}(e) : \textsc{Expr} \longrightarrow \textsc{Formula}$ (Appendix Fig. 21), which returns a formula of accessibility predicates corresponding to field accesses in e. More interestingly, the rule for a method call frames off information in the method's postcondition from θ producing the frame θ_f. If the accessibility predicates and predicate instances in θ_f are not in the method's precondition, then θ_f is joined with the precondition to produce the WLP. Consider computing the WLP for the call to `insertLastHelper` on line 26 in Figure 2. In this example, θ = `acyclic(this)`, the precondition is `acyclic(this) * unfolding acyclic(this) in this.head != null`, and the postcondition is `acyclic(this)`. Therefore, θ_f = `true` and the WLP is `this != null * acyclic(this) * unfolding acyclic(this) in this.head != null * true`.

5.3.2 Static Verification. A SVL_{RP} program is statically verified if it is a *valid program*:

Definition 5.4 (Valid Method). A method with contract `requires` θ_p `ensures` θ_q, parameters $\overline{x}$, and body s is considered valid if $\theta_p \Rightarrow \mathsf{WLP}(s, \theta_q)[\overline{x/\mathbf{old}(x)}]$ holds.

Definition 5.5 (Valid Program). A program with entry point statement s is considered valid if `true` $\Rightarrow \mathsf{WLP}(s, \mathtt{true})$ holds, $\theta_i \wedge (\mathtt{e = true}) \Rightarrow \mathsf{WLP}(r, \theta_i)$ and $\theta_i \Rightarrow \mathrm{acc}(e)$ hold for all loops with condition e, body r, and invariant θ_i, and all methods are valid.

5.4 Dynamic Semantics

The soundness of static verification is relative to SVL_{RP}'s dynamic semantics, which we now expose.

5.4.1 Program States. Program states consist of a heap and a stack, *i.e.* $\textsc{State} = \textsc{Heap} \times \textsc{Stack}$. A stack is made of stack frames that contain a variable environment $\rho \in \textsc{Env}$, a dynamic footprint $\pi \in \textsc{DynFprint} = \mathcal{P}(\textsc{Loc} \times \textsc{FieldName})$, and a program statement $s \in \textsc{Stmt}$:

$$S \in \textsc{Stack} ::= E \cdot S \mid \mathsf{nil} \qquad \text{where} \qquad E \in \textsc{StackFrame} = \textsc{Env} \times \textsc{DynFprint} \times \textsc{Stmt}$$

During execution of an SVL_{RP} program, expressions and statements operate on the topmost variable environment ρ. Expressions and statements may additionally access and mutate the heap as long as the topmost dynamic footprint contains the corresponding object-field permissions. Thus, the memory accessible at any point of execution can be viewed as a tuple of type $\textsc{Mem} = \textsc{Heap} \times \textsc{Env} \times \textsc{DynFprint}$.

5.4.2 Reduction Rules. Figure 13 presents select rules for SVL_{RP}'s small-step semantics $\cdot \longrightarrow \cdot \subseteq \textsc{State} \times \textsc{State}$. Complete rules are in Appendix Figure 23. Notably, we structure the rules so as

$$\frac{\langle H,\, \rho,\, \pi \rangle \vDash_E \phi}{\langle H,\, \langle \rho,\, \pi,\, \texttt{assert}\ \phi\ ;\ s \rangle \cdot S \rangle \longrightarrow \langle H,\, \langle \rho,\, \pi,\, s \rangle \cdot S \rangle}\ \textsc{SsAssert}$$

$$\frac{\langle H,\, \rho,\, \pi \rangle \vDash_E \mathrm{acc}(e) \qquad H,\, \rho \vdash e \Downarrow v \qquad \rho' = \rho[x \mapsto v]}{\langle H,\, \langle \rho,\, \pi,\, x := e\ ;\ s \rangle \cdot S \rangle \longrightarrow \langle H,\, \langle \rho',\, \pi,\, s \rangle \cdot S \rangle}\ \textsc{SsAssign}$$

$$\frac{\begin{array}{c}\mathrm{method}(m) = T_r\ m(\overline{T\ x'})\ \texttt{requires}\ \theta_p\ \texttt{ensures}\ \theta_q\ \{\ r\ \} \qquad H,\, \rho \vdash z \Downarrow o \qquad \overline{H,\, \rho \vdash x \Downarrow v} \\ \rho' = [\texttt{this} \mapsto o,\, \overline{x' \mapsto v},\, \overline{\textbf{old}(x') \mapsto v}] \qquad \pi' = \langle\langle\, \lfloor \theta_p \rfloor_{H,\rho'}\, \rangle\rangle_H \qquad \pi' \subseteq \pi \qquad \langle H,\, \rho',\, \pi' \rangle \vDash_E \theta_p\end{array}}{\langle H,\, \langle \rho,\, \pi,\, y := z.m(\overline{x})\ ;\ s \rangle \cdot S \rangle \longrightarrow \langle H,\, \langle \rho',\, \pi',\, r\ ;\ \texttt{skip} \rangle \cdot \langle \rho,\, \pi \backslash \pi',\, y := z.m(\overline{x})\ ;\ s \rangle \cdot S \rangle}\ \textsc{SsCall}$$

$$\frac{\mathrm{mpost}(m) = \theta_q \qquad \langle H,\, \rho',\, \pi' \rangle \vDash_E \theta_q \qquad \rho'' = \rho[y \mapsto \rho'(\mathrm{result})]}{\langle H,\, \langle \rho',\, \pi',\, \texttt{skip} \rangle \cdot \langle \rho,\, \pi,\, y := z.m(\overline{x})\ ;\ s \rangle \cdot S \rangle \longrightarrow \langle H,\, \langle \rho'',\, \pi \cup \pi',\, s \rangle \cdot S \rangle}\ \textsc{SsCallFinish}$$

$$\frac{}{\langle H,\, \langle \rho,\, \pi,\, \texttt{fold}\, p(e_1, \ldots, e_n)\ ;\ s \rangle \cdot S \rangle \longrightarrow \langle H,\, \langle \rho,\, \pi,\, s \rangle \cdot S \rangle}\ \textsc{SsFold}$$

Fig. 13. $\mathrm{SVL}_{\mathrm{RP}}$: Small-step semantics (select rules)

to not require a sequence rule. This aligns the small-step semantics more closely with the WLP calculus, and makes the $\mathrm{SVL}_{\mathrm{RP}}$ soundness proof easier.

The semantics gets stuck when a statement accesses a field that the current state does not own, as specified in SsAssign. Notice that SsAssign relies on acc(e) to check the accessibility of field accesses on the right-hand side. The semantics also gets stuck when preconditions (SsCall), postconditions (SsCallFinish), loop invariants, or assertions (SsAssert) do not hold.

To determine whether a field access is valid at runtime, the semantics tracks a set of owned heap locations π. This set is expanded during allocation with heap locations for the object's fields. At a method call (SsCall) π is split into disjoint caller and callee sets using the method's precondition. The *callee set* π' is derived from the precondition's accessibility predicates and the accessibility predicates gained from unrolling the predicates in the precondition. Ownership of the heap locations in π' is passed to the callee, so the *caller set* is defined as $\pi \setminus \pi'$. After execution of the callee's body finishes (SsCallFinish), execution resumes at the call site. The callee returns to the call site ownership of all received heap locations and new heap locations gained during execution.

Notice that we treat predicates equi-recursively when we track π, determine whether field accesses are valid, and determine whether contracts, loop invariants, or assertions hold. We also treat folds and unfolds equi-recursively as skip statements (SsFold). $\mathrm{SVL}_{\mathrm{RP}}$'s dynamic semantics is equi-recursively defined so the gradual verifier, which builds on $\mathrm{SVL}_{\mathrm{RP}}$'s semantics, adheres to the dynamic gradual guarantee (as discussed in §4.2).

5.5 Soundness

As explained above, the dynamic semantics of $\mathrm{SVL}_{\mathrm{RP}}$ is designed to get stuck when assertions, method contracts, or loop invariants are violated during program execution. The dynamic semantics also gets stuck if a program accesses fields it does not own during execution. Thus informally, *soundness* says that valid $\mathrm{SVL}_{\mathrm{RP}}$ programs do not get stuck, *i.e.* verified programs respect program specifications at runtime. Just as with SVL [Bader et al. 2018], we use a syntactic statement of soundness via progress and preservation.

Now, we introduce the formal definition of a valid state in Definition 5.6. This definition is an invariant that relates the static verification and dynamic semantics of valid $\mathrm{SVL}_{\mathrm{RP}}$ programs. It also relates the formal statements of progress and preservation in Propositions 5.7 and 5.8. Informally, if the current program state satisfies the WLP of a program, then execution does not get stuck (progress), and after each step of execution, the new state satisfies the WLP of the remaining program (preservation).

Definition 5.6 (Valid State, Final State). We call the state $\langle H, \langle \rho_n, \pi_n, s_n\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \in$ STATE *valid* if $s_n = s$ `; skip` or `skip` for some $s \in$ STMT, $s_i = s_i'$ `; skip` for some $s_i' \in$ STMT for all $1 \leq i < n$, $\pi_i \cap \pi_j = \emptyset$ for all $1 \leq i \leq n$, $1 \leq j \leq n$ such that $i \neq j$, and $\langle H, \rho_i, \pi_i\rangle \vDash_E \mathsf{sWLP}_i(s_n \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathsf{true})$ for all $1 \leq i \leq n$ ($\mathsf{sWLP}_i(\overline{s}, \theta)$ is the i-th component of $\mathsf{sWLP}(\overline{s}, \theta)$). A state ψ is *final* if $\psi = \langle H, \langle \rho, \pi, skip\rangle \cdot \mathsf{nil}\rangle$ for some H, ρ, π.

Note that the definition above relies on sWLP, a stack-aware extension of WLP (defined in Appendix Fig. 24). sWLP ensures that access permissions are not duplicated in different stack frames. Program validity (Def. 5.5) gives the validity of the initial program state.

PROPOSITION 5.7 (SVL$_{\mathrm{RP}}$ PROGRESS). *If ψ is a valid non-final state then $\psi \longrightarrow \psi'$ for some ψ'.*

PROPOSITION 5.8 (SVL$_{\mathrm{RP}}$ PRESERVATION). *If ψ is a valid state and $\psi \longrightarrow \psi'$ for some ψ' then ψ' is a valid state.*

6 GVL$_{\mathrm{RP}}$: STATIC SEMANTICS

We now derive GVL$_{\mathrm{RP}}$, the gradually-verified language counterpart of SVL$_{\mathrm{RP}}$, essentially following the Abstracting Gradual Typing methodology [Garcia et al. 2016], whose main principles and mechanisms apply beyond type systems. This section presents the syntax and static semantics of GVL$_{\mathrm{RP}}$. §7 develops the runtime semantics, and §8 establishes the main properties of GVL$_{\mathrm{RP}}$.

6.1 Syntax

The syntax of GVL$_{\mathrm{RP}}$ is the same as SVL$_{\mathrm{RP}}$ except for the addition of gradual formulas $\widetilde{\phi}$. Gradual formulas replace formulas θ in method contracts, predicate definitions, and loop invariants:

$$\begin{array}{rcl} \mathtt{pred} & ::= & \mathtt{predicate}\ p(\overline{T\ x}) = \widetilde{\phi} \\ \mathtt{contract} & ::= & \mathtt{requires}\ \widetilde{\phi}\ \mathtt{ensures}\ \widetilde{\phi} \end{array} \qquad \begin{array}{rcl} s & ::= & \ldots \mid \mathtt{while}\ (e)\ \mathtt{inv}\ \widetilde{\phi}\ \{\ s\ \} \\ \widetilde{\phi} & ::= & \theta \mid ? * \phi \end{array}$$

A gradual formula is either a self-framed *syntactically precise formula* θ or an imprecise formula $? * \phi$. Note that the static part of an imprecise formula does not need to be self-framed (as discussed in §4.1) and ? is syntactic sugar for `? * true`. Additionally, the set of all gradual formulas is given by $\widetilde{\text{FORMULA}}$. A *syntactically* precise formula does not contain ? directly, *i.e.* it is not visibly partial. However, it may contain hidden ?s by containing predicates that, when unrolled, expose ?, *e.g.* `acyclic(l)` where `acyclic`'s body is ?. Self-framing is augmented to handle nested imprecision in GVL$_{\mathrm{RP}}$, and its new definition is given in §6.2. We will refer to formulas that do not contain ?, neither directly nor nested in predicates, as *semantically* precise formulas, *e.g.* `acyclic(l)` where `acyclic`'s body is **`acc`**`(l.head) * listSeg(l.head, null)` (as in Figs. 2 & 5). Note that all semantically precise formulas are syntactically precise, but not all syntactically precise formulas are semantically precise.

6.2 Framing

Definitions for framing and self-framing syntactically precise formulas in GVL$_{\mathrm{RP}}$ are redefined to handle imprecise predicate definitions exposed by the FRMUNFOLDING rule. For example, `acyclic(this)`'s body is analyzed for the permissions required to frame `this.head != null` in `unfolding acyclic(this) in this.head != null`. If `acyclic(this)`'s body is imprecise, then SVL$_{\mathrm{RP}}$'s framing definition would be undefined for this formula. Therefore, formula framing in GVL$_{\mathrm{RP}}$, $\langle H, \rho, \Pi\rangle \mathrel{\widetilde{\vdash}_{\mathtt{frmI}}} \phi$, is defined as in SVL$_{\mathrm{RP}}$ except for the FRMUNFOLDING rule:

$$\frac{\begin{array}{c}\langle H, \rho, \Pi\rangle \models_I p(e_1,\dots,e_n) \qquad \langle H, \rho, \Pi\rangle \,\widetilde{\vdash}_{\texttt{frmI}}\, e_1 \qquad \dots \qquad \langle H, \rho, \Pi\rangle \,\widetilde{\vdash}_{\texttt{frmI}}\, e_n \\ \langle H, \rho, \Pi'\rangle \,\widetilde{\vdash}_{\texttt{frmI}}\, \phi \qquad \Pi' = \Pi \cup \lfloor \mathrm{body}_\mu(p)(e_1,\dots,e_n)\rfloor_{\mathrm{TotalFP}(\phi, H, \rho), H, \rho}\end{array}}{\langle H, \rho, \Pi\rangle \,\widetilde{\vdash}_{\texttt{frmI}}\, \texttt{unfolding}\ p(e_1,\dots,e_n)\ \texttt{in}\ \phi}\ \widetilde{\text{FrmUnfolding}}$$

This rule differs from its $\mathrm{SVL_{RP}}$ counterpart in computing Π', which aides in framing ϕ. In particular, the retrieval of accessibility predicates and predicate instances from $\mathrm{body}_\mu(p)(e_1,\dots,e_n)$ now accounts for imprecision. The $\mathrm{TotalFP}(\cdot,\cdot,\cdot)$: Formula × Heap × Env → Permissions function (Appendix Fig. 25) returns the explicit and implicit iso-recursive permissions required by ϕ ($\{\langle \texttt{o}, \texttt{head}\rangle\}$ for `this.head != null` where `this` points to `o`). Then, a new footprint definition $\lfloor\widetilde{\phi}\rfloor_{\Pi,H,\rho}$ is used to either frame ϕ optimistically with this maximal permission set or precisely with calculated permissions from $\mathrm{body}_\mu(p)(e_1,\dots,e_n)$. The result depends on whether $\mathrm{body}_\mu(p)(e_1,\dots,e_n)$ is imprecise or precise, respectively (`acyclic`'s body is ? so $\{\langle \texttt{o}, \texttt{head}\rangle\}$ is used):

$$\lfloor\theta\rfloor_{\Pi,H,\rho} = \lfloor\theta\rfloor_{H,\rho} \qquad\qquad \lfloor ?\, *\, \phi\rfloor_{\Pi,H,\rho} = \Pi$$

Now, a formula ϕ is called *self-framed* (we write $\widetilde{\vdash}_{\texttt{frm}}\ \phi$) if for all H, ρ, Π, $\langle H,\rho,\Pi\rangle \models_I \phi$ implies $\langle H,\rho,\Pi\rangle \,\widetilde{\vdash}_{\texttt{frmI}}\, \phi$. We redefine the set of self-framed formulas: SfrmFormula $\stackrel{\text{def}}{=} \{\ \phi \in$ Formula $\mid \widetilde{\vdash}_{\texttt{frm}}\ \phi\ \}$, and we still write θ to denote self-framed formulas. As a result, `acyclic(this) * unfolding acyclic(this) in this.head != null` is self-framed when `acyclic`'s body is ?.

6.3 Interpretation of Gradual Formulas

Gradual formulas are given meaning by the set of precise formulas that they represent. The interpretation of gradual formulas is used to define variants of formula evaluation, formula implication, and the WLP calculus that operate over gradual formulas and are *consistent liftings* [Bader et al. 2018; Garcia et al. 2016] of their $\mathrm{SVL_{RP}}$ counterparts. Then, the static verification judgment in $\mathrm{GVL_{RP}}$ is defined similarly to $\mathrm{SVL_{RP}}$ using these lifted definitions. The set denoted by a gradual formula is obtained via a concretization function [Lehmann and Tanter 2017]:

Definition 6.1 (Concretization of Gradual Formulas). $\gamma : \widetilde{\text{Formula}} \rightharpoonup \mathcal{P}^{\text{Formula}}$ is defined as:

$$\begin{aligned}&\gamma(\theta) = \{\ \theta\ \}\\ &\gamma(?\, *\, \phi) = \{\ \theta' \in \text{SatFormula} \mid \theta' \Rightarrow \phi\ \} \text{ if } \phi \in \text{SatFormula}\\ &\gamma(?\, *\, \phi) \quad \text{undefined otherwise}\end{aligned}$$

The concretization of a syntactically precise formula is the singleton set of this formula. The concretization of an imprecise formula is the (infinite) set of syntactically precise formulas that are 1) satisfiable and 2) imply the static part of the imprecise formula. For example, $\gamma(? * x \geq 0) = \{x = 2,\ y = x * x \geq 0,\ \dots\}$. Notice, $x < 0 * x \geq 0 \notin \gamma(? * x \geq 0)$, because it is not satisfiable.

Novel compared to Bader et al. [2018]'s work is the requirement that all syntactically precise formulas represented by gradual formulas must be *self-framed* (§6.2). This extra condition allows ? to frame the static part of an imprecise formula, a requirement we motivated in §4.1. Additionally, γ treats predicates opaquely by relying on iso-recursively defined satisfiability, self-framing, and implication. We make this design choice, because γ is an integral part of $\mathrm{GVL_{RP}}$'s static verification system, which we want to be iso-recursive (§4.2). This choice has implications. For example, when both `p(x)` and `q(x)`'s bodies contain **acc**(`x.f`), `p(x)` $*$ `q(x)` is equi-recursively unsatisfiable but iso-recursively satisfiable. Therefore, `p(x)` $*$ `q(x)` $\in \gamma(? *$ `q(x)`$)$. On the other hand, **acc**(`x.f`) $*$ **acc**(`x.f`) $\notin \gamma(? *$ **acc**(`x.f`)$)$, since **acc**(`x.f`) $*$ **acc**(`x.f`) is also iso-recursively unsatisfiable.

Definition 6.1 induces a natural definition of the *(im)precision* of gradual formulas:

Definition 6.2 (Precision of Gradual Formulas). $\widetilde{\phi}_1$ is more precise (*i.e.* less imprecise) than $\widetilde{\phi}_2$, written $\widetilde{\phi}_1 \sqsubseteq \widetilde{\phi}_2$, if and only if $\gamma(\widetilde{\phi}_1) \subseteq \gamma(\widetilde{\phi}_2)$.

Ex. $? * \textbf{acc}(\texttt{l.head}) * \texttt{listSeg}(\texttt{l.head},\texttt{null}) \sqsubseteq\ ? * \textbf{acc}(\texttt{l.head})$.

Semantic Interpretation of Gradual Formulas. Since Definition 6.1 is interpreted iso-recursively, even if `acyclic`'s body is ?, we can have `acyclic(l) * unfolding acyclic(l) in l.head != null` $\in \gamma(? * \texttt{l.head != null})$. That is, γ in Definition 6.1 may give *syntactically* precise, but *semantically* imprecise formulas. We therefore need a semantic interpretation of gradual formulas that extends the concept of concretization to also cover imprecise predicate bodies. As a result, such a *semantic concretization* of gradual formulas would only give *semantically precise* formulas.

A difficulty with writing semantic concretization is that in order to fully interpret formulas, we require an additional function body_μ, which returns predicate bodies from the ambient program given a predicate instance, *e.g.* $\mathsf{body}_\mu\texttt{(acyclic)(this) = ?}$. Since body_μ may return imprecise formulas, we cannot use it to interpret formulas that we want to be semantically precise. Instead, we must rely on some new function $\mathsf{body}_\Delta : \textsc{PredName} \rightarrow \textsc{Expr}^* \rightarrow \textsc{Formula}$, which returns only precise formulas. As a result, we work with *local formulas* $\langle \phi, \mathsf{body}_\Delta \rangle \in \textsc{Formula} \times (\textsc{PredName} \rightarrow \textsc{Expr}^* \rightarrow \textsc{Formula})$ that explicitly drag along their body function.

Existing rules can easily be adjusted in order to deal with this new parameter, for example:

$$\frac{\mathsf{body}_\Delta(p)(e_1, \ldots, e_n) = \phi \qquad H, \rho \vdash e_1 \Downarrow v_1 \qquad \ldots \qquad H, \rho \vdash e_n \Downarrow v_n \qquad \langle H, \rho, \pi \rangle \vDash_E \langle \phi, \mathsf{body}_\Delta \rangle}{\langle H, \rho, \pi \rangle \vDash_E \langle p(e_1, \ldots, e_n), \mathsf{body}_\Delta \rangle}\ \textsc{EvPred}$$

The EvPred rule now uses body_Δ to lookup predicate bodies, rather than using the designated body_μ. Notice the function body_Δ is carried around for reference, simply making explicit what was previously assumed as constant and ambient in $\mathrm{SVL_{RP}}$.

Now, we can give an interpretation to *gradual body functions* $\widetilde{\mathsf{body}_\Delta}$ by concretizing them into sets of body_Δ functions that produce precise, self-framed formulas. Given a $\widetilde{\mathsf{body}_\Delta}$, Definition 6.3 returns a set of body_Δ functions constructed from formulas that are in the γ (Def. 6.1) of each gradual formula in $\widetilde{\mathsf{body}_\Delta}$. For example, if $\mathsf{dom}(\widetilde{\mathsf{body}_\Delta}) = \{\texttt{acyclic}\}$, $\widetilde{\mathsf{body}_\Delta}(\texttt{acyclic})(\texttt{l}) = ?$, and $\mathsf{body}_\Delta(\texttt{acyclic})(\texttt{l}) = \textbf{acc}(\texttt{l.head})$, then $\mathsf{body}_\Delta \in \gamma(\widetilde{\mathsf{body}_\Delta})$. Additionally, each body_Δ function must be well-formed with respect to self-framing, *i.e.* the body that body_Δ returns for each predicate must be self-framed with respect to the body_Δ function itself. For example, if $\mathsf{body}_\Delta(q)(\texttt{l}) = \texttt{acyclic(l)} *$ `unfolding acyclic(l) in l.head != null`, then $\mathsf{body}_\Delta(\texttt{acyclic})(\texttt{l})$ must contain $\textbf{acc}(\texttt{l.head})$.

Definition 6.3 (Concretization of Gradual Formulas (continued)). Concretization of a gradual body function $\gamma : (\textsc{PredName} \rightarrow \textsc{Expr}^* \rightarrow \widetilde{\textsc{Formula}}) \rightharpoonup \mathcal{P}^{\textsc{PredName} \rightarrow \textsc{Expr}^* \rightarrow \textsc{SfrmFormula}}$ is defined as:

$$\gamma(\widetilde{\mathsf{body}_\Delta}) = \{\ \mathsf{body}_\Delta = \lambda p_i \in \mathsf{dom}(\widetilde{\mathsf{body}_\Delta}).\ \lambda \overline{e} \in \textsc{Expr}^*.\ \theta_{p_i}[\overline{e}/\overline{\texttt{tmp}_i}] \mid \langle \theta_{p_1}, \theta_{p_2}, \ldots \rangle \in$$
$$\gamma(\widetilde{\mathsf{body}_\Delta}(p_1)(\overline{\texttt{tmp}_1})) \times \gamma(\widetilde{\mathsf{body}_\Delta}(p_2)(\overline{\texttt{tmp}_2})) \times \ldots, \forall p_i \in \mathsf{dom}(\widetilde{\mathsf{body}_\Delta}).\ \vdash_{\texttt{frm}} \langle \mathsf{body}_\Delta(p_i)(\overline{\texttt{tmp}_i}), \mathsf{body}_\Delta \rangle\ \}$$

where $\mathsf{dom}(\widetilde{\mathsf{body}_\Delta}) = \{\ p_1, p_2, \ldots\ \} \subseteq \textsc{PredName}$.

Given this partial function, we can concretize a gradual formula and its gradual body function, yielding a set of semantically precise self-framed formulas:

$$\gamma(\langle \widetilde{\phi}, \widetilde{\mathsf{body}_\Delta} \rangle) = \{\ \langle \theta, \mathsf{body}_\Delta \rangle \mid \theta \in \gamma(\widetilde{\phi}), \mathsf{body}_\Delta \in \gamma(\widetilde{\mathsf{body}_\Delta}),\ \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta \rangle\ \}$$

As before, Definition 6.3 allows us to give a natural (semantic) definition for formula precision:

Definition 6.4 (Precision of Formulas (continued)). $\langle \widetilde{\phi}_1, \widetilde{\mathsf{body}^1_\Delta} \rangle$ is more precise than $\langle \widetilde{\phi}_2, \widetilde{\mathsf{body}^2_\Delta} \rangle$, written $\langle \widetilde{\phi}_1, \widetilde{\mathsf{body}^1_\Delta} \rangle \sqsubseteq \langle \widetilde{\phi}_2, \widetilde{\mathsf{body}^2_\Delta} \rangle$ if and only if $\gamma(\langle \widetilde{\phi}_1, \widetilde{\mathsf{body}^1_\Delta} \rangle) \subseteq \gamma(\langle \widetilde{\phi}_2, \widetilde{\mathsf{body}^2_\Delta} \rangle)$.

6.4 Lifting Predicates

We lift predicates on formulas in $\mathrm{SVL_{RP}}$ to handle gradual formulas in $\mathrm{GVL_{RP}}$ such that they are consistent liftings of corresponding $\mathrm{SVL_{RP}}$ predicates. Following AGT [Garcia et al. 2016], the *consistent lifting* $\widetilde{P} \subseteq \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Formula}}$ of predicate $P \subseteq \textsc{Formula} \times \textsc{Formula}$ is defined as:

$$\widetilde{P}(\widetilde{\phi}_1, \widetilde{\phi}_2) \overset{\text{def}}{\iff} \exists \phi_1 \in \gamma(\widetilde{\phi}_1), \phi_2 \in \gamma(\widetilde{\phi}_2).\ P(\phi_1, \phi_2).$$

The existential in this definition expresses the optimistic nature of gradual semantics: we want a gradual predicate to be true if there exists any interpetation of ? that makes the static version of the predicate true.

Since we rely on an equi-recursive dynamic semantics for SVL_{RP} and GVL_{RP} and allow predicate definitions to be imprecise, we now give a semantic definition of gradual formula evaluation:

Definition 6.5 (Consistent Formula Evaluation).
Let $\cdot \,\widetilde{\models}\, \cdot \subseteq \textsc{Mem} \times (\widetilde{\textsc{Formula}} \times (\textsc{PredName} \longrightarrow \textsc{Expr}^* \longrightarrow \widetilde{\textsc{Formula}}))$ be defined inductively as

$$\frac{\langle H, \rho, \pi \rangle \models_E \langle \phi, \text{body}_\Delta \rangle \qquad \langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} \langle \phi, \text{body}_\Delta \rangle}{\langle H, \rho, \pi \rangle \,\widetilde{\models}\, \langle ? * \phi, \widetilde{\text{body}_\Delta} \rangle}$$

$$\frac{\langle H, \rho, \pi \rangle \models_E \langle \theta, \text{body}_\Delta \rangle \qquad \langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} \langle \theta, \text{body}_\Delta \rangle}{\langle H, \rho, \pi \rangle \,\widetilde{\models}\, \langle \theta, \widetilde{\text{body}_\Delta} \rangle}$$

where $\text{body}_\Delta = \lambda p \in \text{dom}(\widetilde{\text{body}_\Delta}).\ \lambda \overline{e} \in \textsc{Expr}^*.\ \text{static}(\widetilde{\text{body}_\Delta}(p)(\overline{e}))$

and $\text{static} : \widetilde{\textsc{Formula}} \rightarrow \textsc{Formula}$ s.t. $\text{static}(\theta) = \theta$ and $\text{static}(? * \phi) = \phi$.

Note that $\cdot \,\widetilde{\models}\, \cdot$ is a consistent lifting of $\cdot \models_E \cdot$ (with γ from Def. 6.3). Our definition is conveniently implementable for equi-recursive dynamic checking: it simply evaluates the static parts of predicates, and ensures that any heap accesses touch only owned locations. For example, if `acyclic`'s body is ? and `l` points to `o`, then `acyclic(l) * unfolding acyclic(l) in l.head != null` evaluates to true when `o.head` is owned and `o.head ≠ null`. The static part of ? is true, so `acyclic(l)` is ignored.

Additionally, gradual formula evaluation depends on an equi-recursive framing judgment for semantically precise formulas. The framing judgment $\langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} \phi$ is defined similarly (replacing Π with π and iso-recursive formula evaluation with equi-recursive formula evaluation) to its iso-recursive counterpart in SVL_{RP}, except for FrmPred and FrmUnfolding. Equi-recursive variants of these rules are:

$$\frac{\forall i,\ \langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} e_i \qquad \langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} \text{body}_\mu(p)(e_1, \dots, e_n)}{\langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} p(e_1, \dots, e_n)} \qquad \frac{\langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} \phi}{\langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} \texttt{unfolding}\ p(e_1, \dots, e_n)\ \texttt{in}\ \phi}$$

Then, a formula is said to be (equi-recursively) framed by permissions π if its complete unrolling only mentions fields in π. For example, `acyclic(l)`, where `acyclic`'s body is defined as in Figure 2, is framed by π if π contains all of list `l`'s heap locations. We can also easily adjust the equi-recursive framing judgment to pass around and use a body_Δ context, as described in §6.3.

In contrast to gradual formula evaluation (Lemma 6.5), gradual formula implication is a consistent lifting of SVL_{RP} formula implication with the syntactic interpretation of gradual formulas given in Definition 6.1. This is because SVL_{RP} implication is defined iso-recursively, *i.e.* hides imprecision in predicates. We give the definition for gradual formula implication in Lemma 6.6.

Definition 6.6 (Consistent Formula Implication).
Let $\cdot \,\widetilde{\Rightarrow}\, \cdot \subseteq \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Formula}}$ be defined inductively as

$$\frac{\theta_1 \Rightarrow \text{static}(\widetilde{\phi}_2)}{\theta_1 \,\widetilde{\Rightarrow}\, \widetilde{\phi}_2}\ \widetilde{\textsc{ImplStatic}} \qquad \frac{\theta \in \textsc{SatFormula} \qquad \theta \Rightarrow \phi_1 \qquad \theta \Rightarrow \text{static}(\widetilde{\phi}_2)}{? * \phi_1 \,\widetilde{\Rightarrow}\, \widetilde{\phi}_2}\ \widetilde{\textsc{ImplGrad}}$$

Here also, $\cdot \,\widetilde{\Rightarrow}\, \cdot$ is a consistent lifting of $\cdot \Rightarrow \cdot$ (with γ from Def. 6.3). For example, $? \,\widetilde{\Rightarrow}\, ? *$ **acc**(`l.head`) $*$ `l.head != null` because **acc**(`l.head`) $*$ `l.head != null` is satisfiable and implies the static part of both sides of the implication.

$$\begin{aligned}
&\widetilde{\mathrm{WLP}}(\texttt{if}\ (e)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}, \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\ \{\ \phi' \in \textsc{SatFormula} \mid \phi' \Rightarrow \texttt{if}\ e\ \texttt{then}\ \theta_1\ \texttt{else}\ \theta_2\ \wedge \\
&\qquad \phi' \Rightarrow \mathsf{acc}(e)\ \wedge\ \vdash_{\texttt{frm}} \langle \phi', \mathsf{body}_\Delta{}' \rangle\ \}\ \mid \theta_1 \in \gamma(\widetilde{\mathrm{WLP}}(s_1, \widetilde{\phi})),\ \theta_2 \in \gamma(\widetilde{\mathrm{WLP}}(s_2, \widetilde{\phi})), \\
&\qquad \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta_1, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_2, \mathsf{body}_\Delta{}' \rangle\ \}) \\
&\widetilde{\mathrm{WLP}}(y := z.m(\overline{x}), \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\ \{\ \phi' \in \textsc{SatFormula} \mid y \notin \mathsf{FV}(\phi')\ \wedge\ \vdash_{\texttt{frm}} \langle \phi', \mathsf{body}_\Delta{}' \rangle\ \wedge \\
&\qquad \exists \phi_f\ .\ \phi' \Rightarrow (z\ \neq\ \mathsf{null})\ *\ \theta_p[z/\mathsf{this}, \overline{x/\mathsf{mparam}(m)}]\ *\ \phi_f\ \wedge \\
&\qquad \phi_f\ *\ \theta_q[z/\mathsf{this}, \overline{x/\textbf{old}(\mathsf{mparam}(m))}, y/\mathsf{result}] \Rightarrow \theta\ \wedge\ \vdash_{\texttt{frm}} \langle \phi_f, \mathsf{body}_\Delta{}' \rangle\ \} \\
&\qquad \mid \theta \in \gamma(\widetilde{\phi}),\ \theta_p \in \gamma(\mathsf{mpre}(m)),\ \theta_q \in \gamma(\mathsf{mpost}(m)),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu), \\
&\qquad \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_p, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_q, \mathsf{body}_\Delta{}' \rangle\ \}) \\
&\widetilde{\mathrm{WLP}}(\texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi_i}\ \{\ s\ \}, \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\ \{\ \phi' \in \textsc{SatFormula} \mid \phi' \Rightarrow \mathsf{acc}(e)\ \wedge\ \vdash_{\texttt{frm}} \langle \phi', \mathsf{body}_\Delta{}' \rangle\ \wedge \\
&\qquad \exists \phi_f\ .\ \phi' \Rightarrow \theta_i\ *\ \phi_f\ \wedge\ \overline{x_i} \notin \mathsf{FV}(\phi_f)\ \wedge\ \vdash_{\texttt{frm}} \langle \phi_f, \mathsf{body}_\Delta{}' \rangle\ \wedge \\
&\qquad \phi_f\ *\ (\theta_i\ *\ (e\ =\ \mathsf{false}))[\overline{x_i/y_i}] \Rightarrow \theta[\overline{x_i/y_i}]\ \} \\
&\qquad \mid \theta \in \gamma(\widetilde{\phi}),\ \theta_i \in \gamma(\widetilde{\phi_i}),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_i, \mathsf{body}_\Delta{}' \rangle\ \}) \\
&\qquad \texttt{where}\ \overline{y_i}\ \texttt{are vars modified by the loop body}\ s\ \texttt{and}\ \overline{x_i}\ \texttt{are fresh} \\
&\widetilde{\mathrm{WLP}}(\texttt{fold}\ p(\overline{e}), \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\ \{\ \phi' \in \textsc{SatFormula} \mid \phi'\ *\ p(\overline{e}) \Rightarrow \theta\ \wedge\ \phi'\ *\ p(\overline{e}) \in \textsc{SatFormula}\ \wedge \\
&\qquad \vdash_{\texttt{frm}} \langle \phi' * \mathsf{body}_\Delta{}'(p)(\overline{e}), \mathsf{body}_\Delta{}' \rangle\ \} * \mathsf{body}_\Delta{}'(p)(\overline{e}) \in \textsc{SatFormula} \\
&\qquad \mid \theta \in \gamma(\widetilde{\phi}),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle\ \})
\end{aligned}$$

Fig. 14. $\mathrm{GVL}_{\mathrm{RP}}$: Weakest liberal precondition calculus (select rules).

6.5 Lifting Functions

Functions that operate over formulas in $\mathrm{SVL}_{\mathrm{RP}}$ must also be lifted to handle gradual formulas in $\mathrm{GVL}_{\mathrm{RP}}$. The resulting $\mathrm{GVL}_{\mathrm{RP}}$ functions should approximate consistent liftings of corresponding $\mathrm{SVL}_{\mathrm{RP}}$ functions. Following AGT [Garcia et al. 2016], given a partial function $f\ :\ \textsc{Formula} \rightharpoonup \textsc{Formula}$, its *consistent lifting* $\widetilde{f} : \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$ is defined as:

$$\widetilde{f}(\widetilde{\phi}) = \alpha(\{\ f(\phi) \mid \phi \in \gamma(\widetilde{\phi})\ \}).$$

Notice, the definition of a *consistent function lifting* requires an abstraction function α, which given a set of formulas produces the most precise gradual formula representing this set. We define $\alpha : \mathcal{P}^{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$ as $\alpha(\overline{\phi}) = \min_{\sqsubseteq}\{\ \widetilde{\phi} \in \widetilde{\textsc{Formula}} \mid \overline{\phi} \subseteq \gamma(\widetilde{\phi})\ \}$, *e.g.* $\alpha(\{\textbf{acc}(\texttt{l}_1\texttt{.head}), \textbf{acc}(\texttt{l}_1\texttt{.head}) * \textbf{acc}(\texttt{l}_2\texttt{.head})\}) = ? * \textbf{acc}(\texttt{l}_1\texttt{.head})$. Then, α clearly creates a Galois connection with γ from Def. 6.1.

Figure 14 shows select rules for $\widetilde{\mathrm{WLP}}$ (complete rules are in Appendix Fig. 26), which approximate the consistent function lifting of WLP. Rules for method call, while loop, and if statements lift the corresponding WLP rules with respect to two (while loop and if statements) or three (method call statements) formula parameters instead of one formula parameter as in other rules. These corresponding WLP rules rely on extra (often implicit) formula parameters that may be imprecise in $\mathrm{GVL}_{\mathrm{RP}}$, and therefore, must be accounted for in the lifting. Similarly, WLP implicitly exposes predicate definitions in body_μ through self-framing (§6.2) and in fold and unfold rules. In $\mathrm{GVL}_{\mathrm{RP}}$, predicate definitions may be imprecise, so non-sequence statement WLP rules are lifted with respect to body_μ. The $\widetilde{\mathrm{WLP}}$ rules are applied to a program in §3.1.

6.6 Lifting the Verification Judgment

We define static verification in $\mathrm{GVL}_{\mathrm{RP}}$ using lifted formula implication ($\widetilde{\Rightarrow}$, §6.4) and lifted WLP ($\widetilde{\mathrm{WLP}}$, §6.5):

Definition 6.7 (Valid Method). A method with contract requires $\widetilde{\phi}_p$ ensures $\widetilde{\phi}_q$, parameters $\overline{x}$, and body s is considered valid if $\widetilde{\phi}_p \widetilde{\Rightarrow} \widetilde{\mathsf{WLP}}(s, \widetilde{\phi}_q)[\overline{x/\mathbf{old}(x)}]$ holds.

Definition 6.8 (Valid Program). A program with entry point statement s is considered valid if $\mathtt{true} \widetilde{\Rightarrow} \widetilde{\mathsf{WLP}}(s, \mathtt{true})$ holds, $\widetilde{\phi}_i \wedge \mathrm{acc}(e) \wedge (\mathtt{e = true}) \widetilde{\Rightarrow} \widetilde{\mathsf{WLP}}(r, \widetilde{\phi}_i \wedge \mathrm{acc}(e))$ holds for all loops with condition e, body r, and invariant $\widetilde{\phi}_i$, and all methods are valid.

7 GVL$_{\text{RP}}$: DYNAMIC SEMANTICS

A valid GVL$_{\text{RP}}$ program will plausibly remain valid during each step of execution. To ensure that it does, the dynamic semantics of SVL$_{\text{RP}}$ are extended with runtime checks and considerations for imprecise specifications.

7.1 Footprint Splitting

To split dynamic footprints at method calls and loop entries in GVL$_{\text{RP}}$'s small-step semantics, we use $\lfloor \widetilde{\phi} \rfloor_{\pi,H,\rho}$:

$$\lfloor \theta \rfloor_{\pi,H,\rho} = \langle\langle\ \lfloor \theta \rfloor_{H,\rho}\ \rangle\rangle_{\pi,H} \qquad\qquad \lfloor ?\ *\ \phi \rfloor_{\pi,H,\rho} = \pi$$

This definition relies on $\langle\langle\ \Pi\ \rangle\rangle_{\pi,H}$: $\textsc{Permissions} \times \textsc{DynFprint} \times \textsc{Heap} \rightharpoonup \textsc{DynFprint}$, which returns the given dynamic footprint when any predicate bodies analyzed by the function are imprecise. Otherwise, the function returns the dynamic footprint generated from unrolling predicates in Π[2]:

$$\begin{aligned}
&\langle\langle\ \Pi\ \rangle\rangle_{\pi,H} = \{\langle o, f\rangle \mid \langle o, f\rangle \in \Pi\} \cup \pi' \\
&\mathtt{where}\ \pi' \ = \begin{cases} \pi & \mathtt{if}\ \exists \langle p, v_1, \ldots, v_n\rangle \in \Pi . \exists \phi \in \textsc{Formula}.\mathrm{body}_\mu(p)(v_1, \ldots, v_n) = ?\ *\ \phi \\ \langle\langle\ \Pi'\ \rangle\rangle_{\pi,H} & \mathtt{otherwise} \\ \mathtt{for}\ \Pi' = \cup_{\langle p, v_1,\ldots,v_n\rangle \in \Pi} \lfloor \mathrm{body}_\mu(p)(v_1, \ldots, v_n) \rfloor_{H,[]} & \end{cases}
\end{aligned}$$

Therefore, $\lfloor \widetilde{\phi} \rfloor_{\pi,H,\rho}$ returns the given dynamic footprint π when $\widetilde{\phi}$ is imprecise or contains nested imprecision, and it returns a more precise dynamic footprint computed when $\widetilde{\phi}$ is semantically precise. Example, if `acyclic`'s body is ?, then $\lfloor \mathtt{acyclic(l)} * \mathbf{unfolding}\ \mathtt{acyclic(l)}\ \mathtt{in}\ \mathtt{l.head}\ != \mathbf{null} \rfloor_{\pi,H,\rho}$ will return π. It will return all of list `l`'s heap locations when `acyclic` is defined as in Figures 2 & 5.

7.2 Small-Step Semantics

We give an augmented version of SVL$_{\text{RP}}$'s small-step semantics ($\cdot \widetilde{\longrightarrow} \cdot \subseteq \textsc{State} \times (\textsc{State} \cup \{\mathbf{error}\})$) for GVL$_{\text{RP}}$. We make considerations for imprecision and for runtime verification. Representative rules are given in Figure 15 (complete rules are in Appendix Fig. 27).

Imprecision in Specifications. Method preconditions, postconditions, and loop invariants are now checked with gradual formula evaluation (SsCall, SsCallFinish). Asserted formulas must also be checked with gradual formula evaluation due to potentially hidden imprecision (SsAssert). Additionally, we must ensure that introducing imprecision will not introduce a runtime error caused by lack of accessibility (dynamic gradual guarantee, Prop. 8.6). Therefore, if a method precondition in SsCall (or loop invariant) is imprecise or contains nested imprecision, then all owned heap locations are forwarded from the call site to the callee (or loop body) for execution. Otherwise, the call site's owned heap locations can be precisely transferred to the callee (or loop body) as in

[2] Note that $\langle\langle\ \Pi\ \rangle\rangle_{\pi,H}$ is a partial function, as it may not be well-defined if a predicate instance held in Π has an infinite completely unrolling and no nested imprecise predicates.

$$\frac{\langle H, \rho, \pi\rangle \mathrel{\widetilde{\vDash}} \langle ? * \phi, \mathsf{body}_\mu\rangle}{\langle H, \langle \rho, \pi, \mathtt{assert}\ \phi\ ;\ s\rangle \cdot S\rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \textsc{SsAssert}$$

$$\frac{\begin{array}{c}\mathsf{method}(m) = T_r\ m(\overline{T\ x'})\ \mathtt{requires}\ \widetilde{\phi}_p\ \mathtt{ensures}\ \widetilde{\phi}_q\ \{\ r\ \} \\ H, \rho \vdash z \Downarrow o \qquad \overline{H, \rho \vdash x \Downarrow v} \qquad \rho' = [\mathtt{this} \mapsto o, \overline{x' \mapsto v}, \overline{\mathbf{old}(x') \mapsto v}] \\ \pi' = \lfloor \widetilde{\phi}_p \rfloor_{\pi, H, \rho'} \qquad \pi' \subseteq \pi \qquad \langle H, \rho', \pi'\rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_p, \mathsf{body}_\mu\rangle\end{array}}{\langle H, \langle \rho, \pi, y := z.m(\overline{x})\ ;\ s\rangle \cdot S\rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho', \pi', r\ ;\ \mathtt{skip}\rangle \cdot \langle \rho, \pi \backslash \pi', y := z.m(\overline{x})\ ;\ s\rangle \cdot S\rangle}\ \textsc{SsCall}$$

$$\frac{\mathsf{mpost}(m) = \widetilde{\phi}_q \qquad \langle H, \rho', \pi'\rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_q, \mathsf{body}_\mu\rangle \qquad \rho'' = \rho[y \mapsto \rho'(\mathtt{result})]}{\langle H, \langle \rho', \pi', \mathtt{skip}\rangle \cdot \langle \rho, \pi, y := z.m(\overline{x})\ ;\ s\rangle \cdot S\rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho'', \pi \cup \pi', s\rangle \cdot S\rangle}\ \textsc{SsCallFinish}$$

Fig. 15. $\mathrm{GVL_{RP}}$: Small-step semantics adjusted from Fig. 13 (select rules)

$\mathrm{SVL_{RP}}$. Heap locations held after the callee's (or loop body's) execution are returned as usual to the call site.

Runtime Verification. Even for valid $\mathrm{GVL_{RP}}$ programs, when specifications are imprecise the formula evaluation premises in $\mathrm{GVL_{RP}}$'s small-step semantics are not guaranteed to hold. Therefore, these premises are turned into runtime checks. If an assertion, accessibility predicate, method precondition, method postcondition, or loop invariant does not hold in a program state where it should, then program execution steps into a dedicated **error** state (extra rules illustrating this can be found in the Appendix in Fig. 27).

8 PROPERTIES OF $\mathrm{GVL_{RP}}$

$\mathrm{GVL_{RP}}$ is a sound gradually-verified language that conservatively extends $\mathrm{SVL_{RP}}$ and adheres to gradual guarantees. $\mathrm{GVL_{RP}}$ is a *conservative extension* of $\mathrm{SVL_{RP}}$—meaning that $\mathrm{GVL_{RP}}$ and $\mathrm{SVL_{RP}}$ coincide on fully precise programs—by construction following the Abstracting Gradual Typing methodology [Bader et al. 2018; Garcia et al. 2016].

Soundness. Soundness for $\mathrm{GVL_{RP}}$ is conceptually similar to soundness for $\mathrm{SVL_{RP}}$ except that a $\mathrm{GVL_{RP}}$ program may step to a dedicated error state when runtime verification fails. We establish soundness via progress and preservation.

Definition 8.1 (Valid State, Final State). We call the state $\langle H, \langle \rho_n, \pi_n, s_n\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \in$ $\textsc{State}$ *valid* if $s_n = s\ ;\ \mathtt{skip}$ or $\mathtt{skip}$ for some $s \in \textsc{Stmt}$, $s_i = s_i'\ ;\ \mathtt{skip}$ for some $s_i' \in \textsc{Stmt}$ $\forall\ .\ 1 \leq i < n$, and $s_i = s_i^1\ ;\ s_i^2$ for some $s_i^1, s_i^2 \in \textsc{Stmt}$ where s_i^1 is a method call or while loop statement $\forall\ .\ 1 \leq i < n$. A state ψ is *final* if $\psi = \langle H, \langle \rho, \pi, \mathtt{skip}\rangle \cdot \mathsf{nil}\rangle$ for some H, ρ, π.

Proposition 8.2 ($\mathrm{GVL_{RP}}$ Progress). *If ψ is a valid non-final state then $\psi \mathrel{\widetilde{\longrightarrow}} \psi'$ for some ψ' or $\psi \mathrel{\widetilde{\longrightarrow}}$* **error**.

Proposition 8.3 ($\mathrm{GVL_{RP}}$ Preservation). *If ψ is a valid state and $\psi \mathrel{\widetilde{\longrightarrow}} \psi'$ for some ψ' then ψ' is a valid state.*

Gradual Guarantees. $\mathrm{GVL_{RP}}$ satisfies both the static and the dynamic gradual guarantees, originally formulated for gradual type systems [Siek et al. 2015], and first adapted to gradual verification by Bader et al. [2018]. These properties ensure in $\mathrm{GVL_{RP}}$ that decreasing the precision of specifications never breaks the verifiability and reducibility of a program, *i.e.* losing precision is harmless.

These properties rely on a notion of precision for programs. We say a program p_1 is more precise than program p_2 ($p_1 \sqsubseteq p_2$) if 1) p_1 and p_2 are equivalent except in terms of contracts, loop invariants, and/or predicate definitions, and 2) p_1's contracts, loop invariants, and predicate definitions are more precise than p_2's corresponding contracts, loop invariants, and predicate definitions. A contract `requires` $\widetilde{\phi}_p^1$ `ensures` $\widetilde{\phi}_q^1$ is more precise than contract `requires` $\widetilde{\phi}_p^2$ `ensures` $\widetilde{\phi}_q^2$ if $\widetilde{\phi}_p^1 \sqsubseteq \widetilde{\phi}_p^2$ and $\widetilde{\phi}_q^1 \sqsubseteq \widetilde{\phi}_q^2$. Similarly, a loop invariant (predicate definition) $\widetilde{\phi}_i^1$ is more precise than loop invariant (predicate definition) $\widetilde{\phi}_i^2$ if $\widetilde{\phi}_i^1 \sqsubseteq \widetilde{\phi}_i^2$.

Using this notion of program precision, the static gradual guarantee can now be stated as follows:

Proposition 8.4 (GVL$_{\text{RP}}$ Static gradual guarantee).
Let $p_1, p_2 \in$ Program *such that* $p_1 \sqsubseteq p_2$. *If* p_1 *is valid then* p_2 *is valid.*

In general, the static gradual guarantee ensures that reducing the precision of specifications never breaks static verification (*i.e.* makes a valid program invalid).

For the dynamic gradual guarantee, the fact that footprint tracking and splitting is influenced by increasing imprecision (*i.e.* increasing imprecision results in larger parts of footprints being passed up the stack) means that we must define an asymmetric *state precision* relation $\lesssim$:

Definition 8.5 (State Precision). Let $\psi_1, \psi_2 \in$ State. Then ψ_1 is more precise than ψ_2, written $\psi_1 \lesssim \psi_2$, if and only if all of the following applies:
a) ψ_1 and ψ_2 have stacks of size n and identical heaps.
b) ψ_1 and ψ_2 have stacks of variable environments that are identical.
c) Let $s^1_{1..n}$ and $s^2_{1..n}$ be the stack of statements of ψ_1 and ψ_2, respectively. Then for $1 \leq i \leq n$, $s^1_i \sqsubseteq s^2_i$:
$s \sqsubseteq s'$ if and only if s is a fold or unfold statement and s' is a skip statement or equal to s,
$s = \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \}$ and $s' = \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}'_i\ \{\ r\ \}$ where $\widetilde{\phi}_i \sqsubseteq \widetilde{\phi}'_i$,
$s = s_{c_1}; s_{c_2}$ and $s' = s'_{c_1}; s'_{c_2}$ where $s_{c_1} \sqsubseteq s'_{c_1}$ and $s_{c_2} \sqsubseteq s'_{c_2}$, or $s = s'$.
d) Let $\pi^1_{1..n}$ and $\pi^2_{1..n}$ be the stack of footprints of ψ_1 and ψ_2, respectively. Then the following holds for $1 \leq m \leq n$:

$$\bigcup_{i=m}^{n} \pi_i^1 \subseteq \bigcup_{i=m}^{n} \pi_i^2$$

Additionally, as long as it does not break the static gradual guarantee, we allow increased imprecision through dropped fold and unfold statements from one program to the next. This is reflected in condition c) in Definition 8.5 and an adjusted program precision definition $\sqsubseteq_d$. That is, a program p_1 is more precise than a program p_2 if 1) the programs are equivalent except for in terms of contracts, loop invariants, and/or predicate definitions and fold and unfold statements in p_1 may be replaced with `skip` statements in p_2, and 2) p_1's contracts, loop invariants, and predicate definitions are more precise than p_2's corresponding contracts, loop invariants, and predicate definitions. Now, the *dynamic gradual guarantee* can be given:

Proposition 8.6 (GVL$_{\text{RP}}$ Dynamic gradual guarantee).
Let $p_1, p_2 \in$ Program *such that* $p_1 \sqsubseteq_d p_2$, *and* $\psi_1, \psi_2 \in$ State *such that* $\psi_1 \lesssim \psi_2$.
If $\psi_1 \widetilde{\longrightarrow}_{p_1} \psi'_1$, *then* $\psi_2 \widetilde{\longrightarrow}_{p_2} \psi'_2$, *with* $\psi'_1 \lesssim \psi'_2$.

Since GVL$_{\text{RP}}$ adheres to the dynamic gradual guarantee, reducing the precision of specifications and/or dropping fold and unfold statements does not affect the program's observable behavior.

9 RELATED WORK

We have already discussed the most-closely related research, including the underlying logics [Parkinson and Bierman 2005; Reynolds 2002; Smans et al. 2009] and foundational work on gradual typing and gradual verification [Bader et al. 2018; Garcia et al. 2016; Siek and Taha 2007, 2006; Siek et al. 2015]. The contribution of this work compared to [Bader et al. 2018] is to identify and solve key technical challenges related to recursive heap data structures, namely semantically connecting iso- and equi-recursive interpretations of abstract predicates, and dynamically checking heap ownership.

Lehmann and Tanter [2017] extend the gradual typing paradigm to logical specifications in the form of refinement types. Their language setting is quite different from the one considered here: they deal with higher-order, purely functional programs, while we deal with first-order imperative programs. Therefore they do not have to consider heap ownership. Also, they do not deal with abstract recursive predicates. Combining both approaches in order to account for higher-order stateful programs is a challenging venue for future work.

Prior work on gradual typestate [Garcia et al. 2014; Wolff et al. 2011] and gradual ownership [Sergey and Clarke 2012] integrates static and dynamic checking of ownership of heap data structures. Neither of these efforts considered verifying logical assertions. Both predate the AGT framework that guided our design [Garcia et al. 2016], and the formulation of the gradual guarantees Siek et al. [2015]; it is unclear whether these guarantees are hold in these proposals.

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. Their approach cannot infer invariants not expressible as weakenings of postconditions; in contrast, our work can always insert run-time checks where specifications are insufficient for static verification. 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 of missing specifications needed for verification. Similarly, StaDy [Petiot et al. 2014] relies on a combination of static and dynamic analysis techniques to aide developers with debugging specifications. But, it does not reduce specification burden and does not support recursive data structures. Several 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 these heuristics in our setting may be challenging due to imprecise specifications, but it is a promising direction for future work.

10 CONCLUSION

Gradual verification is a promising way to enable more incrementality in proofs of programs: developers can focus on the most critical specifications first, benefiting from a combination of static and dynamic checking, and increase the scope of verification over time. By extending sound gradual verification to support programs that manipulate recursive heap data structures, we lay the groundwork for the application of these ideas to realistic programs. Our paper describes how we overcame several key technical challenges, including the semantics of imprecise formulas in the presence of accessibility predicates and recursive predicates, and consistency between iso-recursive static checking and equi-recursive dynamic checking. This opens the door to future work developing prototype gradual verifiers based on our theory, and exploring practical questions such as the efficiency of run-time verification in this setting.

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.

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (Aug. 1975), 453–457. https://doi.org/10.1145/360933.360975

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.

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.

Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. 2014. StaDy: Deep Integration of Static and Dynamic Analysis in Frama-C. (2014).

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.

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.

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

A APPENDIX

A.1 Full gradual verification examples

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

class List {
  Node head;

  predicate acyclic(List l) = ?

  void insertLast(int val)
    requires acyclic(this)
    ensures  acyclic(this)
  {
    acyclic(this) ⇛   ? * acyclic(this)
    ? * acyclic(this)
    unfold acyclic(this);
    ? ⇛   ? * acc(this.head)
    ? * acc(this.head)
    if (this.head == null) {
      ? * acc(this.head)
      this.head = new Node(val,null);
      ?
      fold acyclic(this);
      acyclic(this)
    } else {
      ? * this != null * this.head != null
      fold acyclic(this);
      this != null * acyclic(this) *
        unfolding acyclic(this) in
          this.head != null
      insertLastHelper(val);
      acyclic(this)
    }
    acyclic(this)
  }

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

■	Intermediate condition produced by $\widetilde{WLP}$	■	Dynamically checked right side of ⇛
■	Left side of ⇛	■	Statically checked right side of ⇛

Fig. 16. The gradual verification of Figure 3.

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

class List {
  Node head;

  predicate acyclic(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 acyclic(this)
    ensures  acyclic(this)
  {
    acyclic(this) ⇒ acyclic(this)
    acyclic(this)
    unfold acyclic(this);
    acc(this.head) * if this.head == null then true
      else listSeg(this.head, null)
    if (this.head == null) {
      acc(this.head)
      this.head = new Node(val,null);
      acc(this.head) * acc(this.head.val) *
        acc(this.head.next) *
      if (this.head.next == null) then true
        else ...
      fold listSeg(this.head.next, null);
    acc(this.head) * if (this.head == null) then true
      else acc(this.head.val) * acc(this.head.next)
        * listSeg(this.head.next, null)
      fold listSeg(this.head, null);
      acc(this.head) * listSeg(this.head, null)
      fold acyclic(this);
      acyclic(this)
    } else {
      acc(this.head) * listSeg(this.head, null) *
        this != null * this.head != null
      fold acyclic(this);
      this != null * acyclic(this) *
        unfolding acyclic(this) in
          this.head != null
      insertLastHelper(val);
      ? ⇛ acyclic(this)
      acyclic(this)
    }
  }

  void insertLastHelper(int val)
    requires acyclic(this) *
      unfolding acyclic(this) in
        this.head != null
    ensures  ?
  {
    acyclic(this) * unfolding acyclic(this) in
    this.head != null ⇛
    ? * acyclic(this) * unfolding acyclic(this)
    in this.head != null
    ? * acyclic(this) * unfolding acyclic(this)
    in this.head != null
    unfold acyclic(this);
    ? * acc(this.head) * this.head != null *
      listSeg(this.head, null)
    Node y = this.head;
    ? * y != null * listSeg(y, null)
    unfold listSeg(y, null);
    ? * y != null * acc(y.val) * acc(y.next) *
      listSeg(y.next, null)
    while (y.next != null)
      invariant y != null * acc(y.val) *
      acc(y.next) * listSeg(y.next, null)
    {
      (y != null * acc(y.val) * acc(y.next) *
      listSeg(y.next, null)) ∧ y.next != null
      ⇒ acc(y.next) * y.next != null *
        ⇒ listSeg(y.next, null)
      acc(y.next) * listSeg(y.next, null) *
        y.next != null
      y = y.next;
      listSeg(y, null) * y != null
      unfold listSeg(y, null);
      y != null * acc(y.val) * acc(y.next) *
        listSeg(y.next, null)
    }
    ? * acc(y.next)
    y.next = new Node(val,null);
    ?
  }
}
```

Intermediate condition produced by $\widetilde{WLP}$	Dynamically checked right side of ⇛
Left side of ⇛	Statically checked right side of ⇛

Fig. 17. The gradual verification of Figure 5.

A.2 $\mathbf{SVL_{RP}}$

A.2.1 Formula Semantics.

$$\frac{}{H, \rho \vdash v \Downarrow v}\ \textsc{EVal} \qquad \frac{}{H, \rho \vdash x \Downarrow \rho(x)}\ \textsc{EVar} \qquad \frac{H, \rho \vdash e \Downarrow o \qquad H(o) = \langle C, l\rangle}{H, \rho \vdash e.f \Downarrow l(f)}\ \textsc{EvField}$$

$$\frac{H, \rho \vdash e_1 \Downarrow v_1 \qquad H, \rho \vdash e_2 \Downarrow v_2}{H, \rho \vdash e_1 \oplus e_2 \Downarrow v_1 \oplus v_2}\ \textsc{EvOp} \qquad \frac{H, \rho \vdash e_1 \Downarrow v_1 \qquad H, \rho \vdash e_2 \Downarrow v_2}{H, \rho \vdash e_1 \odot e_2 \Downarrow v_1 \odot v_2}\ \textsc{EvComp}$$

Fig. 18. $\mathrm{SVL_{RP}}$: Expression dynamic semantics

$$\frac{}{\langle H, \rho, \pi\rangle \vDash_E \texttt{true}}\ \textsc{EvTrueExpr} \qquad \frac{H, \rho \vdash e_1 \odot e_2 \Downarrow \texttt{true}}{\langle H, \rho, \pi\rangle \vDash_E e_1 \odot e_2}\ \textsc{EvCompExpr}$$

$$\frac{H, \rho \vdash e \Downarrow o \qquad H, \rho \vdash e.f \Downarrow v \qquad \langle o, f\rangle \in \pi}{\langle H, \rho, \pi\rangle \vDash_E \mathbf{acc}(\texttt{e.f})}\ \textsc{EvAcc} \qquad \frac{\langle H, \rho, \pi\rangle \vDash_E \phi_1 \qquad \langle H, \rho, \pi\rangle \vDash_E \phi_2}{\langle H, \rho, \pi\rangle \vDash_E \phi_1 \wedge \phi_2}\ \textsc{EvAndOp}$$

$$\frac{\langle H, \rho, \pi_1\rangle \vDash_E \phi_1 \qquad \langle H, \rho, \pi_2\rangle \vDash_E \phi_2}{\langle H, \rho, \pi_1 \uplus \pi_2\rangle \vDash_E \phi_1 * \phi_2}\ \textsc{EvSepOp}$$

$$\frac{H, \rho \vdash e_1 \Downarrow v_1 \qquad \ldots \qquad H, \rho \vdash e_n \Downarrow v_n \qquad \langle H, \rho, \pi\rangle \vDash_E \mathrm{body}_\mu(p)(e_1, \ldots, e_n)}{\langle H, \rho, \pi\rangle \vDash_E p(e_1, \ldots, e_n)}\ \textsc{EvPred}$$

$$\frac{H, \rho \vdash e \Downarrow \texttt{true} \qquad \langle H, \rho, \pi\rangle \vDash_E \phi_T}{\langle H, \rho, \pi\rangle \vDash_E \texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F}\ \textsc{EvCondTrue} \qquad \frac{H, \rho \vdash e \Downarrow \texttt{false} \qquad \langle H, \rho, \pi\rangle \vDash_E \phi_F}{\langle H, \rho, \pi\rangle \vDash_E \texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F}\ \textsc{EvCondFalse}$$

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \phi}{\langle H, \rho, \pi\rangle \vDash_E \texttt{unfolding}\ p(e_1, \ldots, e_n)\ \texttt{in}\ \phi}\ \textsc{EvUnfolding}$$

Fig. 19. $\mathrm{SVL_{RP}}$: Formula evaluation

$$\frac{}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} v}\ \textsc{FrmVal} \qquad \frac{}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} x}\ \textsc{FrmVar}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_2}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \oplus e_2}\ \textsc{FrmOp} \qquad \frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_2}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \odot e_2}\ \textsc{FrmComp}$$

$$\frac{\langle H, \rho, \Pi \rangle \vDash_I \mathbf{acc}(\texttt{e.f}) \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e.f}\ \textsc{FrmField} \qquad \frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \mathbf{acc}(\texttt{e.f})}\ \textsc{FrmAcc}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_1 \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_2}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_1 \land \phi_2}\ \textsc{FrmAndOp}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_1 \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_2}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_1 * \phi_2}\ \textsc{FrmSepOp}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \qquad \ldots \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_n}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} p(e_1, \ldots, e_n)}\ \textsc{FrmPred}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e \qquad H, \rho \vdash e \Downarrow \texttt{true} \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_T}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F}\ \textsc{FrmCondTrue}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e \qquad H, \rho \vdash e \Downarrow \texttt{false} \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_F}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F}\ \textsc{FrmCondFalse}$$

$$\frac{\begin{array}{c}\langle H, \rho, \Pi \rangle \vDash_I p(e_1, \ldots, e_n) \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \qquad \ldots \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_n \\ \langle H, \rho, \Pi' \rangle \vdash_{\texttt{frmI}} \phi \qquad \Pi' = \Pi \cup \lfloor \mathrm{body}_\mu(p)(e_1, \ldots, e_n) \rfloor_{H,\rho}\end{array}}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \texttt{unfolding}\ p(e_1, \ldots, e_n)\ \texttt{in}\ \phi}\ \textsc{FrmUnfolding}$$

Fig. 20. $\mathrm{SVL}_{\mathrm{RP}}$: Framing

A.2.2 *Static Verification.*

$$\begin{aligned}
\mathrm{acc}(v) &= \texttt{true} \\
\mathrm{acc}(x) &= \texttt{true} \\
\mathrm{acc}(e_1 \odot e_2) &= \mathrm{acc}(e_1) \land \mathrm{acc}(e_2) \\
\mathrm{acc}(e_1 \oplus e_2) &= \mathrm{acc}(e_1) \land \mathrm{acc}(e_2) \\
\mathrm{acc}(e.f) &= \mathrm{acc}(e) \land \mathbf{acc}(\texttt{e.f})
\end{aligned}$$

Fig. 21. $\mathrm{acc}(e) : \textsc{Expr} \to \textsc{Formula}$

$$
\begin{array}{ll}
\mathsf{WLP}(\mathtt{skip}, \theta) & = \theta \\
\mathsf{WLP}(s_1 \ ; \ s_2, \theta) & = \mathsf{WLP}(s_1, \mathsf{WLP}(s_2, \theta)) \\
\mathsf{WLP}(T\ x, \theta) & = \begin{cases} \theta & \mathtt{if}\ x \notin \mathsf{FV}(\theta) \\ \mathtt{undefined} & \mathtt{otherwise} \end{cases} \\
\mathsf{WLP}(\mathtt{if}\ (e)\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \}, \theta) & = \max_{\Rightarrow} \{\ \theta' \mid \theta' \Rightarrow \mathtt{if}\ e\ \mathtt{then}\ \mathsf{WLP}(s_1, \theta)\ \mathtt{else}\ \mathsf{WLP}(s_2, \theta) \land \\
& \theta' \Rightarrow \mathsf{acc}(e)\ \} \\
\mathsf{WLP}(x := e, \theta) & = \max_{\Rightarrow} \{\ \theta' \mid \theta' \Rightarrow \theta[e/x]\ \land\ \theta' \Rightarrow \mathsf{acc}(e)\ \} \\
\mathsf{WLP}(\mathtt{while}\ (e)\ \mathtt{inv}\ \theta_i\ \{\ s\ \}, \theta) & = \max_{\Rightarrow} \{\ \theta' \mid\ \theta' \Rightarrow \mathsf{acc}(e)\ \land\ \exists \theta_f\ .\ \theta' \Rightarrow \theta_i\ *\ \theta_f\ \land\ \overline{x_i} \notin \mathsf{FV}(\theta_f) \land \\
& \theta_f\ *\ (\theta_i\ *\ (e\ =\ \mathtt{false}))[\overline{x_i/y_i}] \Rightarrow \theta[\overline{x_i/y_i}]\ \} \\
& \mathtt{where}\ \overline{y_i}\ \mathtt{are\ variables\ modified\ by\ the\ loop\ body}\ s \\
& \mathtt{and}\ \overline{x_i}\ \mathtt{are\ fresh\ logical\ variables} \\
\mathsf{WLP}(x.f := y, \theta) & = \mathbf{acc}(\mathtt{x.f})\ \land\ \theta[y/x.f] \\
\mathsf{WLP}(x := \mathsf{new}\ C, \theta) & = \max_{\Rightarrow} \{\ \theta' \mid x \notin \mathsf{FV}(\theta') \land \\
& \theta' * x \neq null * \overline{x \neq e_i} * \overline{acc(x.f_i)} * \overline{x.f_i = \mathsf{defaultValue}(T_i)} \Rightarrow \theta\ \} \\
& \mathtt{where\ fields(C)} = \overline{T_i\ f_i}\ \mathtt{and}\ x \neq e_i\ \mathtt{is\ a\ conjunctive\ term\ in}\ \theta \\
\mathsf{WLP}(y := z.m(\overline{x}), \theta) & = \max_{\Rightarrow} \{\ \theta' \mid\ y \notin \mathsf{FV}(\theta') \land \\
& \exists \theta_f\ .\ \theta' \Rightarrow (z\ \neq\ \mathtt{null})\ *\ \mathsf{mpre}(m)[z/\mathtt{this}, \overline{x/\mathsf{mparam}(m)}]\ *\ \theta_f \land \\
& \theta_f\ *\ \mathsf{mpost}(m)[z/\mathtt{this}, \overline{x/\mathbf{old}(\mathsf{mparam}(m))}, y/\mathtt{result}] \Rightarrow \theta\ \} \\
\mathsf{WLP}(\mathsf{assert}\ \phi_a, \theta) & = \max_{\Rightarrow} \{\ \theta' \mid \theta' \Rightarrow \theta\ \land\ \theta' \Rightarrow \phi_a\ \} \\
\mathsf{WLP}(\mathsf{fold}\ p(e_1, \ldots, e_n), \theta) & = \max_{\Rightarrow} \{\ \phi' \mid \phi'\ *\ p(e_1, \ldots, e_n) \Rightarrow \theta\ \land\ \phi'\ *\ p(e_1, \ldots, e_n) \in \textsc{SatFormula} \land \\
& \qquad \phi'\ *\ \mathsf{body}_\mu(p)(e_1, \ldots, e_n) \in \textsc{SfrmFormula}\ \} * \mathsf{body}_\mu(p)(e_1, \ldots, e_n) \\
& \mathtt{if\ this\ result\ exists\ and\ is\ satisfiable,\ undefined\ otherwise} \\
\mathsf{WLP}(\mathsf{unfold}\ p(e_1, \ldots, e_n), \theta) & = \max_{\Rightarrow} \{\ \phi' \mid \phi'\ *\ \mathsf{body}_\mu(p)(e_1, \ldots, e_n) \Rightarrow \theta \land \\
& \qquad \phi'\ *\ \mathsf{body}_\mu(p)(e_1, \ldots, e_n) \in \textsc{SatFormula} \land \\
& \qquad \phi'\ *\ p(e_1, \ldots, e_n) \in \textsc{SfrmFormula}\ \} * p(e_1, \ldots, e_n) \\
& \mathtt{if\ this\ result\ exists\ and\ is\ satisfiable,\ undefined\ otherwise}
\end{array}
$$

Fig. 22. SVL_{RP}: Weakest liberal precondition calculus

A.2.3 Dynamic Semantics.

$$\frac{}{\langle H, \langle \rho, \pi, \mathsf{skip}\rangle \cdot nil\rangle \quad \texttt{final}}\ \textsc{SsSkipFin} \qquad \frac{}{\langle H, \langle \rho, \pi, \mathsf{skip}\ ;\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \textsc{SsSkip}$$

$$\frac{}{\langle H, \langle \rho, \pi, T\ x\ ;\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \textsc{SsDeclare}$$

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \phi}{\langle H, \langle \rho, \pi, \mathsf{assert}\ \phi\ ;\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \textsc{SsAssert}$$

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \mathbf{acc}(\mathtt{x.f}) \qquad H, \rho \vdash y \Downarrow v \qquad H' = H[o \mapsto [f \mapsto v]]}{\langle H, \langle \rho, \pi, x.f := y\ ;\ s\rangle \cdot S\rangle \longrightarrow \langle H', \langle \rho, \pi, s\rangle \cdot S\rangle}\ \textsc{SsFAssign}$$

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow v \qquad \rho' = \rho[x \mapsto v]}{\langle H, \langle \rho, \pi, x := e\ ;\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', \pi, s\rangle \cdot S\rangle}\ \textsc{SsAssign}$$

$$\frac{o \notin \mathsf{dom}(H) \qquad \mathsf{fields}(C) = \overline{T_i\ \ f_i\,;} \qquad H' = H[o \mapsto [\overline{f_i \mapsto \mathsf{defaultValue}(T_i)}]]}{\langle H, \langle \rho, \pi, x := \mathsf{new}\ C\ ;\ s\rangle \cdot S\rangle \longrightarrow \langle H', \langle \rho[x \mapsto o], \pi \cup \overline{\langle o, f_i\rangle}, s\rangle \cdot S\rangle}\ \textsc{SsAlloc}$$

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow \mathtt{true}}{\langle H, \langle \rho, \pi, \mathtt{if}\ (e)\ \{\ s_1\ \}\ \mathsf{else}\ \{\ s_2\ \}\ ;\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, \pi, s_1\ ;\ s\rangle \cdot S\rangle}\ \textsc{SsIfTrue}$$

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow \mathtt{false}}{\langle H, \langle \rho, \pi, \mathtt{if}\ (e)\ \{\ s_1\ \}\ \mathsf{else}\ \{\ s_2\ \}\ ;\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, \pi, s_2\ ;\ s\rangle \cdot S\rangle}\ \textsc{SsIfFalse}$$

Fig. 23. $\mathrm{SVL_{RP}}$: Small-step semantics

$$\frac{\begin{array}{c}\mathsf{method}(m) = T_r\ m(\overline{T\ x'})\ \texttt{requires}\ \theta_p\ \texttt{ensures}\ \theta_q\ \{\ r\ \} \qquad H, \rho \vdash z \Downarrow o \qquad \overline{H, \rho \vdash x \Downarrow v} \\ \rho' = [\texttt{this} \mapsto o, \overline{x' \mapsto v}, \overline{\textbf{old}(x') \mapsto v}] \qquad \pi' = \langle\langle \lfloor \theta_p \rfloor_{H,\rho'} \rangle\rangle_H \qquad \pi' \subseteq \pi \qquad \langle H, \rho', \pi' \rangle \vDash_E \theta_p\end{array}}{\langle H, \langle \rho, \pi, y := z.m(\overline{x})\ ;\ s \rangle \cdot S \rangle \longrightarrow \langle H, \langle \rho', \pi', r\ ;\ \mathsf{skip} \rangle \cdot \langle \rho, \pi \backslash \pi', y := z.m(\overline{x})\ ;\ s \rangle \cdot S \rangle}\ \textsc{SsCall}$$

$$\frac{\mathsf{mpost}(m) = \theta_q \qquad \langle H, \rho', \pi' \rangle \vDash_E \theta_q \qquad \rho'' = \rho[y \mapsto \rho'(\mathsf{result})]}{\langle H, \langle \rho', \pi', \mathsf{skip} \rangle \cdot \langle \rho, \pi, y := z.m(\overline{x})\ ;\ s \rangle \cdot S \rangle \longrightarrow \langle H, \langle \rho'', \pi \cup \pi', s \rangle \cdot S \rangle}\ \textsc{SsCallFinish}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \theta_i \qquad \langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow \mathsf{false}}{\langle H, \langle \rho, \pi, \texttt{while}\ (e)\ \texttt{inv}\ \theta_i\ \{\ r\ \}\ ;\ s \rangle \cdot S \rangle \longrightarrow \langle H, \langle \rho, \pi, s \rangle \cdot S \rangle}\ \textsc{SsWhileFalse}$$

$$\frac{\begin{array}{c}\langle H, \rho, \pi \rangle \vDash_E \theta_i \qquad \langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow \mathsf{true} \\ \pi' = \langle\langle \lfloor \theta_i \rfloor_{H,\rho} \rangle\rangle_H\end{array}}{\begin{array}{c}\langle H, \langle \rho, \pi, \texttt{while}\ (e)\ \texttt{inv}\ \theta_i\ \{\ r\ \}\ ;\ s \rangle \cdot S \rangle \\ \longrightarrow \\ \langle H, \langle \rho, \pi', r\ ;\ \mathsf{skip} \rangle \cdot \langle \rho, \pi \backslash \pi', \texttt{while}\ (e)\ \texttt{inv}\ \theta_i\ \{\ r\ \}\ ;\ s \rangle \cdot S \rangle\end{array}}\ \textsc{SsWhileTrue}$$

$$\frac{}{\begin{array}{c}\langle H, \langle \rho', \pi', \mathsf{skip} \rangle \cdot \langle \rho, \pi, \texttt{while}\ (e)\ \texttt{inv}\ \theta_i\ \{\ r\ \}\ ;\ s \rangle \cdot S \rangle \\ \longrightarrow \\ \langle H, \langle \rho', \pi \cup \pi', \texttt{while}\ (e)\ \texttt{inv}\ \theta_i\ \{\ r\ \}\ ;\ s \rangle \cdot S \rangle\end{array}}\ \textsc{SsWhileFinish}$$

$$\frac{}{\langle H, \langle \rho, \pi, \mathsf{fold}\ p(e_1, \ldots, e_n)\ ;\ s \rangle \cdot S \rangle \longrightarrow \langle H, \langle \rho, \pi, s \rangle \cdot S \rangle}\ \textsc{SsFold}$$

$$\frac{}{\langle H, \langle \rho, \pi, \mathsf{unfold}\ p(e_1, \ldots, e_n)\ ;\ s \rangle \cdot S \rangle \longrightarrow \langle H, \langle \rho, \pi, s \rangle \cdot S \rangle}\ \textsc{SsUnfold}$$

Fig. 23. $\mathrm{SVL_{RP}}$: Small-step semantics (continued)

A.2.4 Weakest Precondition across stack frames. The formal statement of soundness relies on an extended definition of WLP given in Figure 24. It is used to validate arbitrary intermediate program states (Def. 5.6), and in particular, program states with multiple stack frames. sWLP accepts a stack of statements and postcondition θ and returns a stack of preconditions by recursively picking up the postconditions of methods or loop invariants of loops. sWLP relies on sWLP^{θ_f} to weaken each precondition in the stack except the top-most one. A precondition is weakened by ensuring its accessibility predicates and predicate instances are *disjoint* from those in θ_f. Effectively, θ_f represents the implicit frame of the executing method or loop, so ownership given by θ_f is withdrawn from the call site aligning with $\mathrm{SVL_{RP}}$'s runtime semantics.

For example, imagine a program state with a lower stack frame i having a WLP of `acc(x.f) * (x.f = 3)`. Assume that access to `x.f` was passed up the call stack (*i.e.* it was demanded by the preconditions of called methods or invariants of executing loops), so currently executing statements can *change* the value of `x.f`. As a result, $\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i(s_n \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathsf{true})$ is violated.

$$\begin{aligned}
&\mathsf{sWLP}(s \cdot \mathsf{nil}, \theta) = \mathsf{WLP}(s, \theta) \cdot \mathsf{nil} \\
&\mathsf{sWLP}(s \cdot (y := z.m(\overline{x})\ ;\ s') \cdot \overline{s}, \theta) = \mathsf{WLP}(s, \mathsf{mpost}(m)) \cdot \\
&\qquad\qquad \mathsf{sWLP}^{\mathsf{mpre}(m)[z/\mathsf{this}, \overline{x/\mathsf{mparam}(m)}]}((y := z.m(\overline{x})\ ;\ s') \cdot \overline{s}, \theta) \\
&\mathsf{sWLP}(s \cdot (\texttt{while}\ (e)\ \texttt{inv}\ \theta_i\ \{\ r\ \}\ ;\ s') \cdot \overline{s}, \theta) = \mathsf{WLP}(s, \theta_i) \cdot \mathsf{sWLP}^{\theta_i}((\texttt{while}\ (e)\ \texttt{inv}\ \theta_i\ \{\ r\ \}\ ;\ s') \cdot \overline{s}, \theta) \\
&\text{where}\quad \mathsf{sWLP}^{\theta_f}(\overline{s}, \theta) = \min_{\Rightarrow}\{\ \theta'_n \mid \theta_n \Rightarrow \theta_f * \theta'_n\ \} \cdot \theta_{n-1} \cdot \ldots \cdot \theta_1 \cdot \mathsf{nil} \\
&\quad\text{and}\quad \theta_n \cdot \theta_{n-1} \cdot \ldots \cdot \theta_1 \cdot \mathsf{nil} = \mathsf{sWLP}(\overline{s}, \theta)
\end{aligned}$$

Fig. 24. Heap aware weakest liberal precondition across multiple stack frames

We solve this problem by making sure that the stack frame does *not* have a WLP of **acc**(`x.f`) $*$ (`x.f = 3`) if it is currently buried under other stack frames that own `x.f`.

A.3 GVL$_{\mathbf{RP}}$

A.3.1 Framing.

$$
\begin{aligned}
&\mathsf{TotalFP}(v, H, \rho) &&= \emptyset\\
&\mathsf{TotalFP}(x, H, \rho) &&= \emptyset\\
&\mathsf{TotalFP}(e_1 \odot e_2, H, \rho) &&= \mathsf{TotalFP}(e_1, H, \rho) \cup \mathsf{TotalFP}(e_2, H, \rho)\\
&\mathsf{TotalFP}(e_1 \oplus e_2, H, \rho) &&= \mathsf{TotalFP}(e_1, H, \rho) \cup \mathsf{TotalFP}(e_2, H, \rho)\\
&\mathsf{TotalFP}(e.f, H, \rho) &&= \mathsf{TotalFP}(e, H, \rho) \cup \{\langle o, f\rangle | H, \rho \vdash e \Downarrow o\}\\
&\mathsf{TotalFP}(\mathbf{acc}(\texttt{e.f}), H, \rho) &&= \mathsf{TotalFP}(e.f, H, \rho)\\
&\mathsf{TotalFP}(\phi_1 \wedge \phi_2, H, \rho) &&= \mathsf{TotalFP}(\phi_1, H, \rho) \cup \mathsf{TotalFP}(\phi_2, H, \rho)\\
&\mathsf{TotalFP}(\phi_1 * \phi_2, H, \rho) &&= \mathsf{TotalFP}(\phi_1, H, \rho) \cup \mathsf{TotalFP}(\phi_2, H, \rho)\\
&\mathsf{TotalFP}(p(e_1, \ldots, e_n), H, \rho) &&= \mathsf{TotalFP}(e_1, H, \rho) \cup \ldots \cup \mathsf{TotalFP}(e_n, H, \rho) \cup\\
& &&\quad \{\langle p, v_1, \ldots, v_n\rangle | H, \rho \vdash e_1 \Downarrow v_1, \ldots, H, \rho \vdash e_n \Downarrow v_n\}\\
&\mathsf{TotalFP}(\texttt{if } e \texttt{ then } \phi_1 \texttt{ else } \phi_2, H, \rho) &&= \begin{cases} \mathsf{TotalFP}(e, H, \rho) \cup \mathsf{TotalFP}(\phi_1, H, \rho) & \texttt{if } H, \rho \vdash e \Downarrow \texttt{true}\\ \mathsf{TotalFP}(e, H, \rho) \cup \mathsf{TotalFP}(\phi_2, H, \rho) & \texttt{if } H, \rho \vdash e \Downarrow \texttt{false}\\ \emptyset & \texttt{otherwise} \end{cases}\\
&\mathsf{TotalFP}(\texttt{unfolding } p(e_1, \ldots, e_n) \texttt{ in } \phi, H, \rho) &&= \mathsf{TotalFP}(p(e_1, \ldots, e_n), H, \rho) \cup \mathsf{TotalFP}(\phi, H, \rho)
\end{aligned}
$$

Fig. 25. Definition of the TotalFP function.

A.3.2 Lifting functions.

$$\widetilde{\mathsf{WLP}}(s_1\ ;\ s_2, \widetilde{\phi}) = \widetilde{\mathsf{WLP}}(s_1, \widetilde{\mathsf{WLP}}(s_2, \widetilde{\phi}))$$

$$\begin{aligned}\widetilde{\mathsf{WLP}}(\texttt{if }(e)\ \{\ s_1\ \}\texttt{ else }\{\ s_2\ \}, \widetilde{\phi}) = \alpha(\{\ &\max_{\Rightarrow}\ \{\ \phi' \in \textsc{SatFormula} \mid \phi' \Rightarrow \texttt{if}\ e\ \texttt{then}\ \theta_1\ \texttt{else}\ \theta_2\ \land \\ &\phi' \Rightarrow \mathsf{acc}(e)\ \land\ \vdash_{\texttt{frm}} \langle \phi', \mathsf{body}_\Delta{}' \rangle\ \}\mid \theta_1 \in \gamma(\widetilde{\mathsf{WLP}}(s_1, \widetilde{\phi})), \theta_2 \in \gamma(\widetilde{\mathsf{WLP}}(s_2, \widetilde{\phi})), \\ &\mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta_1, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_2, \mathsf{body}_\Delta{}' \rangle\ \})\end{aligned}$$

$$\begin{aligned}\widetilde{\mathsf{WLP}}(y := z\,.\,m(\overline{x}), \widetilde{\phi}) = \alpha(\{\ &\max_{\Rightarrow}\ \{\ \phi' \in \textsc{SatFormula} \mid y \notin \mathsf{FV}(\phi')\ \land\ \vdash_{\texttt{frm}} \langle \phi', \mathsf{body}_\Delta{}' \rangle\ \land \\ &\exists \phi_f\ .\ \phi' \Rightarrow (z\ \neq\ \texttt{null})\ *\ \theta_p[z/\texttt{this}, \overline{x/\mathsf{mparam}(m)}]\ *\ \phi_f\ \land \\ &\phi_f\ *\ \theta_q[z/\texttt{this}, \overline{x/\textbf{old}(\mathsf{mparam}(m))}, y/\texttt{result}] \Rightarrow \theta\ \land\ \vdash_{\texttt{frm}} \langle \phi_f, \mathsf{body}_\Delta{}' \rangle\ \} \\ &\mid \theta \in \gamma(\widetilde{\phi}),\ \theta_p \in \gamma(\mathsf{mpre}(m)),\ \theta_q \in \gamma(\mathsf{mpost}(m)),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu), \\ &\vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_p, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_q, \mathsf{body}_\Delta{}' \rangle\ \})\end{aligned}$$

$$\begin{aligned}\widetilde{\mathsf{WLP}}(\texttt{while }(e)\texttt{ inv }\widetilde{\phi_i}\ \{\ s\ \}, \widetilde{\phi}) = \alpha(\{\ &\max_{\Rightarrow}\ \{\ \phi' \in \textsc{SatFormula} \mid \phi' \Rightarrow \mathsf{acc}(e)\ \land\ \vdash_{\texttt{frm}} \langle \phi', \mathsf{body}_\Delta{}' \rangle\ \land \\ &\exists \phi_f\ .\ \phi' \Rightarrow \theta_i\ *\ \phi_f\ \land\ \overline{x_i} \notin \mathsf{FV}(\phi_f)\ \land\ \vdash_{\texttt{frm}} \langle \phi_f, \mathsf{body}_\Delta{}' \rangle\ \land \\ &\phi_f\ *\ (\theta_i\ *\ (e\ =\ \texttt{false}))[\overline{x_i/y_i}] \Rightarrow \theta[\overline{x_i/y_i}]\ \} \\ &\mid \theta \in \gamma(\widetilde{\phi}),\ \theta_i \in \gamma(\widetilde{\phi_i}),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_i, \mathsf{body}_\Delta{}' \rangle\ \}) \\ &\texttt{where } \overline{y_i} \texttt{ are vars modified by the loop body } s \texttt{ and } \overline{x_i} \texttt{ are fresh}\end{aligned}$$

$$\begin{aligned}\widetilde{\mathsf{WLP}}(\texttt{fold}\ p(\overline{e}), \widetilde{\phi}) = \alpha(\{\ &\max_{\Rightarrow}\ \{\ \phi' \in \textsc{SatFormula} \mid \phi'\ *\ p(\overline{e}) \Rightarrow \theta\ \land\ \phi'\ *\ p(\overline{e}) \in \textsc{SatFormula}\ \land \\ &\vdash_{\texttt{frm}} \langle \phi' * \mathsf{body}_\Delta{}'(p)(\overline{e}), \mathsf{body}_\Delta{}' \rangle\ \} * \mathsf{body}_\Delta{}'(p)(\overline{e}) \in \textsc{SatFormula} \\ &\mid \theta \in \gamma(\widetilde{\phi}),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle\ \})\end{aligned}$$

$$\begin{aligned}\widetilde{\mathsf{WLP}}(\texttt{unfold}\ p(\overline{e}), \widetilde{\phi}) = \alpha(\{\ &\max_{\Rightarrow}\ \{\ \phi' \in \textsc{SatFormula} \mid \phi'\ *\ \mathsf{body}_\Delta{}'(p)(\overline{e}) \Rightarrow \theta\ \land \\ &\phi'\ *\ \mathsf{body}_\Delta{}'(p)(\overline{e}) \in \textsc{SatFormula}\ \land\ \vdash_{\texttt{frm}} \langle \phi' * p(\overline{e}), \mathsf{body}_\Delta{}' \rangle\ \} * p(\overline{e}) \in \textsc{SatFormula} \\ &\mid \theta \in \gamma(\widetilde{\phi}),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle\ \})\end{aligned}$$

$$\widetilde{\mathsf{WLP}}(s, \widetilde{\phi}) = \alpha(\{\ \mathsf{WLP}(s, \theta, \mathsf{body}_\Delta{}') \mid \theta \in \gamma(\widetilde{\phi}),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle\ \}) \quad \text{otherwise}$$

Fig. 26. $\mathrm{GVL_{RP}}$: Weakest liberal precondition calculus.

A.3.3 Dynamic semantics.

$$\frac{}{\langle H, \langle \rho, \pi, \mathsf{skip}\rangle \cdot nil\rangle \quad \mathtt{final}}\ \text{SsSkipFin} \qquad \frac{}{\langle H, \langle \rho, \pi, \mathsf{skip}\ ;\ s\rangle \cdot S\rangle \xrightarrow{\sim} \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \text{SsSkip}$$

$$\frac{}{\langle H, \langle \rho, \pi, T\ x\ ;\ s\rangle \cdot S\rangle \xrightarrow{\sim} \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \text{SsDeclare}$$

$$\frac{\langle H, \rho, \pi\rangle \mathrel{\widetilde{\vDash}} \langle ?\ *\ \phi, \mathsf{body}_\mu\rangle}{\langle H, \langle \rho, \pi, \mathsf{assert}\ \phi\ ;\ s\rangle \cdot S\rangle \xrightarrow{\sim} \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \text{SsAssert}$$

$$\frac{\langle H, \rho, \pi\rangle \mathrel{\widetilde{\nvDash}} \langle ?\ *\ \phi, \mathsf{body}_\mu\rangle}{\langle H, \langle \rho, \pi, \mathsf{assert}\ \phi\ ;\ s\rangle \cdot S\rangle \xrightarrow{\sim} \mathbf{error}}\ \text{SsAssertError}$$

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \mathbf{acc}(\mathtt{x.f}) \qquad H, \rho \vdash y \Downarrow v \qquad H' = H[o \mapsto [f \mapsto v]]}{\langle H, \langle \rho, \pi, x.f\ := y\ ;\ s\rangle \cdot S\rangle \xrightarrow{\sim} \langle H', \langle \rho, \pi, s\rangle \cdot S\rangle}\ \text{SsFAssign}$$

$$\frac{\langle H, \rho, \pi\rangle \nvDash_E \mathbf{acc}(\mathtt{x.f})}{\langle H, \langle \rho, \pi, x.f\ := y\ ;\ s\rangle \cdot S\rangle \xrightarrow{\sim} \mathbf{error}}\ \text{SsFAssignError}$$

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow v \qquad \rho' = \rho[x \mapsto v]}{\langle H, \langle \rho, \pi, x\ := e\ ;\ s\rangle \cdot S\rangle \xrightarrow{\sim} \langle H, \langle \rho', \pi, s\rangle \cdot S\rangle}\ \text{SsAssign}$$

$$\frac{\langle H, \rho, \pi\rangle \nvDash_E \mathsf{acc}(e)}{\langle H, \langle \rho, \pi, x\ := e\ ;\ s\rangle \cdot S\rangle \xrightarrow{\sim} \mathbf{error}}\ \text{SsAssignError}$$

$$\frac{o \notin \mathsf{dom}(H) \qquad \mathsf{fields}(C) = \overline{T_i\ \ f_i\,;} \qquad H' = H[o \mapsto [\overline{f_i \mapsto \mathsf{defaultValue}(T_i)}]]}{\langle H, \langle \rho, \pi, x\ := \mathsf{new}\ C\ ;\ s\rangle \cdot S\rangle \xrightarrow{\sim} \langle H', \langle \rho[x \mapsto o], \pi \cup \overline{\langle o, f_i\rangle}, s\rangle \cdot S\rangle}\ \text{SsAlloc}$$

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow \mathtt{true}}{\langle H, \langle \rho, \pi, \mathtt{if}\ (e)\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \}\ ;\ s\rangle \cdot S\rangle \xrightarrow{\sim} \langle H, \langle \rho, \pi, s_1\ ;\ s\rangle \cdot S\rangle}\ \text{SsIfTrue}$$

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow \mathtt{false}}{\langle H, \langle \rho, \pi, \mathtt{if}\ (e)\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \}\ ;\ s\rangle \cdot S\rangle \xrightarrow{\sim} \langle H, \langle \rho, \pi, s_2\ ;\ s\rangle \cdot S\rangle}\ \text{SsIfFalse}$$

$$\frac{\langle H, \rho, \pi\rangle \nvDash_E \mathsf{acc}(e)}{\langle H, \langle \rho, \pi, \mathtt{if}\ (e)\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \}\ ;\ s\rangle \cdot S\rangle \xrightarrow{\sim} \mathbf{error}}\ \text{SsIfError}$$

Fig. 27. GVL_{RP}: Small-step semantics adjusted from Fig. 23 for gradual formulas

$$\frac{\begin{array}{c}\text{method}(m) = T_r\ m(\overline{T\ x'})\ \texttt{requires}\ \widetilde{\phi}_p\ \texttt{ensures}\ \widetilde{\phi}_q\ \{\ r\ \} \qquad H, \rho \vdash z \Downarrow o \qquad \overline{H, \rho \vdash x \Downarrow v} \\ \rho' = [\texttt{this} \mapsto o, \overline{x' \mapsto v}, \overline{\textbf{old}(x') \mapsto v}] \qquad \pi' = \lfloor \widetilde{\phi}_p \rfloor_{\pi, H, \rho'} \qquad \pi' \subseteq \pi \qquad \langle H, \rho', \pi' \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_p, \text{body}_\mu \rangle\end{array}}{\langle H, \langle \rho, \pi, y := z.m(\overline{x})\ ;\ s \rangle \cdot S \rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho', \pi', r\ ;\ \texttt{skip} \rangle \cdot \langle \rho, \pi \backslash \pi', y := z.m(\overline{x})\ ;\ s \rangle \cdot S \rangle}\ \textsc{SsCall}$$

$$\frac{\langle H, \rho, \pi \rangle \mathrel{\widetilde{\nvDash}} \langle \text{mpre}(m)[z/\texttt{this}, \overline{x/\text{mparam}(m)}], \text{body}_\mu \rangle}{\langle H, \langle \rho, \pi, y := z.m(\overline{x})\ ;\ s \rangle \cdot S \rangle \mathrel{\widetilde{\longrightarrow}} \textbf{error}}\ \textsc{SsCallError}$$

$$\frac{\text{mpost}(m) = \widetilde{\phi}_q \qquad \langle H, \rho', \pi' \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_q, \text{body}_\mu \rangle \qquad \rho'' = \rho[y \mapsto \rho'(\texttt{result})]}{\langle H, \langle \rho', \pi', \texttt{skip} \rangle \cdot \langle \rho, \pi, y := z.m(\overline{x})\ ;\ s \rangle \cdot S \rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho'', \pi \cup \pi', s \rangle \cdot S \rangle}\ \textsc{SsCallFinish}$$

$$\frac{\langle H, \rho', \pi' \rangle \mathrel{\widetilde{\nvDash}} \langle \text{mpost}(m), \text{body}_\mu \rangle}{\langle H, \langle \rho', \pi', \texttt{skip} \rangle \cdot \langle \rho, \pi, y := z.m(\overline{x})\ ;\ s \rangle \cdot S \rangle \mathrel{\widetilde{\longrightarrow}} \textbf{error}}\ \textsc{SsCallFinishError}$$

$$\frac{\langle H, \rho, \pi \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_i, \text{body}_\mu \rangle \qquad \langle H, \rho, \pi \rangle \vDash_E \text{acc}(e) \qquad H, \rho \vdash e \Downarrow \texttt{false}}{\langle H, \langle \rho, \pi, \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \}\ ;\ s \rangle \cdot S \rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho, \pi, s \rangle \cdot S \rangle}\ \textsc{SsWhileFalse}$$

$$\frac{\begin{array}{c}\langle H, \rho, \pi \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_i, \text{body}_\mu \rangle \qquad \langle H, \rho, \pi \rangle \vDash_E \text{acc}(e) \qquad H, \rho \vdash e \Downarrow \texttt{true} \\ \pi' = \lfloor \widetilde{\phi}_i \rfloor_{\pi, H, \rho}\end{array}}{\begin{array}{c}\langle H, \langle \rho, \pi, \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \}\ ;\ s \rangle \cdot S \rangle \\ \widetilde{\longrightarrow} \\ \langle H, \langle \rho, \pi', r\ ;\ \texttt{skip} \rangle \cdot \langle \rho, \pi \backslash \pi', \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \}\ ;\ s \rangle \cdot S \rangle\end{array}}\ \textsc{SsWhileTrue}$$

$$\frac{\langle H, \rho, \pi \rangle \mathrel{\widetilde{\nvDash}} \langle \widetilde{\phi}_i \wedge \text{acc}(e), \text{body}_\mu \rangle}{\langle H, \langle \rho, \pi, \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \}\ ;\ s \rangle \cdot S \rangle \mathrel{\widetilde{\longrightarrow}} \textbf{error}}\ \textsc{SsWhileError}$$

$$\frac{}{\begin{array}{c}\langle H, \langle \rho', \pi', \texttt{skip} \rangle \cdot \langle \rho, \pi, \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \}\ ;\ s \rangle \cdot S \rangle \\ \widetilde{\longrightarrow} \\ \langle H, \langle \rho', \pi \cup \pi', \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \}\ ;\ s \rangle \cdot S \rangle\end{array}}\ \textsc{SsWhileFinish}$$

$$\frac{}{\langle H, \langle \rho, \pi, \texttt{fold}\ p(e_1, \ldots, e_n)\ ;\ s \rangle \cdot S \rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho, \pi, s \rangle \cdot S \rangle}\ \textsc{SsFold}$$

$$\frac{}{\langle H, \langle \rho, \pi, \texttt{unfold}\ p(e_1, \ldots, e_n)\ ;\ s \rangle \cdot S \rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho, \pi, s \rangle \cdot S \rangle}\ \textsc{SsUnfold}$$

Fig. 27. GVL_{RP}: Small-step semantics adjusted from Fig. 23 for gradual formulas (continued)

A.4 Proofs

1 Cross cutting lemmas

Lemma 1 (Equi Permissions Supersets & Formula Evaluation). *If $\pi \subseteq \pi'$ and $\langle H, \rho, \pi \rangle \vDash_E \phi$, then $\langle H, \rho, \pi' \rangle \vDash_E \phi$.*

Proof. Falls out easily by structural induction on the derivation of $\langle H, \rho, \pi \rangle \vDash_E \phi$.

Lemma 2 (Iso Permissions Supersets & Formula Evaluation). *If $\Pi \subseteq \Pi'$ and $\langle H, \rho, \Pi \rangle \vDash_I \phi$, then $\langle H, \rho, \Pi' \rangle \vDash_I \phi$.*

Proof. Falls out easily by structural induction on the derivation of $\langle H, \rho, \Pi \rangle \vDash_I \phi$.

Lemma 3 (Sequence Stmt Rearrangement). *If $s = s_1\,;\ s_2$, then $s = s_h\,;\ s_t$ such that s_h is not a sequence statement and $s_t = s_2$ or $s_{t_1}\,;\ s_2$ where $s_1 = s_h\,;\ s_{t_1}$.*

Proof. Suppose $s = s_1\,;\ s_2$.

We will show by induction on the syntax of s_1 that $s = s_h\,;\ s_t$ such that s_h is not a sequence statement and $s_t = s_2$ or $s_{t_1}\,;\ s_2$ where $s_1 = s_h\,;\ s_{t_1}$.

Case 1 (s_1 is not a sequence statement). Then $s = s_1\,;\ s_2$ such that s_1 is not a sequence statement and the tail statement is s_2.

Case 2 ($s_1 = s_1'\,;\ s_1''$). Then $s = s_1'\,;\ (s_1''\,;\ s_2)$.

Then by the IH on s_1', we get that $s = s_h\,;\ s_t$ such that s_h is not a sequence statement and $s_t = (s_1''\,;\ s_2)$ or $s_{t_1}\,;\ (s_1''\,;\ s_2)$ where $s_1' = s_h\,;\ s_{t_1}$.

If $s_t = (s_1''\,;\ s_2)$, then

$s_1'\,;\ (s_1''\,;\ s_2) = s = s_h\,;\ (s_1''\,;\ s_2)$, and so $s_h = s_1'$.

Therefore, $s = s_h\,;\ s_t$ such that s_h is not a sequence statement and $s_t = s_1''\,;\ s_2$ where $s_1 = s_h\,;\ s_1''$ $(s_{t_1} = s_1'')$.

If $s_t = s_{t_1}\,;\ (s_1''\,;\ s_2)$ where $s_1' = s_h\,;\ s_{t_1}$, then

$s_1 = s_1'\,;\ s_1'' = s_h\,;\ s_{t_1}\,;\ s_1'' = s_h\,;\ (s_{t_1}\,;\ s_1'')$.

Therefore, $s = s_h\,;\ s_t$ such that s_h is not a sequence statement and $s_t = (s_{t_1}\,;\ s_1'')\,;\ s_2$ where $s_1 = s_h\,;\ (s_{t_1}\,;\ s_1'')$.

Lemma 4 (Permission Erasure Subset Preservation). *If $\Pi \subseteq \Pi'$ and $\langle\langle\ \Pi'\ \rangle\rangle_H$ is defined, then $\langle\langle\ \Pi\ \rangle\rangle_H$ is defined and $\langle\langle\ \Pi\ \rangle\rangle_H \subseteq \langle\langle\ \Pi'\ \rangle\rangle_H$.*

Proof. The proof of this lemma can be found here: extra-proofs/extra-proofs.pdf.

2 SVL$_{\text{RP}}$ Soundness

2.1 Progress

Claim (SVL$_{RP}$ Progress). If $\psi \in \textsc{State}$ is a valid state and
$\psi \notin \{\langle H, \langle \rho, \pi, skip \rangle \cdot nil \rangle \mid H \in \textsc{Heap}, \rho \in \textsc{Env}, \pi \in \textsc{DynFprint}\}$
then $\psi \longrightarrow \psi'$ for some $\psi' \in \textsc{State}$.

Proof. Suppose $\psi = \langle H, \langle \rho_n, \pi_n, s_n \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \in \textsc{State}$ such that ψ is a valid state and $\psi \notin \{\langle H, \langle \rho, \pi, skip \rangle \cdot nil \rangle \mid H \in \textsc{Heap}, \rho \in \textsc{Env}, \pi \in \textsc{DynFprint}\}$.

Since ψ is a valid state, by definition, we get that $s_n = s_{n_1}$; $\texttt{skip}$ for some $s_{n_1} \in \textsc{Stmt}$ or $s_n = \texttt{skip}$.

If $s_n = \texttt{skip}$, then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip} \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

If $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip} \rangle \cdot \langle \rho_{n-1}, \pi_{n-1}, y := z.m(\overline{x})\ ;\ s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$, then

Since ψ is a valid state, by definition, we get that $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{sWLP}_n(\texttt{skip} \cdot (y := z.m(\overline{x})\ ;\ s) \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$.

Also, by the definition of sWLP, we get that $\mathsf{sWLP}_n(\texttt{skip} \cdot (y := z.m(\overline{x})\ ;\ s) \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \mathsf{mpost}(m)$. Then by SsCallFinish,

$\psi \longrightarrow \langle H, \langle \rho_{n-1}[y \mapsto \rho_n(\texttt{result})], \pi_n \cup \pi_{n-1}, s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

If $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip} \rangle \cdot \langle \rho_{n-1}, \pi_{n-1}, \texttt{while (e) inv } \theta \ \{ \texttt{ s}' \ \};\ s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$, then by SsWhileFinish,

$\psi \longrightarrow \langle H, \langle \rho_n, \pi_n \cup \pi_{n-1}, \texttt{while (e) inv } \theta \ \{ \texttt{ s}' \ \};\ s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

If $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip} \rangle \cdot \langle \rho_{n-1}, \pi_{n-1}, s_{n-1} \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$ and $n > 1$ where $s_{n-1} \neq y := z.m(\overline{x})\ ;\ s$ and $s_{n-1} \neq \texttt{while (e) inv } \theta \ \{ \texttt{ s}' \ \};\ s$, then we have a contradiction because ψ being a valid state gives that $\mathsf{sWLP}(\texttt{skip} \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is defined, but according to the definition of sWLP, $\mathsf{sWLP}(\texttt{skip} \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is not defined since $s_{n-1} \neq y := z.m(\overline{x})\ ;\ s$, $s_{n-1} \neq \texttt{while (e) inv } \theta \ \{ \texttt{ s}' \ \};\ s$, and $n > 1$.

If $\psi = \langle H, \langle \rho_n, \pi_n, skip \rangle \cdot nil \rangle$, then contradiction because we assumed $\psi \notin \{\langle H', \langle \rho', \pi', skip \rangle \cdot nil \rangle \mid H' \in \textsc{Heap}, \rho' \in \textsc{Env}, \pi' \in \textsc{DynFprint}\}$.

If $s_n = s_{n_1}$; $\texttt{skip}$ for some $s_{n_1} \in \textsc{Stmt}$, then by lemma 3, we get $s_n = s_h\ ;\ s_t$ such that s_h is not a sequence statement and $s_t = \texttt{skip}$ or s_{t_1} ; $\texttt{skip}$ where $s_{n_1} = s_h\ ;\ s_{t_1}$.

We will show $\psi \longrightarrow \psi'$ for some $\psi' \in \textsc{State}$ by casing on s_h.

Case 3 ($s_h = \texttt{skip}$). Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip};\ s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$ and by SsSkip, $\langle H, \langle \rho_n, \pi_n, \texttt{skip};\ s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

Case 4 ($s_h = \texttt{T x}$). Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{T x};\ s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$ and by SsDeclare, $\langle H, \langle \rho_n, \pi_n, \texttt{T x};\ s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

Case 5 ($s_h = \texttt{assert } \phi$). Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{assert } \phi;\ s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

Since ψ is a valid state, by definition we get that $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{sWLP}_n((\texttt{assert } \phi;\ s_t) \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$.

Also, by the definition of sWLP, we get that $\mathsf{sWLP}_n((\texttt{assert } \phi;\ s_t) \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) \Rightarrow \phi$. Then lemma 6 gives $\langle H, \rho_n, \pi_n \rangle \vDash_E \phi$.

Thus, by SsAssert, $\langle H, \langle \rho_n, \pi_n, \texttt{assert } \phi;\ s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

Case 6 ($s_h = \texttt{x.f := y}$). Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{x.f := y};\ s_t\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$.

Since ψ is a valid state, by definition we get that $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{sWLP}_n((\texttt{x.f := y};\ s_t)\cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$.

Also, by the definition of sWLP, we get that $\mathsf{sWLP}_n((\texttt{x.f := y};\ s_t) \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) \Rightarrow \mathbf{acc}(\texttt{x.f})$. Then lemma 6 gives $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathbf{acc}(\texttt{x.f})$. Additionally, $H, \rho_n \vdash y \Downarrow v = \rho_n(y)$ by EVar, since $y \in \mathsf{dom}(\rho_n)$ is given by the fact that we are operating over well-typed programs .

Thus, by SsFAssign, $\langle H, \langle \rho_n, \pi_n, \texttt{x.f := y};\ s_t\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \longrightarrow$
$\langle H[o \mapsto [f \mapsto v]], \langle \rho_n, \pi_n, s_t\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$.

Case 7 ($s_h = x := e$). Then $\psi = \langle H, \langle \rho_n, \pi_n, x := e;\ s_t\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$.

Since ψ is a valid state, by definition we get that $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{sWLP}_n((x := e;\ s_t)\cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$.

Also, by the definition of sWLP, we get that $\mathsf{sWLP}_n((x := e;\ s_t) \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) \Rightarrow \mathsf{acc}(e)$. Then lemma 6 gives $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{acc}(e)$.

We are operating over well-typed programs, so $H, \rho_n \vdash e \Downarrow v$.

Thus, by SsAssign, $\langle H, \langle \rho_n, \pi_n, x := e;\ s_t\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \longrightarrow$
$\langle H, \langle \rho_n[x \mapsto v], \pi_n, s_t\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$.

Case 8 ($s_h = \texttt{x := new C}$). Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{x := new C};\ s_t\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$.

Let $o \in \textsc{Loc}$ and $o \notin \mathsf{dom}(H)$.

Thus, by SsAlloc, $\langle H, \langle \rho_n, \pi_n, \texttt{x := new C};\ s_t\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \longrightarrow$
$\langle H', \langle \rho_n[x \mapsto o], \pi_n \cup \overline{\langle o, f_i\rangle}, s_t\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$ where $\mathsf{fields}(C) = \overline{T_i\ \ f_i};$ and $H' = H[o \mapsto [\overline{f_i \mapsto \mathsf{defaultValue}(T_i)}]]$.

Case 9 ($s_h = \texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}}$). Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}};\ s_t\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$.

Since ψ is a valid state, by definition we get that $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{sWLP}_n((\texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}};\ s_t)\cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$.

Also, by the definition of sWLP, we get that $\mathsf{sWLP}_n((\texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}};\ s_t) \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) \Rightarrow \mathsf{acc}(e)$ and $\mathsf{sWLP}_n((\texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}};\ s_t) \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) \Rightarrow$
$(\texttt{if}\ e\ \texttt{then}\ \mathsf{WLP}(s'_1, \mathsf{WLP}(s_t, \theta))\ \texttt{else}\ \mathsf{WLP}(s'_2, \mathsf{WLP}(s_t, \theta)))$ for

$$\theta = \begin{cases} \mathsf{mpost}(m) & \texttt{if}\ s_{n-1} = y := z.m(\overline{x});\ s''_2 \wedge n > 1 \\ \theta_i & \texttt{if}\ s_{n-1} = \texttt{while (e) inv}\ \theta_i\ \texttt{\{ r \}};\ s''_2 \wedge n > 1 \\ \texttt{true} & \texttt{if}\ n = 1 \end{cases}$$

(all cases covered by definition of sWLP and since $\mathsf{sWLP}((\texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}};\ s_t)\cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is defined).

Then lemma 6 gives $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{acc}(e)$ and
$\langle H, \rho_n, \pi_n\rangle \vDash_E (\texttt{if}\ e\ \texttt{then}\ \mathsf{WLP}(s'_1, \mathsf{WLP}(s_t, \theta))\ \texttt{else}\ \mathsf{WLP}(s'_2, \mathsf{WLP}(s_t, \theta)))$.

By inversion on $\langle H, \rho_n, \pi_n\rangle \vDash_E (\texttt{if}\ e\ \texttt{then}\ \mathsf{WLP}(s'_1, \mathsf{WLP}(s_t, \theta))\ \texttt{else}\ \mathsf{WLP}(s'_2, \mathsf{WLP}(s_t, \theta)))$, either $H, \rho_n \vdash e \Downarrow \texttt{true}$ and $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{WLP}(s'_1, \mathsf{WLP}(s_t, \theta))$ or $H, \rho_n \vdash e \Downarrow \texttt{false}$ and $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{WLP}(s'_2, \mathsf{WLP}(s_t, \theta))$.

If $H, \rho_n \vdash e \Downarrow \texttt{true}$ and $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{WLP}(s'_1, \mathsf{WLP}(s_t, \theta))$, then by SsIfTrue,
$\langle H, \langle \rho_n, \pi_n, \texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}};\ s_t\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \longrightarrow$

$\langle H, \langle \rho_n, \pi_n, s'_1 \,;\, s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

If $H, \rho_n \vdash e \Downarrow \texttt{false}$ and $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{WLP}(s'_2, \mathsf{WLP}(s_t, \theta))$, then by SsIf-False,

$\langle H, \langle \rho_n, \pi_n, \texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}} \,;\, s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \longrightarrow$
$\langle H, \langle \rho_n, \pi_n, s'_2 \,;\, s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

Case 10 $(s_h = y := z\,.\,m(\overline{x}))$. Then $\psi = \langle H, \langle \rho_n, \pi_n, y := z\,.\,m(\overline{x}) \,;\, s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$

Since ψ is a valid state, by definition we get that
$\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{sWLP}_n((y := z\,.\,m(\overline{x}) \,;\, s_t) \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$.

Then, by the definition of sWLP, we get that
$\mathsf{sWLP}_n((y := z\,.\,m(\overline{x}) \,;\, s_t) \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) \Rightarrow \mathsf{mpre}(m)[z/\texttt{this}, \overline{x/\mathsf{mparam}(m)}]$ and
$\mathsf{sWLP}_n((y := z\,.\,m(\overline{x}) \,;\, s_t) \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) \Rightarrow z \neq \mathit{null}$. Then lemma 6 gives
$\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{mpre}(m)[z/\texttt{this}, \overline{x/\mathsf{mparam}(m)}]$ and $\langle H, \rho_n, \pi_n \rangle \vDash_E z \neq \mathit{null}$.

Then $\mathsf{mparam}(m)$ and $\mathsf{mpre}(m)$ are defined for m, which implies $\mathsf{method}(m)$ and all related look-up functions are also defined for m.

We are operating over well-typed programs so we get $H, \rho_n \vdash z \Downarrow o$ and $\overline{H, \rho_n \vdash x \Downarrow v}$

.

Since $\rho_n(z) = o$, $\overline{\rho_n(x) = v}$, $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{mpre}(m)[z/\texttt{this}, \overline{x/\mathsf{mparam}(m)}]$, and a method's precondition can only mention variables, $\texttt{this}$ and $\overline{\mathsf{mparam}(m)}$, we get $\langle H, \rho'_n = [\texttt{this} \mapsto o, \overline{\mathsf{mparam}(m) \mapsto v}, \overline{\texttt{old}(\mathsf{mparam}(m)) \mapsto v}], \pi_n \rangle \vDash_E \mathsf{mpre}(m)$.

Since $\langle H, \rho'_n, \pi_n \rangle \vDash_E \mathsf{mpre}(m)$, by lemma 8, we get $\exists\, \Pi\,.\, \langle H, \rho'_n, \Pi \rangle$ is a good iso-state, $\langle\langle\, \Pi \,\rangle\rangle_H = \pi_n$ and $\langle H, \rho'_n, \Pi \rangle \vDash_I \mathsf{mpre}(m)$.

Then, by definition, $\lfloor \mathsf{mpre}(m) \rfloor_{H, \rho'_n}$ is defined and $\lfloor \mathsf{mpre}(m) \rfloor_{H, \rho'_n} \subseteq \Pi$. And so, by lemma 4, we get that $\langle\langle\, \lfloor \mathsf{mpre}(m) \rfloor_{H, \rho'_n} \,\rangle\rangle_H$ is defined and $\pi'_n = \langle\langle\, \lfloor \mathsf{mpre}(m) \rfloor_{H, \rho'_n} \,\rangle\rangle_H \subseteq \langle\langle\, \Pi \,\rangle\rangle_H = \pi_n$. Finally, by lemma 5, we get $\langle H, \rho'_n, \pi'_n \rangle \vDash_E \mathsf{mpre}(m)$.

Then by SsCall,

$\psi \longrightarrow \langle H, \langle \rho'_n, \pi'_n, \mathsf{mbody}(m) ; \texttt{skip} \rangle \cdot \langle \rho_n, \pi_n \backslash \pi'_n, y := z\,.\,m(\overline{x}) \,;\, s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

Case 11 $(s_h = \texttt{while (e) inv}\ \theta\ \texttt{\{ s' \}})$. Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{while (e) inv}\ \theta\ \texttt{\{ s' \}} \,;\, s_t \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$

Since ψ is a valid state, by definition we get that
$\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{sWLP}_n((\texttt{while (e) inv}\ \theta\ \texttt{\{ s' \}} \,;\, s_t) \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$.

Then, by the definition of sWLP, we get that
$\mathsf{sWLP}_n((\texttt{while (e) inv}\ \theta\ \texttt{\{ s' \}} \,;\, s_t) \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) \Rightarrow \theta$ and
$\mathsf{sWLP}_n((\texttt{while (e) inv}\ \theta\ \texttt{\{ s' \}} \,;\, s_t) \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) \Rightarrow \mathsf{acc}(e)$. Then lemma 6 gives $\langle H, \rho_n, \pi_n \rangle \vDash_E \theta$ and $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$.

We are operating over well-typed programs, so we get $H, \rho_n \vdash e \Downarrow \texttt{true}$ or $H, \rho_n \vdash e \Downarrow \texttt{false}$.

If $H, \rho_n \vdash e \Downarrow \texttt{true}$, then

since $\langle H, \rho_n, \pi_n \rangle \vDash_E \theta$, by lemma 8, we get $\exists\, \Pi\,.\, \langle H, \rho_n, \Pi \rangle$ is a good iso-state, $\langle\langle\, \Pi \,\rangle\rangle_H = \pi_n$ and $\langle H, \rho_n, \Pi \rangle \vDash_I \theta$.

Then, by definition, $\lfloor\theta\rfloor_{H,\rho_n}$ is defined and $\lfloor\theta\rfloor_{H,\rho_n} \subseteq \Pi$. And so, by lemma 4, we get that $\langle\langle\, \lfloor\theta\rfloor_{H,\rho_n} \rangle\rangle_H$ is defined and $\pi'_n = \langle\langle\, \lfloor\theta\rfloor_{H,\rho_n} \rangle\rangle_H \subseteq \langle\langle\, \Pi \,\rangle\rangle_H = \pi_n$.

Then by SsWhileTrue,

$\psi \longrightarrow \langle H, \langle \rho_n, \pi'_n, s' \,;\, \texttt{skip}\rangle \cdot \langle \rho_n, \pi_n \setminus \pi'_n, \texttt{while (e) inv}\ \theta\ \{\ \texttt{s}'\ \};\ s_t\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$.

If $H, \rho_n \vdash e \Downarrow \texttt{false}$, then

by SsWhileFalse,

$\psi \longrightarrow \langle H, \langle \rho_n, \pi_n, s_t\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$.

Case 12 ($s_h = \texttt{fold}\ p(\overline{e})$). Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{fold}\ p(\overline{e});\ s_t\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$ and by SsFold,

$\langle H, \langle \rho_n, \pi_n, \texttt{fold}\ p(\overline{e});\ s_t\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s_t\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$.

Case 13 ($s_h = \texttt{unfold}\ p(\overline{e})$). Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{unfold}\ p(\overline{e});\ s_t\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$ and by SsUnfold,

$\langle H, \langle \rho_n, \pi_n, \texttt{unfold}\ p(\overline{e});\ s_t\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s_t\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$.

2.2 Preservation

Claim (SVL_{RP} *Preservation*). If ψ is a valid state and $\psi \longrightarrow \psi'$ for some $\psi' \in$ State then ψ' is a valid state.

Proof. Suppose $\psi \in$ State such that ψ is a valid state and $\psi \longrightarrow \psi'$ for some $\psi' \in$ State.

We will show ψ' is a valid state by case analysis on $\psi \longrightarrow \psi'$.

Case 14 (SsSkip). We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip};\ s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$.

Since ψ is a valid state, by definition we get that $\texttt{skip};\ s = s' \,;\, \texttt{skip}$ for some $s' \in$ Stmt, $s_i = s'_i \,;\, \texttt{skip}$ for some $s'_i \in$ Stmt for all $1 \le i < n$, $\pi_i \cap \pi_j = \emptyset$ for all $1 \le i \le n$, $1 \le j \le n$ such that $i \ne j$, and $\langle H, \rho_i, \pi_i\rangle \vDash_E \mathsf{sWLP}_i((\texttt{skip};\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \le i \le n$.

If $n > 1$, then

by the definition of sWLP and since $\mathsf{sWLP}((\texttt{skip};\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is defined, we get that $s_{n-1} = y := z.m(\overline{x})\,;\ s''_2$ or $s_{n-1} = \texttt{while (e) inv}\ \theta_{\texttt{i}}\ \{\ \texttt{r}\ \};\ s''_2$.

$$\text{Then let } \theta = \begin{cases} \mathsf{mpost}(m) & \texttt{if } s_{n-1} = y := z.m(\overline{x})\,;\ s''_2 \\ \theta_i & \texttt{if } s_{n-1} = \texttt{while (e) inv}\ \theta_{\texttt{i}}\ \{\ \texttt{r}\ \};\ s''_2 \end{cases}$$

$$\text{and } \theta_f = \begin{cases} \mathsf{mpre}(m)[z/\texttt{this}, \overline{x/\mathsf{mparam}(m)}] & \texttt{if } s_{n-1} = y := z.m(\overline{x})\,;\ s''_2 \\ \theta_i & \texttt{if } s_{n-1} = \texttt{while (e) inv}\ \theta_{\texttt{i}}\ \{\ \texttt{r}\ \};\ s''_2 \end{cases}.$$

Now,

$\mathsf{sWLP}((\texttt{skip; }s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$
$= \mathsf{WLP}(\texttt{skip; }s, \theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$
$= \mathsf{WLP}(\texttt{skip}, \mathsf{WLP}(s, \theta)) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$
$= \mathsf{WLP}(s, \theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$
$= \mathsf{sWLP}(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ `by the definition of` sWLP.

Therefore, $\mathsf{sWLP}((\texttt{skip; }s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \mathsf{sWLP}(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$, so we get $\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \leq i \leq n$.

If $n = 1$, then

$\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip; } s \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s \rangle \cdot \mathsf{nil} \rangle = \psi'$ and by the definition of sWLP, we get that $\mathsf{sWLP}((\texttt{skip; }s) \cdot \mathsf{nil}, \texttt{true}) = \mathsf{WLP}(s, \texttt{true}) \cdot \mathsf{nil} = \mathsf{sWLP}(s \cdot \mathsf{nil}, \texttt{true})$.

Therefore, $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{sWLP}_n(s \cdot \mathsf{nil}, \texttt{true})$.

In either case $\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \leq i \leq n$. Also, s is clearly `skip` or s''`; skip` for some $s'' \in$ STMT , and we are given that $s_i = s_i'$`; skip` for some $s_i' \in$ STMT for all $1 \leq i < n$. Additionally, $\pi_i \cap \pi_j = \emptyset$ for all $1 \leq i \leq n$, $1 \leq j \leq n$ such that $i \neq j$. Therefore, ψ' is a valid state.

*Case 15 (*SSDECLARE*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{T x; } s \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$.

Since ψ is a valid state, by definition we get that `T x;` $s = s'$`; skip` for some $s' \in$ STMT, $s_i = s_i'$`; skip` for some $s_i' \in$ STMT for all $1 \leq i < n$, $\pi_i \cap \pi_j = \emptyset$ for all $1 \leq i \leq n$, $1 \leq j \leq n$ such that $i \neq j$, and
$\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i((\texttt{T x; } s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \leq i \leq n$.

If $n > 1$, then

by the definition of sWLP and since $\mathsf{sWLP}((\texttt{T x; } s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is defined, we get that $s_{n-1} = y := z.m(\overline{x})\texttt{; } s_2''$ or $s_{n-1} = \texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s_2''$.

Then let $\theta = \begin{cases} \mathsf{mpost}(m) & \texttt{if } s_{n-1} = y := z.m(\overline{x})\texttt{; } s_2'' \\ \theta_i & \texttt{if } s_{n-1} = \texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s_2'' \end{cases}$

and $\theta_f = \begin{cases} \mathsf{mpre}(m)[z/\texttt{this}, \overline{x/\mathsf{mparam}(m)}] & \texttt{if } s_{n-1} = y := z.m(\overline{x})\texttt{; } s_2'' \\ \theta_i & \texttt{if } s_{n-1} = \texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s_2'' \end{cases}$.

Now,

$\mathsf{sWLP}((\texttt{T x; } s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$
$= \mathsf{WLP}(\texttt{T x; } s, \theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$
$= \mathsf{WLP}(s, \theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ `since` $\mathsf{WLP}(\texttt{T x}, \mathsf{WLP}(s, \theta))$ `is defined`
$= \mathsf{sWLP}(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ `by the definition of` sWLP.

Therefore, $\mathsf{sWLP}((\texttt{T x; } s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \mathsf{sWLP}(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$, so we get $\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \leq i \leq n$.

If $n = 1$, then

$\psi = \langle H, \langle \rho_n, \pi_n, \mathtt{T\ x;}\ s\rangle \cdot \mathsf{nil}\rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s\rangle \cdot \mathsf{nil}\rangle = \psi'$ and by the definition of sWLP, $\mathsf{sWLP}((\mathtt{T\ x;}\ s) \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{WLP}(\mathtt{T\ x;}\ s, \mathtt{true}) \cdot \mathsf{nil} = \mathsf{WLP}(\mathtt{T\ x}, \mathsf{WLP}(s, \mathtt{true})) \cdot \mathsf{nil} = \mathsf{WLP}(s, \mathtt{true}) \cdot \mathsf{nil} = \mathsf{sWLP}(s \cdot \mathsf{nil}, \mathtt{true})$.

Therefore, $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{sWLP}_n(s \cdot \mathsf{nil}, \mathtt{true})$.

In either case $\langle H, \rho_i, \pi_i\rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i \le n$. Also, s is clearly $\mathtt{skip}$ or $s''\mathtt{;\ skip}$ for some $s'' \in$ STMT , and we are given that $s_i = s_i'\mathtt{;\ skip}$ for some $s_i' \in$ STMT for all $1 \le i < n$. Additionally, $\pi_i \cap \pi_j = \emptyset$ for all $1 \le i \le n$, $1 \le j \le n$ such that $i \ne j$. Therefore, ψ' is a valid state.

*Case 16 (*SSASSERT*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, \mathtt{assert}\ \phi'\mathtt{;}\ s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$ and $\langle H, \rho_n, \pi_n\rangle \vDash_E \phi'$.

Since ψ is a valid state, by definition we get that $\mathtt{assert}\ \phi'\mathtt{;}\ s = s'\mathtt{;\ skip}$ for some $s' \in$ STMT, $s_i = s_i'\mathtt{;\ skip}$ for some $s_i' \in$ STMT for all $1 \le i < n$, $\pi_i \cap \pi_j = \emptyset$ for all $1 \le i \le n$, $1 \le j \le n$ such that $i \ne j$, and $\langle H, \rho_i, \pi_i\rangle \vDash_E \mathsf{sWLP}_i((\mathtt{assert}\ \phi'\mathtt{;}\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i \le n$.

If $n > 1$, then

by the definition of sWLP and since $\mathsf{sWLP}((\mathtt{assert}\ \phi'\mathtt{;}\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined, we get that $s_{n-1} = y\ \mathtt{:=}\ z\mathtt{.}m(\overline{x})\ \mathtt{;}\ s_2''$ or $s_{n-1} = \mathtt{while\ (e)\ inv}\ \theta_{\mathtt{i}}\ \mathtt{\{\ r\ \};}\ s_2''$.

Then let $\theta = \begin{cases} \mathsf{mpost}(m) & \mathtt{if}\ s_{n-1} = y\ \mathtt{:=}\ z\mathtt{.}m(\overline{x})\ \mathtt{;}\ s_2'' \\ \theta_i & \mathtt{if}\ s_{n-1} = \mathtt{while\ (e)\ inv}\ \theta_{\mathtt{i}}\ \mathtt{\{\ r\ \};}\ s_2'' \end{cases}$

and $\theta_f = \begin{cases} \mathsf{mpre}(m)[z/\mathtt{this}, \overline{x/\mathsf{mparam}(m)}] & \mathtt{if}\ s_{n-1} = y\ \mathtt{:=}\ z\mathtt{.}m(\overline{x})\ \mathtt{;}\ s_2'' \\ \theta_i & \mathtt{if}\ s_{n-1} = \mathtt{while\ (e)\ inv}\ \theta_{\mathtt{i}}\ \mathtt{\{\ r\ \};}\ s_2'' \end{cases}$.

Since $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{sWLP}_n((\mathtt{assert}\ \phi'\mathtt{;}\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, by the definition of sWLP, we get that $\mathsf{sWLP}_n((\mathtt{assert}\ \phi'\mathtt{;}\ s) \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) \Rightarrow \mathsf{WLP}(s, \theta)$. Then lemma 6 gives $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{WLP}(s, \theta)$.

Now, $\mathsf{sWLP}((\mathtt{assert}\ \phi'\mathtt{;}\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{WLP}(\mathtt{assert}\ \phi', \mathsf{WLP}(s, \theta)) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, so $\mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined.

By the definition of sWLP, we get $\mathsf{sWLP}(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{WLP}(s, \theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, and so $\mathsf{sWLP}(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined.

Finally, since $\mathsf{sWLP}_n(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{WLP}(s, \theta)$ and $\mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{sWLP}_i(\mathtt{assert}\ \phi'\mathtt{;}\ s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i < n$, we get $\langle H, \rho_i, \pi_i\rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i \le n$.

If $n = 1$, then

$\psi = \langle H, \langle \rho_n, \pi_n, \mathtt{assert}\ \phi'\mathtt{;}\ s\rangle \cdot \mathsf{nil}\rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s\rangle \cdot \mathsf{nil}\rangle = \psi'$.

Since $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{sWLP}_n((\mathtt{assert}\ \phi'\mathtt{;}\ s) \cdot \mathsf{nil}, \mathtt{true})$, by the definition of sWLP, we get that $\mathsf{sWLP}_n((\mathtt{assert}\ \phi'\mathtt{;}\ s) \cdot \mathsf{nil}, \mathtt{true}) \Rightarrow \mathsf{WLP}(s, \mathtt{true})$. Then lemma 6 gives $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{WLP}(s, \mathtt{true}) = \mathsf{sWLP}_n(s \cdot \mathsf{nil}, \mathtt{true})$.

In either case $\langle H, \rho_i, \pi_i\rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i \le n$. Also, s is clearly skip or s''; skip for some $s'' \in$ STMT, and we are given that $s_i = s_i'$; skip for some $s_i' \in$ STMT for all $1 \le i < n$. Additionally, $\pi_i \cap \pi_j = \emptyset$ for all $1 \le i \le n$, $1 \le j \le n$ such that $i \ne j$. Therefore, ψ' is a valid state.

*Case 17 (*SSFASSIGN*).*
The proof of this case can be found here: extra-proofs/svlrp-preservation-ssfassign.pdf.

*Case 18 (*SSASSIGN*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, x := e;\ s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \longrightarrow \langle H, \langle \rho_n[x \mapsto v], \pi_n, s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$, $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{acc}(e)$, and $H, \rho_n \vdash e \Downarrow v$.

Since ψ is a valid state, by definition we get that $x := e;\ s = s'$; skip for some $s' \in$ STMT, $s_i = s_i'$; skip for some $s_i' \in$ STMT for all $1 \le i < n$, $\pi_i \cap \pi_j = \emptyset$ for all $1 \le i \le n$, $1 \le j \le n$ such that $i \ne j$, and
$\langle H, \rho_i, \pi_i\rangle \vDash_E \mathsf{sWLP}_i((x := e;\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i \le n$.

If $n > 1$, then

by the definition of sWLP and since $\mathsf{sWLP}((x := e;\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined, we get that $s_{n-1} = y := z.m(\overline{x})\ ;\ r'$ or $s_{n-1} = \mathtt{while}\ (e')\ \mathtt{inv}\ \theta_i\ \{\ \mathtt{r}\ \};\ r'$.

Then let $\theta = \begin{cases} \mathsf{mpost}(m) & \mathtt{if}\ s_{n-1} = y := z.m(\overline{x})\ ;\ r' \\ \theta_i & \mathtt{if}\ s_{n-1} = \mathtt{while}\ (e')\ \mathtt{inv}\ \theta_i\ \{\ \mathtt{r}\ \};\ r' \end{cases}$

and $\theta_f = \begin{cases} \mathsf{mpre}(m)[z/\mathtt{this}, \overline{x/\mathsf{mparam}(m)}] & \mathtt{if}\ s_{n-1} = y := z.m(\overline{x})\ ;\ r' \\ \theta_i & \mathtt{if}\ s_{n-1} = \mathtt{while}\ (e')\ \mathtt{inv}\ \theta_i\ \{\ \mathtt{r}\ \};\ r' \end{cases}$.

Since $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{sWLP}_n((x := e;\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, by the definition of sWLP, we get that $\mathsf{sWLP}_n((x := e;\ s) \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) \Rightarrow \mathsf{WLP}(s, \theta)[e/x]$. Then lemma 6 gives $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{WLP}(s, \theta)[e/x]$.
Since $H, \rho_n \vdash e \Downarrow v$, we get $\langle H, \rho_n[x \mapsto v], \pi_n\rangle \vDash_E \mathsf{WLP}(s, \theta)$

.

Now, $\mathsf{sWLP}((x := e;\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{WLP}(x := e, \mathsf{WLP}(s, \theta)) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, so $\mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined.

By the definition of sWLP, we get $\mathsf{sWLP}(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{WLP}(s, \theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, and so $\mathsf{sWLP}(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined.

Finally, since $\mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{sWLP}_i((x := e;\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i < n$, we get
$\langle H, \rho_i, \pi_i\rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i < n$
and we have $\langle H, \rho_n[x \mapsto v], \pi_n\rangle \vDash_E \mathsf{WLP}(s, \theta) = \mathsf{sWLP}_n(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$.

If $n = 1$, then

$\psi = \langle H, \langle \rho_n, \pi_n, x := e;\ s\rangle \cdot \mathsf{nil}\rangle \longrightarrow \langle H, \langle \rho_n[x \mapsto v], \pi_n, s\rangle \cdot \mathsf{nil}\rangle = \psi'$.
Since $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{sWLP}_n((x := e;\ s) \cdot \mathsf{nil}, \mathtt{true})$, by the definition of sWLP, we get that $\mathsf{sWLP}_n((x := e;\ s) \cdot \mathsf{nil}, \mathtt{true}) \Rightarrow \mathsf{WLP}(s, \mathtt{true})[e/x]$.
Then lemma 6 gives $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{WLP}(s, \mathtt{true})[e/x]$.

Since $H, \rho_n \vdash e \Downarrow v$, we get $\langle H, \rho_n[x \mapsto v], \pi_n\rangle \vDash_E \mathsf{WLP}(s, \mathtt{true}) = \mathsf{sWLP}_n(s \cdot \mathsf{nil}, \mathtt{true})$.

In either case, $\langle H, \rho_i, \pi_i\rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i < n$. and $\langle H, \rho_n[x \mapsto v], \pi_n\rangle \vDash_E \mathsf{sWLP}_n(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$.
Also, s is clearly `skip` or s''; `skip` for some $s'' \in$ STMT , and we are given that $s_i = s'_i$; `skip` for some $s'_i \in$ STMT for all $1 \le i < n$. Additionally, $\pi_i \cap \pi_j = \emptyset$ for all $1 \le i \le n$, $1 \le j \le n$ such that $i \neq j$. Therefore, ψ' is a valid state.

*Case 19 (*SSALLOC*).*
The proof of this case can be found here: extra-proofs/svlrp-preservation-ssalloc.pdf.

*Case 20 (*SSIFTRUE*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, \mathtt{if\ (e)\ \{s'_1\}\ else\ \{s'_2\}};\ s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s'_1;\ s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$, $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{acc}(e)$, and $H, \rho_n \vdash e \Downarrow \mathtt{true}$.

Since ψ is a valid state, by definition we get that $\mathtt{if\ (e)\ \{s'_1\}\ else\ \{s'_2\}};\ s = s'$; `skip` for some $s' \in$ STMT, $s_i = s''_i$; `skip` for some $s''_i \in$ STMT for all $1 \le i < n$, $\pi_i \cap \pi_j = \emptyset$ for all $1 \le i \le n$, $1 \le j \le n$ such that $i \neq j$, and
$\langle H, \rho_i, \pi_i\rangle \vDash_E \mathsf{sWLP}_i((\mathtt{if\ (e)\ \{s'_1\}\ else\ \{s'_2\}};\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i \le n$.

If $n > 1$, then

by the definition of sWLP and since $\mathsf{sWLP}((\mathtt{if\ (e)\ \{s'_1\}\ else\ \{s'_2\}};\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined, we get that $s_{n-1} = y := z.m(\overline{x})\ ;\ r'$ or $s_{n-1} = \mathtt{while\ (e)\ inv}\ \theta_{\mathtt{i}}\ \{\ \mathtt{r}\ \};\ r'$.

Then let $\theta = \begin{cases} \mathsf{mpost}(m) & \mathtt{if}\ s_{n-1} = y := z.m(\overline{x})\ ;\ r' \\ \theta_i & \mathtt{if}\ s_{n-1} = \mathtt{while\ (e)\ inv}\ \theta_{\mathtt{i}}\ \{\ \mathtt{r}\ \};\ r' \end{cases}$

and $\theta_f = \begin{cases} \mathsf{mpre}(m)[z/\mathtt{this}, \overline{x/\mathsf{mparam}(m)}] & \mathtt{if}\ s_{n-1} = y := z.m(\overline{x})\ ;\ r' \\ \theta_i & \mathtt{if}\ s_{n-1} = \mathtt{while\ (e)\ inv}\ \theta_{\mathtt{i}}\ \{\ \mathtt{r}\ \};\ r' \end{cases}$.

Since $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{sWLP}_n((\mathtt{if\ (e)\ \{s'_1\}\ else\ \{s'_2\}};\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, by the definition of sWLP, we get that $\mathsf{sWLP}_n((\mathtt{if\ (e)\ \{s'_1\}\ else\ \{s'_2\}};\ s) \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) \Rightarrow$
$(\mathtt{if}\ e\ \mathtt{then}\ \mathsf{WLP}(s'_1, \mathsf{WLP}(s, \theta))\ \mathtt{else}\ \mathsf{WLP}(s'_2, \mathsf{WLP}(s, \theta)))$.
Then lemma 6 gives
$\langle H, \rho_n, \pi_n\rangle \vDash_E (\mathtt{if}\ e\ \mathtt{then}\ \mathsf{WLP}(s'_1, \mathsf{WLP}(s, \theta))\ \mathtt{else}\ \mathsf{WLP}(s'_2, \mathsf{WLP}(s, \theta)))$.
By inversion on this and since $H, \rho_n \vdash e \Downarrow \mathtt{true}$, we get
$\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{WLP}(s'_1, \mathsf{WLP}(s, \theta)) = \mathsf{WLP}(s'_1;\ s, \theta)$.

Now, $\mathsf{sWLP}((\mathtt{if\ (e)\ \{s'_1\}\ else\ \{s'_2\}};\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{WLP}(\mathtt{if\ (e)\ \{s'_1\}\ else\ \{s'_2\}}, \mathsf{WLP}(s, \theta)) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, so $\mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined.

By the definition of sWLP, we get $\mathsf{sWLP}((s'_1;\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{WLP}(s'_1;\ s, \theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, and so $\mathsf{sWLP}((s'_1;\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined.

Finally, since $\mathsf{sWLP}_i((s'_1;\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{sWLP}_i((\mathtt{if\ (e)\ \{s'_1\}\ else\ \{s'_2\}};\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i < n$ and $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{WLP}(s'_1;\ s, \theta)$, we get

$\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i((s'_1\,;\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \leq i \leq n$.

If $n = 1$, then

$\psi = \langle H, \langle \rho_n, \pi_n, \mathtt{if}\ (\mathtt{e})\ \{\mathtt{s}'_1\}\ \mathtt{else}\ \{\mathtt{s}'_2\}\,;\ s\rangle \cdot \mathsf{nil}\rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s'_1\,;\ s\rangle \cdot \mathsf{nil}\rangle = \psi'$.

Since $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{sWLP}_n((\mathtt{if}\ (\mathtt{e})\ \{\mathtt{s}'_1\}\ \mathtt{else}\ \{\mathtt{s}'_2\}\,;\ s) \cdot \mathsf{nil}, \mathtt{true})$, by the definition of sWLP, we get that

$\mathsf{sWLP}_n((\mathtt{if}\ (\mathtt{e})\ \{\mathtt{s}'_1\}\ \mathtt{else}\ \{\mathtt{s}'_2\}\,;\ s) \cdot \mathsf{nil}, \mathtt{true}) \Rightarrow$

$\quad(\mathtt{if}\ e\ \mathtt{then}\ \mathsf{WLP}(s'_1, \mathsf{WLP}(s, \mathtt{true}))\ \mathtt{else}\ \mathsf{WLP}(s'_2, \mathsf{WLP}(s, \mathtt{true})))$.

Then lemma 6 gives

$\langle H, \rho_n, \pi_n \rangle \vDash_E (\mathtt{if}\ e\ \mathtt{then}\ \mathsf{WLP}(s'_1, \mathsf{WLP}(s, \mathtt{true}))\ \mathtt{else}\ \mathsf{WLP}(s'_2, \mathsf{WLP}(s, \mathtt{true})))$.

By inversion on this and since $H, \rho_n \vdash e \Downarrow \mathtt{true}$, we get

$\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{WLP}(s'_1, \mathsf{WLP}(s, \mathtt{true})) = \mathsf{WLP}(s'_1\,;\ s, \mathtt{true}) = \mathsf{sWLP}_n((s'_1\,;\ s) \cdot \mathsf{nil}, \mathtt{true})$.

In either case $\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i((s'_1\,;\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \leq i \leq n$. Also, s is clearly $\mathtt{skip}$ or $s''\,;\ \mathtt{skip}$ for some $s'' \in \textsc{Stmt}$, which gives that $s'_1\,;\ s = s'_1\,;\ \mathtt{skip}$ or $((s'_1\,;\ s'')\,;\ \mathtt{skip})$ for some $s'' \in \textsc{Stmt}$, and we are given that $s_i = s''_i\,;\ \mathtt{skip}$ for some $s''_i \in \textsc{Stmt}$ for all $1 \leq i < n$. Additionally, $\pi_i \cap \pi_j = \emptyset$ for all $1 \leq i \leq n$, $1 \leq j \leq n$ such that $i \neq j$. Therefore, ψ' is a valid state.

Case 21 (SsIfFalse). We have $\psi = \langle H, \langle \rho_n, \pi_n, \mathtt{if}\ (\mathtt{e})\ \{\mathtt{s}'_1\}\ \mathtt{else}\ \{\mathtt{s}'_2\}\,;\ s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s'_2\,;\ s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$, $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$, and $H, \rho_n \vdash e \Downarrow \mathtt{false}$.

Since ψ is a valid state, by definition we get that $\mathtt{if}\ (\mathtt{e})\ \{\mathtt{s}'_1\}\ \mathtt{else}\ \{\mathtt{s}'_2\}\,;\ s = s'\,;\ \mathtt{skip}$ for some $s' \in \textsc{Stmt}$, $s_i = s''_i\,;\ \mathtt{skip}$ for some $s''_i \in \textsc{Stmt}$ for all $1 \leq i < n$, $\pi_i \cap \pi_j = \emptyset$ for all $1 \leq i \leq n$, $1 \leq j \leq n$ such that $i \neq j$, and
$\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i((\mathtt{if}\ (\mathtt{e})\ \{\mathtt{s}'_1\}\ \mathtt{else}\ \{\mathtt{s}'_2\}\,;\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \leq i \leq n$.

If $n > 1$, then

by the definition of sWLP and since $\mathsf{sWLP}((\mathtt{if}\ (\mathtt{e})\ \{\mathtt{s}'_1\}\ \mathtt{else}\ \{\mathtt{s}'_2\}\,;\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined, we get that $s_{n-1} = y := z.m(\overline{x})\,;\ r'$ or $s_{n-1} = \mathtt{while}\ (\mathtt{e})\ \mathtt{inv}\ \theta_\mathtt{i}\ \{\ \mathtt{r}\ \}\,;\ r'$.

Then let $\theta = \begin{cases} \mathsf{mpost}(m) & \text{if } s_{n-1} = y := z.m(\overline{x})\,;\ r' \\ \theta_i & \text{if } s_{n-1} = \mathtt{while}\ (\mathtt{e})\ \mathtt{inv}\ \theta_\mathtt{i}\ \{\ \mathtt{r}\ \}\,;\ r' \end{cases}$

and $\theta_f = \begin{cases} \mathsf{mpre}(m)[z/\mathtt{this}, \overline{x/\mathsf{mparam}(m)}] & \text{if } s_{n-1} = y := z.m(\overline{x})\,;\ r' \\ \theta_i & \text{if } s_{n-1} = \mathtt{while}\ (\mathtt{e})\ \mathtt{inv}\ \theta_\mathtt{i}\ \{\ \mathtt{r}\ \}\,;\ r' \end{cases}$.

Since $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{sWLP}_n((\mathtt{if}\ (\mathtt{e})\ \{\mathtt{s}'_1\}\ \mathtt{else}\ \{\mathtt{s}'_2\}\,;\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, by the definition of sWLP, we get that

$\mathsf{sWLP}_n((\mathtt{if}\ (\mathtt{e})\ \{\mathtt{s}'_1\}\ \mathtt{else}\ \{\mathtt{s}'_2\}\,;\ s) \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) \Rightarrow$

$\quad(\mathtt{if}\ e\ \mathtt{then}\ \mathsf{WLP}(s'_1, \mathsf{WLP}(s, \theta))\ \mathtt{else}\ \mathsf{WLP}(s'_2, \mathsf{WLP}(s, \theta)))$.

Then lemma 6 gives

$\langle H, \rho_n, \pi_n \rangle \vDash_E (\mathtt{if}\ e\ \mathtt{then}\ \mathsf{WLP}(s'_1, \mathsf{WLP}(s, \theta))\ \mathtt{else}\ \mathsf{WLP}(s'_2, \mathsf{WLP}(s, \theta)))$.

By inversion on this and since $H, \rho_n \vdash e \Downarrow \mathtt{false}$, we get

$\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{WLP}(s'_2, \mathsf{WLP}(s, \theta)) = \mathsf{WLP}(s'_2\,;\ s, \theta)$.

Now, $\mathsf{sWLP}((\texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}};\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \mathsf{WLP}(\texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}}, \mathsf{WLP}(s, \theta)) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$, so $\mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is defined.

By the definition of sWLP, we get $\mathsf{sWLP}((s'_2;\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \mathsf{WLP}(s'_2;\ s, \theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$, and so $\mathsf{sWLP}((s'_2;\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is defined.

Finally, since $\mathsf{sWLP}_i((s'_2;\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \mathsf{sWLP}_i((\texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}};\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \le i < n$ and $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{WLP}(s'_2;\ s, \theta)$, we get $\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i((s'_2;\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \le i \le n$.

If $n = 1$, then

$\psi = \langle H, \langle \rho_n, \pi_n, \texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}};\ s \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s'_2;\ s \rangle \cdot \mathsf{nil} \rangle = \psi'$.

Since $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{sWLP}_n((\texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}};\ s) \cdot \mathsf{nil}, \texttt{true})$, by the definition of sWLP, we get that

$\mathsf{sWLP}_n((\texttt{if (e) \{s}'_1\texttt{\} else \{s}'_2\texttt{\}};\ s) \cdot \mathsf{nil}, \texttt{true}) \Rightarrow$
$(\texttt{if}\ e\ \texttt{then}\ \mathsf{WLP}(s'_1, \mathsf{WLP}(s, \texttt{true}))\ \texttt{else}\ \mathsf{WLP}(s'_2, \mathsf{WLP}(s, \texttt{true})))$.

Then lemma 6 gives

$\langle H, \rho_n, \pi_n \rangle \vDash_E (\texttt{if}\ e\ \texttt{then}\ \mathsf{WLP}(s'_1, \mathsf{WLP}(s, \texttt{true}))\ \texttt{else}\ \mathsf{WLP}(s'_2, \mathsf{WLP}(s, \texttt{true})))$.

By inversion on this and since $H, \rho_n \vdash e \Downarrow \texttt{false}$, we get

$\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{WLP}(s'_2, \mathsf{WLP}(s, \texttt{true})) = \mathsf{WLP}(s'_2;\ s, \texttt{true}) = \mathsf{sWLP}_n((s'_2;\ s) \cdot \mathsf{nil}, \texttt{true})$.

In either case $\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i((s'_2;\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \le i \le n$. Also, s is clearly $\texttt{skip}$ or $s'';\ \texttt{skip}$ for some $s'' \in \textsc{Stmt}$, which gives that $s'_2;\ s = s'_2;\ \texttt{skip}$ or $((s'_2;\ s'');\ \texttt{skip})$ for some $s'' \in \textsc{Stmt}$, and we are given that $s_i = s''_i;\ \texttt{skip}$ for some $s''_i \in \textsc{Stmt}$ for all $1 \le i < n$. Additionally, $\pi_i \cap \pi_j = \emptyset$ for all $1 \le i \le n$, $1 \le j \le n$ such that $i \ne j$. Therefore, ψ' is a valid state.

*Case 22 (*SsCall*).*

This case is structured similarly to the proof case for SsWhileTrue.

*Case 23 (*SsCallFinish*).*

The proof of this case can be found here: extra-proofs/svlrp-preservation-sscallfinish.pdf.

*Case 24 (*SsWhileFalse*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{while (e) inv}\ \theta_\texttt{i}\ \texttt{\{ r \}};\ s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$, $\langle H, \rho_n, \pi_n \rangle \vDash_E \theta_i$, $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$, and $H, \rho_n \vdash e \Downarrow \texttt{false}$.

Since ψ is a valid state, by definition we get that $\texttt{while (e) inv}\ \theta_\texttt{i}\ \texttt{\{ r \}};\ s = s';\ \texttt{skip}$ for some $s' \in \textsc{Stmt}$, $s_i = s'_i;\ \texttt{skip}$ for some $s'_i \in \textsc{Stmt}$ for all $1 \le i < n$, $\pi_i \cap \pi_j = \emptyset$ for all $1 \le i \le n$, $1 \le j \le n$ such that $i \ne j$, and $\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i((\texttt{while (e) inv}\ \theta_\texttt{i}\ \texttt{\{ r \}};\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \le i \le n$.

If $n > 1$, then

by the definition of sWLP and since sWLP((while (e) inv $\theta_{\mathtt{i}}$ { r }; s)$\cdot s_{n-1}\cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}$) is defined, we get that $s_{n-1} = y := z.m(\overline{x})\ ;\ s_2''$ or $s_{n-1} = \mathtt{while}\ (\mathtt{e}')\ \mathtt{inv}\ \theta_{\mathtt{i}}'\ \{\ \mathtt{r}'\ \};\ s_2''$.

Then let $\theta = \begin{cases} \mathsf{mpost}(m) & \mathtt{if}\ s_{n-1} = y := z.m(\overline{x})\ ;\ s_2'' \\ \theta_i' & \mathtt{if}\ s_{n-1} = \mathtt{while}\ (\mathtt{e}')\ \mathtt{inv}\ \theta_{\mathtt{i}}'\ \{\ \mathtt{r}'\ \};\ s_2'' \end{cases}$

and $\theta_f = \begin{cases} \mathsf{mpre}(m)[z/\mathtt{this}, \overline{x/\mathsf{mparam}(m)}] & \mathtt{if}\ s_{n-1} = y := z.m(\overline{x})\ ;\ s_2'' \\ \theta_i' & \mathtt{if}\ s_{n-1} = \mathtt{while}\ (\mathtt{e}')\ \mathtt{inv}\ \theta_{\mathtt{i}}'\ \{\ \mathtt{r}'\ \};\ s_2'' \end{cases}$.

Since $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{sWLP}_n$((while (e) inv $\theta_{\mathtt{i}}$ { r }; s)$\cdot s_{n-1}\cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}$), by the definition of sWLP, we get that sWLP_n((while (e) inv $\theta_{\mathtt{i}}$ { r }; s)$\cdot s_{n-1}\cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) \Rightarrow \theta_i * \theta_f$ for $\theta_f \in$ FORMULA such that $\theta_f * (\theta_i * e = \mathtt{false})[\overline{x_i/y_i}] \Rightarrow \mathsf{WLP}(s,\theta)[\overline{x_i/y_i}]$ where $\overline{x_i} \notin \mathsf{FV}(\theta_f)$, $\overline{y_i}$ are variables modified in the loop body r, and $\overline{x_i}$ are fresh logical variables.

Then lemma 6 gives $\langle H, \rho_n, \pi_n\rangle \vDash_E \theta_i * \theta_f$, and by inversion we get $\langle H, \rho_n, \pi_{n_1}\rangle \vDash_E \theta_i$ and $\langle H, \rho_n, \pi_{n_2}\rangle \vDash_E \theta_f$ such that $\pi_{n_1} \uplus \pi_{n_2} = \pi_n$.

Since $H, \rho_n \vdash e \Downarrow \mathtt{false}$ and $H, \rho_n \vdash \mathtt{false} \Downarrow \mathtt{false}$ by axiom EVAL, we get by EVCOMP that $H, \rho_n \vdash e = \mathtt{false} \Downarrow \mathtt{true}$. Then by EVCOMPEXPR, $\langle H, \rho_n, \emptyset\rangle \vDash_E e = \mathtt{false}$. Finally, by EVSEPOP, $\langle H, \rho_n, \pi_{n_1} \uplus \emptyset\rangle \vDash_E \theta_i * e = \mathtt{false}$.

For $\overline{y_{i_k}} \subseteq \overline{y_i}$ such that $\overline{y_{i_k}} \in \mathsf{FV}(\theta_i * e = \mathtt{false})$, we get $\overline{H, \rho_n \vdash y_{i_k} \Downarrow v_k}$ holds for some values $\overline{v_k}$ from inversion on the derivation of $\langle H, \rho_n, \pi_{n_1}\rangle \vDash_E \theta_i * e = \mathtt{false}$. Then let $\rho_n' = \rho_n[\overline{x_{i_k} \mapsto v_k}]$ where $\overline{x_{i_k}} \subseteq \overline{x_i}$.

Since $\overline{x_i}$ being fresh logical variables gives $\overline{x_i} \notin dom(\rho_n)$, $\overline{H, \rho_n \vdash y_{i_k} \Downarrow v_k}$, no variables in $\overline{y_i} \backslash \overline{y_{i_k}}$ are in $\mathsf{FV}(\theta_i * e = \mathtt{false})$, and $\langle H, \rho_n, \pi_{n_1}\rangle \vDash_E \theta_i * e = \mathtt{false}$, we get $\langle H, \rho_n', \pi_{n_1}\rangle \vDash_E (\theta_i * e = \mathtt{false})[\overline{x_i/y_i}]$.

Also, since $\overline{x_i}$ are fresh logical variables gives $\overline{x_i} \notin dom(\rho_n)$, $\overline{x_i} \notin \mathsf{FV}(\theta_f)$, and $\langle H, \rho_n, \pi_{n_2}\rangle \vDash_E \theta_f$, we get $\langle H, \rho_n', \pi_{n_2}\rangle \vDash_E \theta_f$. Therefore, by EVSEPOP, we get $\langle H, \rho_n', \pi_{n_1} \uplus \pi_{n_2} = \pi_n\rangle \vDash_E \theta_f * (\theta_i * e = \mathtt{false})[\overline{x_i/y_i}]$.

Then, since $\theta_f * (\theta_i * e = \mathtt{false})[\overline{x_i/y_i}] \Rightarrow \mathsf{WLP}(s,\theta)[\overline{x_i/y_i}]$, we get from lemma 6 that $\langle H, \rho_n', \pi_n\rangle \vDash_E \mathsf{WLP}(s,\theta)[\overline{x_i/y_i}]$. Clearly, since $\overline{x_i} \notin dom(\rho_n)$ gives no variables in $\overline{x_i} \setminus \overline{x_{i_k}}$ are in $dom(\rho_n')$ and $\langle H, \rho_n', \pi_n\rangle \vDash_E \mathsf{WLP}(s,\theta)[\overline{x_i/y_i}]$ holds, we get that $\langle H, \rho_n', \pi_n\rangle \vDash_E \mathsf{WLP}(s,\theta)[\overline{x_{i_k}/y_{i_k}}]$. Then, since $\overline{H, \rho_n \vdash y_{i_k} \Downarrow v_k}$, we get $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{WLP}(s,\theta)$.

Now, sWLP((while (e) inv $\theta_{\mathtt{i}}$ { r }; s)$\cdot s_{n-1}\cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) =$ WLP((while (e) inv $\theta_{\mathtt{i}}$ { r }; s)$,\theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1}\cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, so $\mathsf{sWLP}^{\theta_f}(s_{n-1}\cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined.

By the definition of sWLP, we get $\mathsf{sWLP}(s \cdot s_{n-1}\cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{WLP}(s,\theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1}\cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, and so $\mathsf{sWLP}(s \cdot s_{n-1}\cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined.

Finally, since $\mathsf{sWLP}_n(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{WLP}(s, \theta)$ and $\mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{sWLP}_i((\mathtt{while}\ (\mathtt{e})\ \mathtt{inv}\ \theta_{\mathtt{i}}\ \{\ \mathtt{r}\ \};\ s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i < n$, we get $\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i \le n$.

If $n = 1$, then

$\psi = \langle H, \langle \rho_n, \pi_n, \mathtt{while}\ (\mathtt{e})\ \mathtt{inv}\ \theta_{\mathtt{i}}\ \{\ \mathtt{r}\ \};\ s \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s \rangle \cdot \mathsf{nil} \rangle = \psi'$.

Since $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{sWLP}_n((\mathtt{while}\ (\mathtt{e})\ \mathtt{inv}\ \theta_{\mathtt{i}}\ \{\ \mathtt{r}\ \};\ s) \cdot \mathsf{nil}, \mathtt{true})$, by the definition of sWLP, we get that $\mathsf{sWLP}_n((\mathtt{while}\ (\mathtt{e})\ \mathtt{inv}\ \theta_{\mathtt{i}}\ \{\ \mathtt{r}\ \};\ s) \cdot \mathsf{nil}, \mathtt{true}) \Rightarrow \theta_i * \theta_f$ for $\theta_f \in \textsc{Formula}$ such that $\theta_f * (\theta_i * e = \mathtt{false})[\overline{x_i/y_i}] \Rightarrow \mathsf{WLP}(s, \mathtt{true})[\overline{x_i/y_i}]$ where $\overline{x_i} \notin \mathsf{FV}(\theta_f)$, $\overline{y_i}$ are variables modified in the loop body r, and $\overline{x_i}$ are fresh logical variables.

Then lemma 6 gives $\langle H, \rho_n, \pi_n \rangle \vDash_E \theta_i * \theta_f$, and by inversion we get $\langle H, \rho_n, \pi_{n_1} \rangle \vDash_E \theta_i$ and $\langle H, \rho_n, \pi_{n_2} \rangle \vDash_E \theta_f$ such that $\pi_{n_1} \uplus \pi_{n_2} = \pi_n$.

Since $H, \rho_n \vdash e \Downarrow \mathtt{false}$ and $H, \rho_n \vdash \mathtt{false} \Downarrow \mathtt{false}$ by axiom EVal, we get by EvComp that $H, \rho_n \vdash e = \mathtt{false} \Downarrow \mathtt{true}$. Then by EvCompExpr, $\langle H, \rho_n, \emptyset \rangle \vDash_E e = \mathtt{false}$. Finally, by EvSepOp, $\langle H, \rho_n, \pi_{n_1} \uplus \emptyset \rangle \vDash_E \theta_i * e = \mathtt{false}$.

For $\overline{y_{i_k}} \subseteq \overline{y_i}$ such that $\overline{y_{i_k}} \in \mathsf{FV}(\theta_i * e = \mathtt{false})$, we get $\overline{H, \rho_n \vdash y_{i_k} \Downarrow v_k}$ holds for some values $\overline{v_k}$ from inversion on the derivation of $\langle H, \rho_n, \pi_{n_1} \rangle \vDash_E \theta_i * e = \mathtt{false}$. Then let $\rho_n' = \rho_n[\overline{x_{i_k} \mapsto v_k}]$ where $\overline{x_{i_k}} \subseteq \overline{x_i}$.

Since $\overline{x_i}$ being fresh logical variables gives $\overline{x_i} \notin dom(\rho_n)$, $\overline{H, \rho_n \vdash y_{i_k} \Downarrow v_k}$, no variables in $\overline{y_i} \backslash \overline{y_{i_k}}$ are in $\mathsf{FV}(\theta_i * e = \mathtt{false})$, and $\langle H, \rho_n, \pi_{n_1} \rangle \vDash_E \theta_i * e = \mathtt{false}$, we get $\langle H, \rho_n', \pi_{n_1} \rangle \vDash_E (\theta_i * e = \mathtt{false})[\overline{x_i/y_i}]$.

Also, since $\overline{x_i}$ are fresh logical variables gives $\overline{x_i} \notin dom(\rho_n)$, $\overline{x_i} \notin \mathsf{FV}(\theta_f)$, and $\langle H, \rho_n, \pi_{n_2} \rangle \vDash_E \theta_f$, we get $\langle H, \rho_n', \pi_{n_2} \rangle \vDash_E \theta_f$. Therefore, by EvSepOp, we get

$\langle H, \rho_n', \pi_{n_1} \uplus \pi_{n_2} = \pi_n \rangle \vDash_E \theta_f * (\theta_i * e = \mathtt{false})[\overline{x_i/y_i}]$.

Then, since $\theta_f * (\theta_i * e = \mathtt{false})[\overline{x_i/y_i}] \Rightarrow \mathsf{WLP}(s, \mathtt{true})[\overline{x_i/y_i}]$, we get from lemma 6 that $\langle H, \rho_n', \pi_n \rangle \vDash_E \mathsf{WLP}(s, \mathtt{true})[\overline{x_i/y_i}]$. Clearly, since $\overline{x_i} \notin dom(\rho_n)$ gives no variables in $\overline{x_i} \setminus \overline{x_{i_k}}$ are in $dom(\rho_n')$ and $\langle H, \rho_n', \pi_n \rangle \vDash_E \mathsf{WLP}(s, \mathtt{true})[\overline{x_i/y_i}]$ holds, we get that $\langle H, \rho_n', \pi_n \rangle \vDash_E \mathsf{WLP}(s, \mathtt{true})[\overline{x_{i_k}/y_{i_k}}]$. Then, since $\overline{H, \rho_n \vdash y_{i_k} \Downarrow v_k}$, we get $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{WLP}(s, \mathtt{true}) = \mathsf{sWLP}_n(s \cdot \mathsf{nil}, \mathtt{true})$.

In either case $\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \le i \le n$. Also, s is clearly $\mathtt{skip}$ or s''; $\mathtt{skip}$ for some $s'' \in \textsc{Stmt}$, and we are given that $s_i = s_i'$; $\mathtt{skip}$ for some $s_i' \in \textsc{Stmt}$ for all $1 \le i < n$. Additionally, $\pi_i \cap \pi_j = \emptyset$ for all $1 \le i \le n$, $1 \le j \le n$ such that $i \ne j$. Therefore, ψ' is a valid state.

Case 25 (SsWhileTrue). We have $\psi = \langle H, \langle \rho_n, \pi_n, \mathtt{while}\ (\mathtt{e})\ \mathtt{inv}\ \theta_{\mathtt{i}}\ \{\ \mathtt{r}\ \};\ s \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n', r;\ \mathtt{skip} \rangle \cdot \langle \rho_n, \pi_n \setminus \pi_n', \mathtt{while}\ (\mathtt{e})\ \mathtt{inv}\ \theta_{\mathtt{i}}\ \{\ \mathtt{r}\ \};\ s \rangle \cdot$

$\ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$,$\langle H, \rho_n, \pi_n \rangle \vDash_E \theta_i$, $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$, $H, \rho_n \vdash e \Downarrow \texttt{true}$, and $\pi'_n = \langle\langle \lfloor \theta_i \rfloor_{H,\rho_n} \rangle\rangle_H$.

Since ψ is a valid state, by definition we get that $\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s = s'\texttt{; skip}$ for some $s' \in$ STMT, $s_i = s'_i\texttt{; skip}$ for some $s'_i \in$ STMT for all $1 \leq i < n$, $\pi_i \cap \pi_j = \emptyset$ for all $1 \leq i \leq n$, $1 \leq j \leq n$ such that $i \neq j$, and $\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i((\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \leq i \leq n$.

If $n > 1$, then

by the definition of sWLP and since $\mathsf{sWLP}((\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is defined, we get that $s_{n-1} = y \texttt{ := } z.m(\overline{x})\texttt{ ; } s''_2$ or $s_{n-1} = \texttt{while (e') inv } \theta'_{\texttt{i}} \texttt{ \{ r' \}; } s''_2$.

Then let $\theta = \begin{cases} \mathsf{mpost}(m) & \texttt{if } s_{n-1} = y \texttt{ := } z.m(\overline{x})\texttt{ ; } s''_2 \\ \theta'_i & \texttt{if } s_{n-1} = \texttt{while (e') inv } \theta'_{\texttt{i}} \texttt{ \{ r' \}; } s''_2 \end{cases}$

and $\theta_f = \begin{cases} \mathsf{mpre}(m)[z/\texttt{this}, \overline{x/\mathsf{mparam}(m)}] & \texttt{if } s_{n-1} = y \texttt{ := } z.m(\overline{x})\texttt{ ; } s''_2 \\ \theta'_i & \texttt{if } s_{n-1} = \texttt{while (e') inv } \theta'_{\texttt{i}} \texttt{ \{ r' \}; } s''_2 \end{cases}$.

1) Since $H, \rho_n \vdash e \Downarrow \texttt{true}$ and $H, \rho_n \vdash \texttt{true} \Downarrow \texttt{true}$ by EVAL, by EVCOMP we get $H, \rho_n \vdash e = \texttt{true} \Downarrow \texttt{true}$. Then by EVCOMPEXPR we get $\langle H, \rho_n, \pi'_n \rangle \vDash_E e = \texttt{true}$.

Also, since $\langle\langle \lfloor \theta_i \rfloor_{H,\rho_n} \rangle\rangle_H$ is defined and equals π'_n, and $\langle H, \rho_n, \pi_n \rangle \vDash_E \theta_i$, we get by lemma 5 that $\langle H, \rho_n, \pi'_n \rangle \vDash_E \theta_i$. Then, by EVANDOP $\langle H, \rho_n, \pi'_n \rangle \vDash_E \theta_i \wedge e = \texttt{true}$.

Since the program we are operating over is valid ,
we get $\theta_i \wedge e = \texttt{true} \Rightarrow \mathsf{WLP}(r, \theta_i)$. Then, since $\langle H, \rho_n, \pi'_n \rangle \vDash_E \theta_i \wedge e = \texttt{true}$ by lemma 6, we get
$\langle H, \rho_n, \pi'_n \rangle \vDash_E \mathsf{WLP}(r, \theta_i) = \mathsf{WLP}(r, \mathsf{WLP}(\texttt{skip}, \theta_i)) = \mathsf{WLP}(r\texttt{; skip}, \theta_i)$ **(1)**.

2) Now, $\mathsf{sWLP}((\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \mathsf{WLP}((\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s), \theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$, so $\mathsf{WLP}((\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s), \theta)$ and $\mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ are defined.

Then, we have $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{WLP}((\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s), \theta) = \mathsf{WLP}(\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}}, \mathsf{WLP}(s, \theta))$.

Let $\theta'_n = \min_{\Rightarrow} \{\theta''_n \mid \mathsf{WLP}(\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s, \theta) \Rightarrow \theta_i * \theta''_n\}$, which is defined since $\{\theta''_n \mid \mathsf{WLP}(\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s, \theta) \Rightarrow \theta_i * \theta''_n\}$ is non-empty from $\mathsf{WLP}(\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s, \theta)$ being defined.

Since $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{WLP}((\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s), \theta)$ and $\mathsf{WLP}(\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s, \theta) \Rightarrow \theta_i * \theta'_n$, by lemma 6 we get $\langle H, \rho_n, \pi_n \rangle \vDash_E \theta_i * \theta'_n$.

By inversion on $\langle H, \rho_n, \pi_n \rangle \vDash_E \theta_i * \theta'_n$, we get $\langle H, \rho_n, \pi_{n_1} \rangle \vDash_E \theta_i$, $\langle H, \rho_n, \pi_{n_2} \rangle \vDash_E \theta'_n$, and $\pi_{n_1} \uplus \pi_{n_2} = \pi_n$ for some π_{n_1} and π_{n_2}.

Since $\langle\langle \lfloor \theta_i \rfloor_{H,\rho_n} \rangle\rangle_H$ is defined, then $\lfloor \theta_i \rfloor_{H,\rho_n}$ must be defined. Also, by lemma 8 on $\langle H, \rho_n, \pi_{n_1} \rangle \vDash_E \theta_i$, we get that $\exists\ \Pi\ .\ \langle H, \rho_n, \Pi \rangle$ is a good iso-state, $\langle\langle\, \Pi \,\rangle\rangle_H = \pi_{n_1}$ and $\langle H, \rho_n, \Pi \rangle \vDash_I \theta_i$. Then, by definition $\lfloor \theta_i \rfloor_{H,\rho_n} \subseteq \Pi$.

Since $\lfloor \theta_i \rfloor_{H,\rho_n} \subseteq \Pi$ and $\langle\langle\, \Pi \,\rangle\rangle_H$ is defined, we get by lemma 4 that $\langle\langle\, \lfloor \theta_i \rfloor_{H,\rho_n} \,\rangle\rangle_H$ is defined and $\pi'_n = \langle\langle\, \lfloor \theta_i \rfloor_{H,\rho_n} \,\rangle\rangle_H \subseteq \langle\langle\, \Pi \,\rangle\rangle_H = \pi_{n_1}$. Therefore, $\pi'_n \cap \pi_{n_2} = \emptyset$ and so $\pi_{n_2} \subseteq \pi_n \setminus \pi'_n$. Also, $\pi'_n \subseteq \pi_{n_1} \subseteq \pi_n$.

Finally, by lemma 1 on $\pi_{n_2} \subseteq \pi_n \setminus \pi'_n$ and $\langle H, \rho_n, \pi_{n_2} \rangle \vDash_E \theta'_n$, we get $\langle H, \rho_n, \pi_n \setminus \pi'_n \rangle \vDash_E \theta'_n$ **(2)**.

By definition, $\mathsf{sWLP}^{\theta_i}((\texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \min_{\Rightarrow} \{\theta''_n \mid \mathsf{WLP}(\texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s, \theta) \Rightarrow \theta_i * \theta''_n\} \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$, then $\mathsf{sWLP}^{\theta_i}((\texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is defined.

By the definition of sWLP, we get $\mathsf{sWLP}((r; \texttt{skip}) \cdot (\texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \mathsf{WLP}(r; \texttt{skip}, \theta_i) \cdot \mathsf{sWLP}^{\theta_i}((\texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$, and so $\mathsf{sWLP}((r; \texttt{skip}) \cdot (\texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is defined.

Since $\langle H, \rho_n, \pi'_n \rangle \vDash_E \mathsf{WLP}(r; \texttt{skip}, \theta_i) = \mathsf{sWLP}_{n+1}((r; \texttt{skip}) \cdot (\texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$,

$\langle H, \rho_n, \pi_n \setminus \pi'_n \rangle \vDash_E \theta'_n = \mathsf{sWLP}_n((r; \texttt{skip}) \cdot (\texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$, and

$\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i((\texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \mathsf{sWLP}_i((r; \texttt{skip}) \cdot (\texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s) \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \le i < n$.

Also, $r; \texttt{skip}$ is clearly $s''; \texttt{skip}$ for some $s'' \in$ STMT where that s'' is r. Additionally, we are given that $\texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s = s'; \texttt{skip}$ for some $s' \in$ STMT, $s_i = s'_i; \texttt{skip}$ for some $s'_i \in$ STMT for all $1 \le i < n$. Also, $\pi'_n \subseteq \pi_n$ and $\pi_n \cap \pi_i = \emptyset$ for all $1 \le i \le n-1$ gives $\pi'_n \cap \pi_i = \emptyset$ for all $1 \le i \le n-1$ and $(\pi_n \setminus \pi'_n) \cap \pi_i = \emptyset$ for all $1 \le i \le n-1$. Clearly, $\pi'_n \cap (\pi_n \setminus \pi'_n) = \emptyset$. Therefore, ψ' is a valid state in this case.

If $n = 1$, then

$\psi = \langle H, \langle \rho_n, \pi_n, \texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi'_n, r; \texttt{skip} \rangle \cdot \langle \rho_n, \pi_n \setminus \pi'_n, \texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s \rangle \cdot \mathsf{nil} \rangle = \psi'$.

1) Since $H, \rho_n \vdash e \Downarrow \texttt{true}$ and $H, \rho_n \vdash \texttt{true} \Downarrow \texttt{true}$ by EVAL, by EVCOMP we get $H, \rho_n \vdash e = \texttt{true} \Downarrow \texttt{true}$. Then by EVCOMPEXPR we get $\langle H, \rho_n, \pi'_n \rangle \vDash_E e = \texttt{true}$.

Also, since $\langle\langle\, \lfloor \theta_i \rfloor_{H,\rho_n} \,\rangle\rangle_H$ is defined and equals π'_n, and $\langle H, \rho_n, \pi_n \rangle \vDash_E \theta_i$, we get by lemma 5 that $\langle H, \rho_n, \pi'_n \rangle \vDash_E \theta_i$. Then, by EVANDOP $\langle H, \rho_n, \pi'_n \rangle \vDash_E \theta_i \wedge e = \texttt{true}$.

Since the program we are operating over is valid ,
we get $\theta_i \wedge e = \texttt{true} \Rightarrow \mathsf{WLP}(r, \theta_i)$. Then, since $\langle H, \rho_n, \pi'_n \rangle \vDash_E \theta_i \wedge e = \texttt{true}$ by lemma 6, we get

$\langle H, \rho_n, \pi'_n \rangle \vDash_E \mathsf{WLP}(r, \theta_i) = \mathsf{WLP}(r, \mathsf{WLP}(\texttt{skip}, \theta_i)) = \mathsf{WLP}(r; \texttt{skip}, \theta_i)$ **(1)**.

2) Now, $\mathsf{sWLP}((\texttt{while (e) inv } \theta_\texttt{i} \texttt{ \{ r \}; } s) \cdot \mathsf{nil}, \texttt{true}) =$

$\mathsf{WLP}((\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s), \texttt{true}) \cdot \mathsf{nil}$.

So, we have $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{WLP}((\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s), \texttt{true}) = \mathsf{WLP}(\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}}, \mathsf{WLP}(s, \texttt{true}))$.

Let $\theta'_n = \min_{\Rightarrow} \{\theta''_n \mid \mathsf{WLP}(\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s, \theta) \Rightarrow \theta_i * \theta''_n \}$, which is defined since $\{\theta''_n \mid \mathsf{WLP}(\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s, \theta) \Rightarrow \theta_i * \theta''_n \}$ is non-empty from $\mathsf{WLP}(\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s, \theta)$ being defined.

Since $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{WLP}((\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s), \texttt{true})$ and $\mathsf{WLP}(\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s, \texttt{true}) \Rightarrow \theta_i * \theta'_n$, by lemma 6 we get $\langle H, \rho_n, \pi_n \rangle \vDash_E \theta_i * \theta'_n$.

By inversion on $\langle H, \rho_n, \pi_n \rangle \vDash_E \theta_i * \theta'_n$, we get $\langle H, \rho_n, \pi_{n_1} \rangle \vDash_E \theta_i$, $\langle H, \rho_n, \pi_{n_2} \rangle \vDash_E \theta'_n$, and $\pi_{n_1} \uplus \pi_{n_2} = \pi_n$ for some π_{n_1} and π_{n_2}.

Since $\langle\langle \lfloor \theta_i \rfloor_{H,\rho_n} \rangle\rangle_H$ is defined, then $\lfloor \theta_i \rfloor_{H,\rho_n}$ must be defined. Also, by lemma 8 on $\langle H, \rho_n, \pi_{n_1} \rangle \vDash_E \theta_i$, we get that $\exists\, \Pi\ .\ \langle H, \rho_n, \Pi \rangle$ is a good iso-state, $\langle\langle \Pi \rangle\rangle_H = \pi_{n_1}$ and $\langle H, \rho_n, \Pi \rangle \vDash_I \theta_i$. Then, by definition $\lfloor \theta_i \rfloor_{H,\rho_n} \subseteq \Pi$.

Since $\lfloor \theta_i \rfloor_{H,\rho_n} \subseteq \Pi$ and $\langle\langle \Pi \rangle\rangle_H$ is defined, we get by lemma 4 that $\langle\langle \lfloor \theta_i \rfloor_{H,\rho_n} \rangle\rangle_H$ is defined and $\pi'_n = \langle\langle \lfloor \theta_i \rfloor_{H,\rho_n} \rangle\rangle_H \subseteq \langle\langle \Pi \rangle\rangle_H = \pi_{n_1}$. Therefore, $\pi'_n \cap \pi_{n_2} = \emptyset$ and so $\pi_{n_2} \subseteq \pi_n \setminus \pi'_n$. Also, $\pi'_n \subseteq \pi_{n_1} \subseteq \pi_n$.

Finally, by lemma 1 on $\pi_{n_2} \subseteq \pi_n \setminus \pi'_n$ and $\langle H, \rho_n, \pi_{n_2} \rangle \vDash_E \theta'_n$, we get $\langle H, \rho_n, \pi_n \setminus \pi'_n \rangle \vDash_E \theta'_n$ **(2)**.

By definition, $\mathsf{sWLP}^{\theta_i}((\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s) \cdot \mathsf{nil}, \texttt{true}) = \min_{\Rightarrow} \{\theta''_n \mid \mathsf{WLP}(\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s, \texttt{true}) \Rightarrow \theta_i * \theta''_n \} \cdot \mathsf{nil}$, then $\mathsf{sWLP}^{\theta_i}((\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s) \cdot \mathsf{nil}, \texttt{true})$ is defined.

By the definition of sWLP, we get $\mathsf{sWLP}((r\texttt{; skip}) \cdot (\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s) \cdot \mathsf{nil}, \texttt{true}) = \mathsf{WLP}(r\texttt{; skip}, \theta_i) \cdot \mathsf{sWLP}^{\theta_i}((\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s) \cdot \mathsf{nil}, \texttt{true})$, and so $\mathsf{sWLP}((r\texttt{; skip}) \cdot (\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s) \cdot \mathsf{nil}, \texttt{true})$ is defined.

Since $\langle H, \rho_n, \pi'_n \rangle \vDash_E \mathsf{WLP}(r\texttt{; skip}, \theta_i) = \mathsf{sWLP}_{n+1}((r\texttt{; skip}) \cdot (\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s) \cdot \mathsf{nil}, \texttt{true})$, and

$\langle H, \rho_n, \pi_n \setminus \pi'_n \rangle \vDash_E \theta'_n = \mathsf{sWLP}_n((r\texttt{; skip}) \cdot (\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s) \cdot \mathsf{nil}, \texttt{true})$.

Also, $r\texttt{; skip}$ is clearly $s''\texttt{; skip}$ for some $s'' \in \textsc{Stmt}$ where that s'' is r. Additionally, we are given that $\texttt{while (e) inv } \theta_{\texttt{i}} \texttt{ \{ r \}; } s = s'\texttt{; skip}$ for some $s' \in \textsc{Stmt}$. Also, $\pi'_n \subseteq \pi_n$ gives $\pi'_n \cap (\pi_n \setminus \pi'_n) = \emptyset$. Therefore, ψ' is a valid state in this case.

In either case, ψ' is a valid state.

Case 26 (SsWhileFinish).

The proof of this case can be found here: extra-proofs/svlrp-preservation-sswhilefinish.pdf.

Case 27 (SsFold). We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{fold } p(\overline{e})\texttt{; } s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$.

Since ψ is a valid state, by definition we get that $\texttt{fold}\ p(\overline{e});\ s = s';\ \texttt{skip}$ for some $s' \in$ STMT, $s_i = s'_i;\ \texttt{skip}$ for some $s'_i \in$ STMT for all $1 \leq i < n$, $\pi_i \cap \pi_j = \emptyset$ for all $1 \leq i \leq n$, $1 \leq j \leq n$ such that $i \neq j$, and
$\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i((\texttt{fold}\ p(\overline{e});\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \leq i \leq n$.

If $n > 1$, then

by the definition of sWLP and since $\mathsf{sWLP}((\texttt{fold}\ p(\overline{e});\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is defined, we get that $s_{n-1} = y := z.m(\overline{x});\ s''_2$ or $s_{n-1} = \texttt{while (e) inv}\ \theta_\texttt{i}\ \texttt{\{ r \}};\ s''_2$.

Then let $\theta = \begin{cases} \mathsf{mpost}(m) & \texttt{if}\ s_{n-1} = y := z.m(\overline{x});\ s''_2 \\ \theta_i & \texttt{if}\ s_{n-1} = \texttt{while (e) inv}\ \theta_\texttt{i}\ \texttt{\{ r \}};\ s''_2 \end{cases}$

and $\theta_f = \begin{cases} \mathsf{mpre}(m)[z/\texttt{this}, \overline{x/\mathsf{mparam}(m)}] & \texttt{if}\ s_{n-1} = y := z.m(\overline{x});\ s''_2 \\ \theta_i & \texttt{if}\ s_{n-1} = \texttt{while (e) inv}\ \theta_\texttt{i}\ \texttt{\{ r \}};\ s''_2 \end{cases}$.

Since $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{sWLP}_n((\texttt{fold}\ p(\overline{e});\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$, by the definition of sWLP, we get that
$\mathsf{sWLP}_n((\texttt{fold}\ p(\overline{e});\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \phi' * \mathsf{body}_\mu(p)(\overline{e})$ for the weakest $\phi' \in$ FORMULA such that $\phi' * p(\overline{e}) \Rightarrow \mathsf{WLP}(s, \theta)$, $\phi' * p(\overline{e}) \in$ SATFORMULA, and $\phi' * \mathsf{body}_\mu(p)(\overline{e}) \in$ SFRMFORMULA.

By inversion on $\langle H, \rho_n, \pi_n \rangle \vDash_E \phi' * \mathsf{body}_\mu(p)(\overline{e})$, we get $\langle H, \rho_n, \pi_{n_1} \rangle \vDash_E \phi'$ and $\langle H, \rho_n, \pi_{n_2} \rangle \vDash_E \mathsf{body}_\mu(p)(\overline{e})$ for $\pi_{n_1} \uplus \pi_{n_2} = \pi_n$.

Also, since we are operating over well-typed programs, we get $\overline{H, \rho_n \vdash e \Downarrow v}$
.

Then EVPRED gives $\langle H, \rho_n, \pi_{n_2} \rangle \vDash_E p(\overline{e})$, and so EVSEPOP gives $\langle H, \rho_n, \pi_n \rangle \vDash_E \phi' * p(\overline{e})$. Finally, since $\phi' * p(\overline{e}) \Rightarrow \mathsf{WLP}(s, \theta)$, by lemma 6, we get $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{WLP}(s, \theta)$.

Now, $\mathsf{sWLP}((\texttt{fold}\ p(\overline{e});\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \mathsf{WLP}((\texttt{fold}\ p(\overline{e});\ s), \theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$, so $\mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is defined.

By the definition of sWLP, we get $\mathsf{sWLP}(s \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \mathsf{WLP}(s, \theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$, and so $\mathsf{sWLP}(s \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is defined.

Finally, since $\mathsf{sWLP}_n(s \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \mathsf{WLP}(s, \theta)$ and $\mathsf{sWLP}_i(s \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true}) = \mathsf{sWLP}_i((\texttt{fold}\ p(\overline{e});\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \leq i < n$, we get
$\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \leq i \leq n$.

If $n = 1$, then

$\psi = \langle H, \langle \rho_n, \pi_n, \texttt{fold}\ p(\overline{e});\ s \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s \rangle \cdot \mathsf{nil} \rangle = \psi'$.

Since $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{sWLP}_n((\texttt{fold}\ p(\overline{e});\ s) \cdot \mathsf{nil}, \texttt{true})$, by the definition of sWLP, we get that
$\mathsf{sWLP}_n((\texttt{fold}\ p(\overline{e});\ s) \cdot \mathsf{nil}, \texttt{true}) = \phi' * \mathsf{body}_\mu(p)(\overline{e})$ for the weakest $\phi' \in$ FORMULA such that $\phi' * p(\overline{e}) \Rightarrow \mathsf{WLP}(s, \texttt{true})$, $\phi' * p(\overline{e}) \in$ SATFORMULA, and $\phi' * \mathsf{body}_\mu(p)(\overline{e}) \in$ SFRMFORMULA.

By inversion on $\langle H, \rho_n, \pi_n \rangle \vDash_E \phi' * \mathsf{body}_\mu(p)(\overline{e})$, we get $\langle H, \rho_n, \pi_{n_1} \rangle \vDash_E \phi'$ and $\langle H, \rho_n, \pi_{n_2} \rangle \vDash_E \mathsf{body}_\mu(p)(\overline{e})$ for $\pi_{n_1} \uplus \pi_{n_2} = \pi_n$.

Also, since we are operating over well-typed programs, we get $\overline{H, \rho_n \vdash e \Downarrow v}$

.

Then EvPred gives $\langle H, \rho_n, \pi_{n_2} \rangle \vDash_E p(\overline{e})$, and so EvSepOp gives $\langle H, \rho_n, \pi_n \rangle \vDash_E \phi' * p(\overline{e})$. Finally, since $\phi' * p(\overline{e}) \Rightarrow \mathsf{WLP}(s, \mathtt{true})$, by lemma 6, we get $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{WLP}(s, \mathtt{true}) = \mathsf{sWLP}_n(s \cdot \mathsf{nil}, \mathtt{true})$.

In either case $\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \leq i \leq n$. Also, s is clearly $\mathtt{skip}$ or s''; $\mathtt{skip}$ for some $s'' \in$ Stmt , and we are given that $s_i = s'_i$; $\mathtt{skip}$ for some $s'_i \in$ Stmt for all $1 \leq i < n$. Additionally, $\pi_i \cap \pi_j = \emptyset$ for all $1 \leq i \leq n$, $1 \leq j \leq n$ such that $i \neq j$. Therefore, ψ' is a valid state.

Case 28 (SsUnfold). We have $\psi = \langle H, \langle \rho_n, \pi_n, \mathtt{unfold}\ p(\overline{e});\ s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$.

Since ψ is a valid state, by definition we get that $\mathtt{unfold}\ p(\overline{e});\ s = s'$; $\mathtt{skip}$ for some $s' \in$ Stmt, $s_i = s'_i$; $\mathtt{skip}$ for some $s'_i \in$ Stmt for all $1 \leq i < n$, $\pi_i \cap \pi_j = \emptyset$ for all $1 \leq i \leq n$, $1 \leq j \leq n$ such that $i \neq j$, and
$\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i((\mathtt{unfold}\ p(\overline{e});\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \leq i \leq n$.

If $n > 1$, then

by the definition of sWLP and since $\mathsf{sWLP}((\mathtt{unfold}\ p(\overline{e});\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined, we get that $s_{n-1} = y := z.m(\overline{x})\ ;\ s''_2$ or $s_{n-1} = \mathtt{while\ (e)\ inv}\ \theta_{\mathtt{i}}\ \{\ \mathtt{r}\ \};\ s''_2$.

Then let $\theta = \begin{cases} \mathsf{mpost}(m) & \mathtt{if}\ s_{n-1} = y := z.m(\overline{x})\ ;\ s''_2 \\ \theta_i & \mathtt{if}\ s_{n-1} = \mathtt{while\ (e)\ inv}\ \theta_{\mathtt{i}}\ \{\ \mathtt{r}\ \};\ s''_2 \end{cases}$

and $\theta_f = \begin{cases} \mathsf{mpre}(m)[z/\mathtt{this}, \overline{x/\mathsf{mparam}(m)}] & \mathtt{if}\ s_{n-1} = y := z.m(\overline{x})\ ;\ s''_2 \\ \theta_i & \mathtt{if}\ s_{n-1} = \mathtt{while\ (e)\ inv}\ \theta_{\mathtt{i}}\ \{\ \mathtt{r}\ \};\ s''_2 \end{cases}$.

Since $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{sWLP}_n((\mathtt{unfold}\ p(\overline{e});\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, by the definition of sWLP, we get that
$\mathsf{sWLP}_n((\mathtt{unfold}\ p(\overline{e});\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \phi' * p(\overline{e})$ for the weakest $\phi' \in$ Formula such that $\phi' * \mathsf{body}_\mu(p)(\overline{e}) \Rightarrow \mathsf{WLP}(s, \theta)$, $\phi' * \mathsf{body}_\mu(p)(\overline{e}) \in$ SatFormula, and $\phi' * p(\overline{e}) \in$ SfrmFormula.

By inversion on $\langle H, \rho_n, \pi_n \rangle \vDash_E \phi' * p(\overline{e})$, we get $\langle H, \rho_n, \pi_{n_1} \rangle \vDash_E \phi'$ and $\langle H, \rho_n, \pi_{n_2} \rangle \vDash_E p(\overline{e})$ for $\pi_{n_1} \uplus \pi_{n_2} = \pi_n$.

By inversion on $\langle H, \rho_n, \pi_{n_2} \rangle \vDash_E p(\overline{e})$, we get $\langle H, \rho_n, \pi_{n_2} \rangle \vDash_E \mathsf{body}_\mu(p)(\overline{e})$ and $\overline{H, \rho_n \vdash e \Downarrow v}$, and so EvSepOp gives $\langle H, \rho_n, \pi_n \rangle \vDash_E \phi' * \mathsf{body}_\mu(p)(\overline{e})$. Finally, since $\phi' * \mathsf{body}_\mu(p)(\overline{e}) \Rightarrow \mathsf{WLP}(s, \theta)$, by lemma 6, we get $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{WLP}(s, \theta)$.

Now, $\mathsf{sWLP}((\mathtt{unfold}\ p(\overline{e});\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{WLP}((\mathtt{unfold}\ p(\overline{e});\ s), \theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, so $\mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined.

By the definition of sWLP, we get $\mathsf{sWLP}(s \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{WLP}(s, \theta) \cdot \mathsf{sWLP}^{\theta_f}(s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$, and so $\mathsf{sWLP}(s \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ is defined.

Finally, since $\mathsf{sWLP}_n(s \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{WLP}(s, \theta)$ and $\mathsf{sWLP}_i(s \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) = \mathsf{sWLP}_i((\mathtt{unfold}\ p(\overline{e});\ s) \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \leq i < n$, we get
$\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ for all $1 \leq i \leq n$.

If $n = 1$, then

$\psi = \langle H, \langle \rho_n, \pi_n, \texttt{unfold}\ p(\overline{e});\ s\rangle \cdot \mathsf{nil}\rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s\rangle \cdot \mathsf{nil}\rangle = \psi'$.

Since $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{sWLP}_n((\texttt{unfold}\ p(\overline{e});\ s) \cdot \mathsf{nil}, \texttt{true})$, by the definition of sWLP, we get that

$\mathsf{sWLP}_n((\texttt{unfold}\ p(\overline{e});\ s) \cdot \mathsf{nil}, \texttt{true}) = \phi' * p(\overline{e})$ for the weakest $\phi' \in$ FORMULA such that $\phi' * \mathsf{body}_\mu(p)(\overline{e}) \Rightarrow \mathsf{WLP}(s, \texttt{true})$, $\phi' * \mathsf{body}_\mu(p)(\overline{e}) \in$ SATFORMULA, and $\phi' * p(\overline{e}) \in$ SFRMFORMULA.

By inversion on $\langle H, \rho_n, \pi_n\rangle \vDash_E \phi' * p(\overline{e})$, we get $\langle H, \rho_n, \pi_{n_1}\rangle \vDash_E \phi'$ and $\langle H, \rho_n, \pi_{n_2}\rangle \vDash_E p(\overline{e})$ for $\pi_{n_1} \uplus \pi_{n_2} = \pi_n$.

By inversion on $\langle H, \rho_n, \pi_{n_2}\rangle \vDash_E p(\overline{e})$, we get $\langle H, \rho_n, \pi_{n_2}\rangle \vDash_E \mathsf{body}_\mu(p)(\overline{e})$ and $\overline{H, \rho_n \vdash e \Downarrow v}$, and so EVSEPOP gives $\langle H, \rho_n, \pi_n\rangle \vDash_E \phi' * \mathsf{body}_\mu(p)(\overline{e})$.

Finally, since $\phi' * \mathsf{body}_\mu(p)(\overline{e}) \Rightarrow \mathsf{WLP}(s, \texttt{true})$, by lemma 6, we get $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{WLP}(s, \texttt{true}) = \mathsf{sWLP}_n(s \cdot \mathsf{nil}, \texttt{true})$.

In either case $\langle H, \rho_i, \pi_i\rangle \vDash_E \mathsf{sWLP}_i(s \cdot s_{n-1} \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ for all $1 \le i \le n$. Also, s is clearly $\texttt{skip}$ or s''; $\texttt{skip}$ for some $s'' \in$ STMT , and we are given that $s_i = s_i'$; $\texttt{skip}$ for some $s_i' \in$ STMT for all $1 \le i < n$. Additionally, $\pi_i \cap \pi_j = \emptyset$ for all $1 \le i \le n$, $1 \le j \le n$ such that $i \ne j$. Therefore, ψ' is a valid state.

2.3 Soundness Lemmas

Lemma 5 (Formula Footprint & Evaluation). *If $\langle\langle \lfloor\phi\rfloor_{H,\rho} \rangle\rangle_H$ is defined, $\langle\langle \lfloor\phi\rfloor_{H,\rho} \rangle\rangle_H = \pi'$, and $\langle H, \rho, \pi\rangle \vDash_E \phi$, then $\langle H, \rho, \pi'\rangle \vDash_E \phi$.*

Proof. Let $\phi \in$ FORMULA, $\langle H, \rho, \pi\rangle \in$ MEM, and $\pi' \in$ DYNFPRINT, such that $\langle\langle \lfloor\phi\rfloor_{H,\rho} \rangle\rangle_H$ is defined, $\langle\langle \lfloor\phi\rfloor_{H,\rho} \rangle\rangle_H = \pi'$, and $\langle H, \rho, \pi\rangle \vDash_E \phi$.

We will show by structural induction on the derivation of $\langle H, \rho, \pi\rangle \vDash_E \phi$ that $\langle H, \rho, \pi'\rangle \vDash_E \phi$.

Case 29 (BC: EVTRUEEXPR*).* We have $\langle H, \rho, \pi\rangle \vDash_E \texttt{true}$ and so $\phi = \texttt{true}$. By axiom rule EVTRUEEXPR, $\langle H, \rho, \pi'\rangle \vDash_E \texttt{true}$.

Case 30 (IC: EVCOMPEXPR*).* We have $\langle H, \rho, \pi\rangle \vDash_E e_1 \odot e_2$, $H, \rho \vdash e_1 \odot e_2 \Downarrow \texttt{true}$, and $\phi = e_1 \odot e_2$. Then, by EVCOMPEXPR, we get $\langle H, \rho, \pi'\rangle \vDash_E e_1 \odot e_2$.

Case 31 (IC: EVACC*).* We have $\langle H, \rho, \pi\rangle \vDash_E \textbf{acc}(\texttt{e.f})$, $H, \rho \vdash e \Downarrow o$, $H, \rho \vdash e.f \Downarrow v$, $\langle o, f\rangle \in \pi$, and $\phi = \textbf{acc}(\texttt{e.f})$.

Then, $\langle o, f\rangle \in \{\langle o, f\rangle\} = \langle\langle \{\langle o, f\rangle\} \rangle\rangle_H = \langle\langle \lfloor \textbf{acc}(\texttt{e.f})\rfloor_{H,\rho} \rangle\rangle_H = \pi'$.

Thus, by EVACC, we get $\langle H, \rho, \pi'\rangle \vDash_E \textbf{acc}(\texttt{e.f})$.

Case 32 (IC: EVPRED*).* We have $\langle H, \rho, \pi\rangle \vDash_E p(e_1, ..., e_n)$, $\langle H, \rho, \pi\rangle \vDash_E \mathsf{body}_\mu(p)(e_1, ..., e_n)$, and $\overline{H, \rho \vdash e \Downarrow v}$, and $\phi = p(e_1, ..., e_n)$.

Then, $\langle\langle \lfloor p(e_1, ..., e_n)\rfloor_{H,\rho} \rangle\rangle_H$ is defined, and so by definition, $\lfloor p(e_1, ..., e_n)\rfloor_{H,\rho} = \{\langle p, v_1, ..., v_n\rangle\}$. And so, $\langle\langle \lfloor \mathsf{body}_\mu(p)(e_1, ..., e_n)\rfloor_{H,\rho} \rangle\rangle_H = \langle\langle \lfloor \mathsf{body}_\mu(p)(v_1, ..., v_n)\rfloor_{H,[]} \rangle\rangle_H = \langle\langle \{\langle p, v_1, ..., v_n\rangle\} \rangle\rangle_H = \langle\langle \lfloor p(e_1, ..., e_n)\rfloor_{H,\rho} \rangle\rangle_H = \pi'$.

Therefore, by the IH on $\langle H, \rho, \pi\rangle \vDash_E \mathsf{body}_\mu(p)(e_1, ..., e_n)$, we get $\langle H, \rho, \pi'\rangle \vDash_E \mathsf{body}_\mu(p)(e_1, ..., e_n)$.

Finally, by EVPRED, $\langle H, \rho, \pi'\rangle \vDash_E p(e_1, ..., e_n)$.

Case 33 (IC: EVANDOP*).* We have $\langle H, \rho, \pi\rangle \vDash_E \phi_1 \wedge \phi_2, \langle H, \rho, \pi\rangle \vDash_E \phi_1, \langle H, \rho, \pi\rangle \vDash_E \phi_2$, and $\phi = \phi_1 \wedge \phi_2$.

Since $\langle\langle\ \lfloor \phi_1 \wedge \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H$ is defined, we get that $\lfloor \phi_1 \wedge \phi_2 \rfloor_{H,\rho}$ is defined, and so $\lfloor \phi_1 \rfloor_{H,\rho}$ and $\lfloor \phi_2 \rfloor_{H,\rho}$ are defined. Also, by definition, $\lfloor \phi_1 \rfloor_{H,\rho} \subseteq \lfloor \phi_1 \wedge \phi_2 \rfloor_{H,\rho}$ and $\lfloor \phi_2 \rfloor_{H,\rho} \subseteq \lfloor \phi_1 \wedge \phi_2 \rfloor_{H,\rho}$.

Therefore, by lemma 4, we get that $\langle\langle\ \lfloor \phi_1 \rfloor_{H,\rho}\ \rangle\rangle_H$ and $\langle\langle\ \lfloor \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H$ are defined and $\langle\langle\ \lfloor \phi_1 \rfloor_{H,\rho}\ \rangle\rangle_H \subseteq \langle\langle\ \lfloor \phi_1 \wedge \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H = \pi'$ and $\langle\langle\ \lfloor \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H \subseteq \langle\langle\ \lfloor \phi_1 \wedge \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H = \pi'$.

By the IH on $\langle H, \rho, \pi\rangle \vDash_E \phi_1$ and $\langle H, \rho, \pi\rangle \vDash_E \phi_2$, we get $\langle H, \rho, \langle\langle\ \lfloor \phi_1 \rfloor_{H,\rho}\ \rangle\rangle_H\rangle \vDash_E \phi_1$ and $\langle H, \rho, \langle\langle\ \lfloor \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H\rangle \vDash_E \phi_2$.

Therefore, by lemma 1, we get $\langle H, \rho, \pi'\rangle \vDash_E \phi_1$ and $\langle H, \rho, \pi'\rangle \vDash_E \phi_2$

Then, by EVANDOP, we get $\langle H, \rho, \pi'\rangle \vDash_E \phi_1 \wedge \phi_2$.

Case 34 (IC: EVSEPOP*).* We have $\langle H, \rho, \pi_1 \uplus \pi_2\rangle \vDash_E \phi_1 * \phi_2$, $\langle H, \rho, \pi_1\rangle \vDash_E \phi_1$, $\langle H, \rho, \pi_2\rangle \vDash_E \phi_2$, $\phi = \phi_1 * \phi_2$, and $\pi = \pi_1 \uplus \pi_2$.

Since $\langle\langle\ \lfloor \phi_1 * \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H$ is defined, we get that $\lfloor \phi_1 * \phi_2 \rfloor_{H,\rho}$ is defined, and so $\lfloor \phi_1 \rfloor_{H,\rho}$ and $\lfloor \phi_2 \rfloor_{H,\rho}$ are defined. Also, by definition, $\lfloor \phi_1 \rfloor_{H,\rho} \subseteq \lfloor \phi_1 * \phi_2 \rfloor_{H,\rho}$ and $\lfloor \phi_2 \rfloor_{H,\rho} \subseteq \lfloor \phi_1 * \phi_2 \rfloor_{H,\rho}$.

Therefore, by lemma 4, we get that $\langle\langle\ \lfloor \phi_1 \rfloor_{H,\rho}\ \rangle\rangle_H$ and $\langle\langle\ \lfloor \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H$ are defined and $\langle\langle\ \lfloor \phi_1 \rfloor_{H,\rho}\ \rangle\rangle_H \subseteq \langle\langle\ \lfloor \phi_1 * \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H = \pi'$ and $\langle\langle\ \lfloor \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H \subseteq \langle\langle\ \lfloor \phi_1 * \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H = \pi'$.

Also, since $\langle H, \rho, \pi_1\rangle \vDash_E \phi_1$ and $\langle H, \rho, \pi_2\rangle \vDash_E \phi_2$, by lemma 8, we get $\exists\, \Pi_1 \,.\, \langle H, \rho, \Pi_1\rangle$ is a good iso-state, $\langle\langle\ \Pi_1\ \rangle\rangle_H = \pi_1$ and $\langle H, \rho, \Pi_1\rangle \vDash_I \phi_1$, and $\exists\, \Pi_2 \,.\, \langle H, \rho, \Pi_2\rangle$ is a good iso-state, $\langle\langle\ \Pi_2\ \rangle\rangle_H = \pi_2$ and $\langle H, \rho, \Pi_2\rangle \vDash_I \phi_2$ respectively.

Then, by definition $\lfloor \phi_1 \rfloor_{H,\rho} \subseteq \Pi_1$ and $\lfloor \phi_2 \rfloor_{H,\rho} \subseteq \Pi_2$. And so, by lemma 4, we get that $\langle\langle\ \lfloor \phi_1 \rfloor_{H,\rho}\ \rangle\rangle_H$ and $\langle\langle\ \lfloor \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H$ are defined and $\langle\langle\ \lfloor \phi_1 \rfloor_{H,\rho}\ \rangle\rangle_H \subseteq \langle\langle\ \Pi_1\ \rangle\rangle_H = \pi_1$ and $\langle\langle\ \lfloor \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H \subseteq \langle\langle\ \Pi_2\ \rangle\rangle_H = \pi_2$.

Since $\pi_1 \uplus \pi_2$, we get $\langle\langle\ \lfloor \phi_1 \rfloor_{H,\rho}\ \rangle\rangle_H \uplus \langle\langle\ \lfloor \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H$.

Now, by the IH on $\langle H, \rho, \pi\rangle \vDash_E \phi_1$ and $\langle H, \rho, \pi\rangle \vDash_E \phi_2$, we get $\langle H, \rho, \langle\langle\ \lfloor \phi_1 \rfloor_{H,\rho}\ \rangle\rangle_H\rangle \vDash_E \phi_1$ and $\langle H, \rho, \langle\langle\ \lfloor \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H\rangle \vDash_E \phi_2$.

Then, by EVSEPOP, we get $\langle H, \rho, \langle\langle\ \lfloor \phi_1 \rfloor_{H,\rho}\ \rangle\rangle_H \uplus \langle\langle\ \lfloor \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H\rangle \vDash_E \phi_1 * \phi_2$.

Since $\langle\langle\ \lfloor \phi_1 \rfloor_{H,\rho}\ \rangle\rangle_H \subseteq \pi'$ and $\langle\langle\ \lfloor \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H \subseteq \pi'$, we get $\langle\langle\ \lfloor \phi_1 \rfloor_{H,\rho}\ \rangle\rangle_H \uplus \langle\langle\ \lfloor \phi_2 \rfloor_{H,\rho}\ \rangle\rangle_H \subseteq \pi'$.

Therefore, by lemma 1, we get $\langle H, \rho, \pi'\rangle \vDash_E \phi_1 * \phi_2$.

Case 35 (IC: EVCONDTRUE*).* We have $\langle H, \rho, \pi\rangle \vDash_E (\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F), \langle H, \rho, \pi\rangle \vDash_E \phi_T$, $H, \rho \vdash e \Downarrow \texttt{true}$, and $\phi = (\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F)$.

Since $\langle\langle\ \lfloor (\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F) \rfloor_{H,\rho}\ \rangle\rangle_H$ is defined, we get that $\lfloor (\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F) \rfloor_{H,\rho}$ is defined. Then, since $H, \rho \vdash e \Downarrow \texttt{true}$, by definition, $\lfloor (\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F) \rfloor_{H,\rho} = \lfloor \phi_T \rfloor_{H,\rho}$. And so, $\langle\langle\ \lfloor \phi_T \rfloor_{H,\rho}\ \rangle\rangle_H = \langle\langle\ \lfloor (\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F) \rfloor_{H,\rho}\ \rangle\rangle_H = \pi'$.

Therefore, by the IH on $\langle H, \rho, \pi\rangle \vDash_E \phi_T$, we get $\langle H, \rho, \pi'\rangle \vDash_E \phi_T$.

Finally, by EVCONDTRUE, we get $\langle H, \rho, \pi'\rangle \vDash_E (\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F)$.

Case 36 (IC: EVCONDFALSE*).* We have $\langle H, \rho, \pi\rangle \vDash_E (\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F), \langle H, \rho, \pi\rangle \vDash_E \phi_F$, $H, \rho \vdash e \Downarrow \texttt{false}$, and $\phi = (\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F)$.

Since $\langle\langle\ \lfloor(\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F)\rfloor_{H,\rho}\ \rangle\rangle_H$ is defined, we get that $\lfloor(\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F)\rfloor_{H,\rho}$ is defined. Then, since $H,\rho \vdash e \Downarrow \texttt{false}$, by definition, $\lfloor(\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F)\rfloor_{H,\rho} = \lfloor\phi_F\rfloor_{H,\rho}$. And so, $\langle\langle\ \lfloor\phi_F\rfloor_{H,\rho}\ \rangle\rangle_H = \langle\langle\ \lfloor(\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F)\rfloor_{H,\rho}\ \rangle\rangle_H = \pi'$.

Therefore, by the IH on $\langle H,\rho,\pi\rangle \vDash_E \phi_F$, we get $\langle H,\rho,\pi'\rangle \vDash_E \phi_F$.

Finally, by EvCondFalse, we get $\langle H,\rho,\pi'\rangle \vDash_E (\texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F)$.

Case 37 (IC: EvUnfolding*).* We have $\langle H,\rho,\pi\rangle \vDash_E (\texttt{unfolding}\ p(\overline{e})\ \texttt{in}\ \phi')$, $\langle H,\rho,\pi\rangle \vDash_E \phi'$, and $\phi = (\texttt{unfolding}\ p(\overline{e})\ \texttt{in}\ \phi')$.

Since $\langle\langle\ \lfloor(\texttt{unfolding}\ p(\overline{e})\ \texttt{in}\ \phi')\rfloor_{H,\rho}\ \rangle\rangle_H$ is defined, we get that $\lfloor(\texttt{unfolding}\ p(\overline{e})\ \texttt{in}\ \phi')\rfloor_{H,\rho}$ is defined. Then, by definition, $\lfloor(\texttt{unfolding}\ p(\overline{e})\ \texttt{in}\ \phi')\rfloor_{H,\rho} = \lfloor\phi'\rfloor_{H,\rho}$. And so, $\langle\langle\ \lfloor\phi'\rfloor_{H,\rho}\ \rangle\rangle_H = \langle\langle\ \lfloor(\texttt{unfolding}\ p(\overline{e})\ \texttt{in}\ \phi')\rfloor_{H,\rho}\ \rangle\rangle_H = \pi'$.

Therefore, by the IH on $\langle H,\rho,\pi\rangle \vDash_E \phi'$, we get $\langle H,\rho,\pi'\rangle \vDash_E \phi'$.

Finally, by EvUnfolding, we get $\langle H,\rho,\pi'\rangle \vDash_E (\texttt{unfolding}\ p(\overline{e})\ \texttt{in}\ \phi')$.

Lemma 6 (Formula Implication & Evaluation). *$\forall\ \phi,\ \phi' \in$ Formula, $\langle H,\rho,\pi\rangle \in$ Mem. $\phi \Rightarrow \phi'$ and $\langle H,\rho,\pi\rangle \vDash_E \phi \implies \langle H,\rho,\pi\rangle \vDash_E \phi'$.*

Proof. Let $\phi,\ \phi' \in$ Formula, $\langle H,\rho,\pi\rangle \in$ Mem. $\phi \Rightarrow \phi'$ and $\langle H,\rho,\pi\rangle \vDash_E \phi$.

Since $\langle H,\rho,\pi\rangle \vDash_E \phi$, by lemma 8 we get that $\exists\ \Pi$. $\langle H,\rho,\Pi\rangle$ is a good iso-state, $\langle\langle\ \Pi\ \rangle\rangle_H = \pi$ and $\langle H,\rho,\Pi\rangle \vDash_I \phi$.

Then $\langle H,\rho,\Pi\rangle \in [\![\phi]\!]$ by the definition of $[\![\cdot]\!]$. $\phi \Rightarrow \phi'$ gives $\langle H,\rho,\Pi\rangle \in [\![\phi]\!] \subseteq [\![\phi']\!]$. Therefore, by the definition of $[\![\cdot]\!]$, we get $\langle H,\rho,\Pi\rangle \vDash_I \phi'$.

Then by theorem 1, $\langle H,\rho,\langle\langle\ \Pi\ \rangle\rangle_H\rangle \vDash_E \phi'$, ie. $\langle H,\rho,\pi\rangle \vDash_E \phi'$.

Definition 1 (Good Iso-state). *An iso-recursive state defined by heap H, variable environment ρ, and permissions Π, is good if:*

1. *$\langle\langle\ \Pi\ \rangle\rangle_H$ is defined*
2. *$\forall\ \langle p, v_1, ..., v_n\rangle \in \Pi$. $\langle H,\rho,\langle\langle\ \Pi\ \rangle\rangle_H\rangle \vDash_E p(v_1,...,v_n)$*
3. *$\{(o,f)\ |\ (o,f) \in \Pi\} \cup \langle\langle\ \Pi^*\ \rangle\rangle_H$ exists and is equal to $\langle\langle\ \Pi\ \rangle\rangle_H$, where $\Pi^* = \biguplus_{\langle p,v_1,\ldots,v_n\rangle\in\Pi}\{\langle o,f\rangle\ |\ \langle o,f\rangle \in \lfloor\mathsf{body}_\mu(p)(v_1,\ldots,v_n)\rfloor\} \uplus \bigcup_{\langle p,v_1,\ldots,v_n\rangle\in\Pi}\{\langle p,v_1,\ldots,v_n\rangle\ |\ \langle p,v_1,\ldots,v_n\rangle \in \lfloor\mathsf{body}_\mu(p)(v_1,\ldots,v_n)\rfloor\}$*

Lemma 7 (sWLP^{θ_f} **Minimal Implication Closure).** *If $\theta_s \Rightarrow \theta_f * \theta$, $\theta_s \Rightarrow \theta_f * \theta'$, and $\theta,\ \theta',\ \theta_s \in$ SatFormula, then **1)** $\theta \wedge \theta' \in$ SatFormula, **2)** $\theta_s \Rightarrow \theta_f * (\theta\ \wedge\ \theta')$, **3)** $\theta\ \wedge\ \theta' \Rightarrow \theta$, and **4)** $\theta\ \wedge\ \theta' \Rightarrow \theta'$.*

Proof. Let $\theta,\ \theta',\ \theta_s,\ \theta_f \in$ Formula such that $\theta_s \Rightarrow \theta_f * \theta$, $\theta_s \Rightarrow \theta_f * \theta'$, and $\theta,\ \theta',\ \theta_s \in$ SatFormula.

Since $\theta_s \in$ SatFormula, $\theta_s \Rightarrow \theta_f * \theta$, and $\theta_s \Rightarrow \theta_f * \theta'$, we get that $\emptyset \neq [\![\theta_s]\!] \subseteq [\![\theta_f * \theta]\!]$ and $\emptyset \neq [\![\theta_s]\!] \subseteq [\![\theta_f * \theta']\!]$. Then let $\langle H,\rho,\Pi\rangle \in [\![\theta_s]\!]$ and so $\langle H,\rho,\Pi\rangle \in [\![\theta_f * \theta]\!]$ and $\langle H,\rho,\Pi\rangle \in [\![\theta_f * \theta']\!]$.

Therefore, $\langle H,\rho,\Pi\rangle \vDash_I \theta_f * \theta$ and $\langle H,\rho,\Pi\rangle \vDash_I \theta_f * \theta'$. By inversion on $\langle H,\rho,\Pi\rangle \vDash_I \theta_f * \theta$ and $\langle H,\rho,\Pi\rangle \vDash_I \theta_f * \theta'$, we get $\langle H,\rho,\Pi_1\rangle \vDash_I \theta_f$, $\langle H,\rho,\Pi_2\rangle \vDash_I \theta$, $\langle H,\rho,\Pi_3\rangle \vDash_I \theta_f$, and $\langle H,\rho,\Pi_4\rangle \vDash_I \theta'$ where $\Pi_1 \uplus \Pi_2 = \Pi$ and $\Pi_3 \uplus \Pi_4 = \Pi$.

1) Therefore, since $\Pi_2 \subseteq \Pi$, $\langle H, \rho, \Pi_2 \rangle \vDash_I \theta$, $\Pi_4 \subseteq \Pi$, and $\langle H, \rho, \Pi_4 \rangle \vDash_I \theta'$, we get by lemma 2 that $\langle H, \rho, \Pi \rangle \vDash_I \theta$ and $\langle H, \rho, \Pi \rangle \vDash_I \theta'$. Then by EvAndOp we get $\langle H, \rho, \Pi \rangle \vDash_I \theta \wedge \theta'$.

Then, $\langle H, \rho, \Pi \rangle \in [\![\theta \wedge \theta']\!]$ and so $\emptyset \neq [\![\theta \wedge \theta']\!]$ and $\theta \wedge \theta' \in$ SatFormula.

2) Then, since $\langle H, \rho, \Pi_1 \rangle \vDash_I \theta_f$ and $\langle H, \rho, \Pi_3 \rangle \vDash_I \theta_f$, by definition $\lfloor \theta_f \rfloor_{H,\rho}$ is defined. By definition, $\lfloor \theta_f \rfloor_{H,\rho} \subseteq \Pi_1$ and $\lfloor \theta_f \rfloor_{H,\rho} \subseteq \Pi_3$.

Therefore, $\lfloor \theta_f \rfloor_{H,\rho} \cap \Pi_2 = \emptyset$ and $\lfloor \theta_f \rfloor_{H,\rho} \cap \Pi_4 = \emptyset$. So, $\lfloor \theta_f \rfloor_{H,\rho} \cap (\Pi_2 \cup \Pi_4) = \emptyset$.

Also, since $\Pi_2 \subseteq \Pi_2 \cup \Pi_4$, $\langle H, \rho, \Pi_2 \rangle \vDash_I \theta$, $\Pi_4 \subseteq \Pi_2 \cup \Pi_4$, and $\langle H, \rho, \Pi_4 \rangle \vDash_I \theta'$, lemma 2 gives $\langle H, \rho, \Pi_2 \cup \Pi_4 \rangle \vDash_I \theta$ and $\langle H, \rho, \Pi_2 \cup \Pi_4 \rangle \vDash_I \theta'$. Then, by EvAndOp $\langle H, \rho, \Pi_2 \cup \Pi_4 \rangle \vDash_I \theta \wedge \theta'$.

Since $\langle H, \rho, \lfloor \theta_f \rfloor_{H,\rho} \rangle \vDash_I \theta_f$ by def of $\lfloor \theta_f \rfloor_{H,\rho}$, $\langle H, \rho, \Pi_2 \cup \Pi_4 \rangle \vDash_I \theta \wedge \theta'$, and $\lfloor \theta_f \rfloor_{H,\rho} \cap (\Pi_2 \cup \Pi_4) = \emptyset$, by EvSepOp we get that $\langle H, \rho, \lfloor \theta_f \rfloor_{H,\rho} \uplus (\Pi_2 \cup \Pi_4) \rangle \vDash_I \theta_f * (\theta \wedge \theta')$.

Then, since $\lfloor \theta_f \rfloor_{H,\rho} \uplus (\Pi_2 \cup \Pi_4) \subseteq \Pi$, by lemma 2 we get $\langle H, \rho, \Pi \rangle \vDash_I \theta_f * (\theta \wedge \theta')$ and $\langle H, \rho, \Pi \rangle \in [\![\theta_f * (\theta \wedge \theta')]\!]$. Therefore, $[\![\theta_s]\!] \subseteq [\![\theta_f * (\theta \wedge \theta')]\!]$ and so $\theta_s \Rightarrow \theta_f * (\theta \wedge \theta')$.

3) & 4) By **1)** we get that $\theta \wedge \theta' \in$ SatFormula. Therefore, $[\![\theta \wedge \theta']\!] \neq \emptyset$ and so let $\langle H', \rho', \Pi' \rangle \in [\![\theta \wedge \theta']\!]$. Then, $\langle H', \rho', \Pi' \rangle \vDash_I \theta \wedge \theta'$.

By inversion on $\langle H', \rho', \Pi' \rangle \vDash_I \theta \wedge \theta'$, we get $\langle H', \rho', \Pi' \rangle \vDash_I \theta$ and $\langle H', \rho', \Pi' \rangle \vDash_I \theta'$.

Therefore, $\langle H', \rho', \Pi' \rangle \in [\![\theta]\!]$ and $\langle H', \rho', \Pi' \rangle \in [\![\theta']\!]$. This gives $[\![\theta \wedge \theta']\!] \subseteq [\![\theta]\!]$ and $[\![\theta \wedge \theta']\!] \subseteq [\![\theta']\!]$. So, finally, $\theta \wedge \theta' \Rightarrow \theta$ and $\theta \wedge \theta' \Rightarrow \theta'$.

Lemma 8 (Summers). *If $\langle H, \rho, \pi \rangle \vDash_E \phi$, then $\exists\, \Pi \,.\, \langle H, \rho, \Pi \rangle$ is a good iso-state, $\langle\langle\, \Pi \,\rangle\rangle_H = \pi$ and $\langle H, \rho, \Pi \rangle \vDash_I \phi$.*

Proof. The proof of this lemma can be found here: extra-proofs/extra-proofs.pdf.

Theorem 1 (Summers). *If $\langle H, \rho, \Pi \rangle \vDash_I \phi$ and $\langle H, \rho, \Pi \rangle$ is a good iso-state, then $\langle H, \rho, \langle\langle\, \Pi \,\rangle\rangle_H \rangle \vDash_E \phi$.*

Proof. The proof of this theorem can be found here: extra-proofs/extra-proofs.pdf.

3 Consistent Formula Evaluation

Claim (Consistent Formula Evaluation). $\cdot \mathrel{\widetilde{\vDash}} \cdot$ is a consistent lifting of $\cdot \vDash_E \cdot$.

$\forall\, \widetilde{\phi} \in \widetilde{\text{FORMULA}}, \widetilde{\mathsf{body}_\Delta} \in \text{PREDNAME} \to \text{EXPR}^* \to \widetilde{\text{FORMULA}}, \langle H, \rho, \pi \rangle \in \text{MEM} \,.\, \langle H, \rho, \pi \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}, \widetilde{\mathsf{body}_\Delta} \rangle \iff \exists\, \langle \theta, \mathsf{body}_\Delta \rangle \in \gamma(\langle \widetilde{\phi}, \widetilde{\mathsf{body}_\Delta} \rangle) \,.\, \langle H, \rho, \pi \rangle \vDash_E \langle \theta, \mathsf{body}_\Delta \rangle$

Proof. The proof of this claim can be found here: extra-proofs/extra-proofs.pdf.

Definition 2 (Body Context Implication). *Let* $\mathsf{body}_\Delta, \mathsf{body}_\Delta' \in \text{PREDNAME} \to \text{EXPR}^* \to \text{FORMULA}$. *Then* $\mathsf{body}_\Delta \Rightarrow \mathsf{body}_\Delta'$ *if and only if* $\mathsf{dom}(\mathsf{body}_\Delta) = \mathsf{dom}(\mathsf{body}_\Delta')$, $\forall p \in \mathsf{dom}(\mathsf{body}_\Delta) \,.\, \mathsf{dom}(\mathsf{body}_\Delta(p)) = \mathsf{dom}(\mathsf{body}_\Delta'(p))$, *and* $\forall p \in \mathsf{dom}(\mathsf{body}_\Delta) \,.\, \forall \overline{e} \in \mathsf{dom}(\mathsf{body}_\Delta(p)) \,.\, \mathsf{body}_\Delta(p)(\overline{e}) \Rightarrow \mathsf{body}_\Delta'(p)(\overline{e})$.

Lemma 9 (Body Context Implication & Formula Evaluation). $\forall\, \phi \in \text{FORMULA}$, $\mathsf{body}_\Delta, \mathsf{body}_\Delta' \in \text{PREDNAME} \to \text{EXPR}^* \to \text{FORMULA}, \langle H, \rho, \pi \rangle \in \text{MEM}.\ \mathsf{body}_\Delta \Rightarrow \mathsf{body}_\Delta'$ *and* $\langle H, \rho, \pi \rangle \vDash_E \langle \phi, \mathsf{body}_\Delta \rangle$

$\implies \langle H, \rho, \pi \rangle \vDash_E \langle \phi, \mathsf{body}_\Delta' \rangle$.

Proof. Let $\phi \in$ FORMULA, body_Δ, $\mathsf{body}_\Delta{}' \in$ PREDNAME $\to$ EXPR* $\to$ FORMULA, $\langle H, \rho, \pi \rangle \in$ MEM. $\mathsf{body}_\Delta \Rightarrow \mathsf{body}_\Delta{}'$ and $\langle H, \rho, \pi \rangle \vDash_E \langle \phi, \mathsf{body}_\Delta \rangle$.

Using structural induction on the derivation of $\langle H, \rho, \pi \rangle \vDash_E \langle \phi, \mathsf{body}_\Delta \rangle$, we will show $\langle H, \rho, \pi \rangle \vDash_E \langle \phi, \mathsf{body}_\Delta{}' \rangle$.

Case 38 (Base Case: EVTRUEEXPR*).* We have $\langle H, \rho, \pi \rangle \vDash_E \langle \mathtt{true}, \mathsf{body}_\Delta \rangle$ and so $\phi = \mathtt{true}$. By axiom rule EVTRUEEXPR, we get $\langle H, \rho, \pi \rangle \vDash_E \langle \mathtt{true}, \mathsf{body}_\Delta{}' \rangle$.

Case 39 (IC: EVCOMPEXPR*).* We have $\langle H, \rho, \pi \rangle \vDash_E \langle e_1 \odot e_2, \mathsf{body}_\Delta \rangle$, $H, \rho \vdash e_1 \odot e_2 \Downarrow \mathtt{true}$, and $\phi = e_1 \odot e_2$. Then, by EVCOMPEXPR, we get $\langle H, \rho, \pi \rangle \vDash_E \langle e_1 \odot e_2, \mathsf{body}_\Delta{}' \rangle$.

Case 40 (IC: EVACC*).* We have $\langle H, \rho, \pi \rangle \vDash_E \langle \mathbf{acc}(\mathtt{e.f}), \mathsf{body}_\Delta \rangle$, $H, \rho \vdash e \Downarrow o$, $H, \rho \vdash e.f \Downarrow v$, $\langle o, f \rangle \in \pi$, and $\phi = \mathbf{acc}(\mathtt{e.f})$. Then, by EVACC, we get $\langle H, \rho, \pi \rangle \vDash_E \langle \mathbf{acc}(\mathtt{e.f}), \mathsf{body}_\Delta{}' \rangle$.

Case 41 (IC: EVPRED*).* We have $\langle H, \rho, \pi \rangle \vDash_E \langle p(\overline{e}), \mathsf{body}_\Delta \rangle$, $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{body}_\Delta(p)(\overline{e}), \mathsf{body}_\Delta \rangle$, and $\phi = p(\overline{e})$.

By the IH on $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{body}_\Delta(p)(\overline{e}), \mathsf{body}_\Delta \rangle$, we get $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{body}_\Delta(p)(\overline{e}), \mathsf{body}_\Delta{}' \rangle$.

Since $\mathsf{body}_\Delta \Rightarrow \mathsf{body}_\Delta{}'$ by definition 2, $\mathsf{body}_\Delta(p)(\overline{e}) \Rightarrow \mathsf{body}_\Delta{}'(p)(\overline{e})$. Then, by lemma 10, we get $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{body}_\Delta{}'(p)(\overline{e}), \mathsf{body}_\Delta{}' \rangle$.

Thus, by EVPRED, $\langle H, \rho, \pi \rangle \vDash_E \langle p(\overline{e}), \mathsf{body}_\Delta{}' \rangle$.

Case 42 (IC: EVANDOP*).* We have $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_1 \wedge \phi_2, \mathsf{body}_\Delta \rangle$, $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_1, \mathsf{body}_\Delta \rangle$, $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_2, \mathsf{body}_\Delta \rangle$, and $\phi = \phi_1 \wedge \phi_2$.

By the IH on both $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_1, \mathsf{body}_\Delta \rangle$ and $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_2, \mathsf{body}_\Delta \rangle$, we get $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_1, \mathsf{body}_\Delta{}' \rangle$ and $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_2, \mathsf{body}_\Delta{}' \rangle$.

Then, by EVANDOP, we get $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_1 \wedge \phi_2, \mathsf{body}_\Delta{}' \rangle$.

Case 43 (IC: EVSEPOP*).* We have $\langle H, \rho, \pi_1 \uplus \pi_2 \rangle \vDash_E \langle \phi_1 * \phi_2, \mathsf{body}_\Delta \rangle$, $\langle H, \rho, \pi_1 \rangle \vDash_E \langle \phi_1, \mathsf{body}_\Delta \rangle$, $\langle H, \rho, \pi_2 \rangle \vDash_E \langle \phi_2, \mathsf{body}_\Delta \rangle$, $\phi = \phi_1 * \phi_2$, and $\pi = \pi_1 \uplus \pi_2$.

By the IH on both $\langle H, \rho, \pi_1 \rangle \vDash_E \langle \phi_1, \mathsf{body}_\Delta \rangle$ and $\langle H, \rho, \pi_2 \rangle \vDash_E \langle \phi_2, \mathsf{body}_\Delta \rangle$, we get $\langle H, \rho, \pi_1 \rangle \vDash_E \langle \phi_1, \mathsf{body}_\Delta{}' \rangle$ and $\langle H, \rho, \pi_2 \rangle \vDash_E \langle \phi_2, \mathsf{body}_\Delta{}' \rangle$.

Then, by EVSEPOP, we get $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_1 * \phi_2, \mathsf{body}_\Delta{}' \rangle$.

Case 44 (IC: EVCONDTRUE*).* We have $\langle H, \rho, \pi \rangle \vDash_E \langle (\mathtt{if}\ e\ \mathtt{then}\ \phi_T\ \mathtt{else}\ \phi_F), \mathsf{body}_\Delta \rangle$, $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_T, \mathsf{body}_\Delta \rangle$, $H, \rho \vdash e \Downarrow \mathtt{true}$, and $\phi = (\mathtt{if}\ e\ \mathtt{then}\ \phi_T\ \mathtt{else}\ \phi_F)$.

By the IH on $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_T, \mathsf{body}_\Delta \rangle$, we get $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_T, \mathsf{body}_\Delta{}' \rangle$.

Then, by EVCONDTRUE, we get $\langle H, \rho, \pi \rangle \vDash_E \langle (\mathtt{if}\ e\ \mathtt{then}\ \phi_T\ \mathtt{else}\ \phi_F), \mathsf{body}_\Delta{}' \rangle$.

Case 45 (IC: EVCONDFALSE*).* We have $\langle H, \rho, \pi \rangle \vDash_E \langle (\mathtt{if}\ e\ \mathtt{then}\ \phi_T\ \mathtt{else}\ \phi_F), \mathsf{body}_\Delta \rangle$, $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_F, \mathsf{body}_\Delta \rangle$, $H, \rho \vdash e \Downarrow \mathtt{false}$, and $\phi = (\mathtt{if}\ e\ \mathtt{then}\ \phi_T\ \mathtt{else}\ \phi_F)$.

By the IH on $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_F, \mathsf{body}_\Delta \rangle$, we get $\langle H, \rho, \pi \rangle \vDash_E \langle \phi_F, \mathsf{body}_\Delta{}' \rangle$.

Then, by EVCONDFALSE, we get $\langle H, \rho, \pi \rangle \vDash_E \langle (\mathtt{if}\ e\ \mathtt{then}\ \phi_T\ \mathtt{else}\ \phi_F), \mathsf{body}_\Delta{}' \rangle$.

Case 46 (IC: EVUNFOLDING*).* We have $\langle H, \rho, \pi \rangle \vDash_E \langle (\mathtt{\mathbf{unfolding}}\ p(\overline{e})\ \mathtt{\mathbf{in}}\ \phi'), \mathsf{body}_\Delta \rangle$, $\langle H, \rho, \pi \rangle \vDash_E \langle \phi', \mathsf{body}_\Delta \rangle$, and $\phi = (\mathtt{\mathbf{unfolding}}\ p(\overline{e})\ \mathtt{\mathbf{in}}\ \phi')$.

By the IH on $\langle H, \rho, \pi \rangle \vDash_E \langle \phi', \mathsf{body}_\Delta \rangle$, we get $\langle H, \rho, \pi \rangle \vDash_E \langle \phi', \mathsf{body}_\Delta{}' \rangle$.

Then, by EVUNFOLDING, we get $\langle H, \rho, \pi \rangle \vDash_E \langle (\mathtt{\mathbf{unfolding}}\ p(\overline{e})\ \mathtt{\mathbf{in}}\ \phi'), \mathsf{body}_\Delta{}' \rangle$.

Lemma 10 (Formula Implication & Evaluation). $\forall\, \phi,\ \phi' \in \textsc{Formula},\ \mathsf{body}_\Delta \in \textsc{PredName} \rightarrow \textsc{Expr}^* \rightarrow \textsc{Formula}, \langle H, \rho, \pi\rangle \in \textsc{Mem}.\ \phi \Rightarrow \phi'$ *and* $\langle H, \rho, \pi\rangle \vDash_E \langle \phi, \mathsf{body}_\Delta\rangle$
$\implies \langle H, \rho, \pi\rangle \vDash_E \langle \phi', \mathsf{body}_\Delta\rangle$.

Proof. Let $\phi,\ \phi' \in \textsc{Formula}$, $\mathsf{body}_\Delta \in \textsc{PredName} \rightarrow \textsc{Expr}^* \rightarrow \textsc{Formula}$, and $\langle H, \rho, \pi\rangle \in \textsc{Mem}$. $\phi \Rightarrow \phi'$ and $\langle H, \rho, \pi\rangle \vDash_E \langle \phi, \mathsf{body}_\Delta\rangle$.

Since $\langle H, \rho, \pi\rangle \vDash_E \langle \phi, \mathsf{body}_\Delta\rangle$, by lemma 11 we get that $\exists\, \Pi\ .\ \langle H, \rho, \Pi, \mathsf{body}_\Delta\rangle$ is a good iso-state, $\langle\langle\, \langle \Pi, \mathsf{body}_\Delta\rangle\, \rangle\rangle_H = \pi$ and $\langle H, \rho, \Pi\rangle \vDash_I \phi$.

Then $\langle H, \rho, \Pi\rangle \in [\![\phi]\!]$ by the definition of $[\![\cdot]\!]$. $\phi \Rightarrow \phi'$ gives $\langle H, \rho, \Pi\rangle \in [\![\phi]\!] \subseteq [\![\phi']\!]$. Therefore, by the definition of $[\![\cdot]\!]$, we get $\langle H, \rho, \Pi\rangle \vDash_I \phi'$.

Then by theorem 2, $\langle H, \rho, \langle\langle\, \langle \Pi, \mathsf{body}_\Delta\rangle\, \rangle\rangle_H\rangle \vDash_E \langle \phi', \mathsf{body}_\Delta\rangle$, ie. $\langle H, \rho, \pi\rangle \vDash_E \langle \phi', \mathsf{body}_\Delta\rangle$.

Definition 3 (Good Iso-state w/ body). *An iso-recursive state defined by heap H, variable environment ρ, permissions Π, and concrete body context body_Δ, is good if:*

1. $\langle\langle\, \langle \Pi, \mathsf{body}_\Delta\rangle\, \rangle\rangle_H$ *is defined*
2. $\forall\, \langle p, v_1, ..., v_n\rangle \in \Pi\ .\ \langle H, \rho, \langle\langle\, \langle \Pi, \mathsf{body}_\Delta\rangle\, \rangle\rangle_H\rangle \vDash_E \langle p(v_1, ..., v_n), \mathsf{body}_\Delta\rangle$
3. $\{(o, f) \mid (o, f) \in \Pi\} \cup \langle\langle\, \langle \Pi^*, \mathsf{body}_\Delta\rangle\, \rangle\rangle_H$ *exists and is equal to* $\langle\langle\, \langle \Pi, \mathsf{body}_\Delta\rangle\, \rangle\rangle_H$, *where* $\Pi^* = \biguplus_{\langle p, v_1, \ldots, v_n\rangle \in \Pi} \{\langle o, f\rangle \mid \langle o, f\rangle \in \lfloor \mathsf{body}_\Delta(p)(v_1, \ldots, v_n)\rfloor\} \uplus \bigcup_{\langle p, v_1, \ldots, v_n\rangle \in \Pi} \{\langle p, v_1, \ldots, v_n\rangle \mid \langle p, v_1, \ldots, v_n\rangle \in \lfloor \mathsf{body}_\Delta(p)(v_1, \ldots, v_n)\rfloor\}$

Lemma 11 (Summers). *If* $\langle H, \rho, \pi\rangle \vDash_E \langle \phi, \mathsf{body}_\Delta\rangle$, *then* $\exists\, \Pi\ .\ \langle H, \rho, \Pi, \mathsf{body}_\Delta\rangle$ *is a good iso-state,* $\langle\langle\, \langle \Pi, \mathsf{body}_\Delta\rangle\, \rangle\rangle_H = \pi$ *and* $\langle H, \rho, \Pi\rangle \vDash_I \phi$.

Proof. The proof of this lemma can be found here: extra-proofs/extra-proofs.pdf.

Theorem 2 (Summers). *If* $\langle H, \rho, \Pi\rangle \vDash_I \phi$ *and* $\langle H, \rho, \Pi, \mathsf{body}_\Delta\rangle$ *is a good iso-state, then* $\langle H, \rho, \langle\langle\, \langle \Pi, \mathsf{body}_\Delta\rangle\, \rangle\rangle_H\rangle \vDash_E \langle \phi, \mathsf{body}_\Delta\rangle$.

Proof. The proof of this theorem can be found here: extra-proofs/extra-proofs.pdf.

4 Consistent Formula Implication

Claim (Consistent Formula Implication). $\cdot \mathrel{\widetilde{\Rightarrow}} \cdot$ is a consistent lifting of $\cdot \Rightarrow \cdot$.
$\forall\, \widetilde{\phi}_1,\ \widetilde{\phi}_2 \in \widetilde{\textsc{Formula}}\ .\ \widetilde{\phi}_1 \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi}_2 \iff \exists\, \theta_1 \in \gamma(\widetilde{\phi}_1),\ \theta_2 \in \gamma(\widetilde{\phi}_2)\ .\ \theta_1 \Rightarrow \theta_2$

Proof. Let $\widetilde{\phi}_1,\ \widetilde{\phi}_2 \in \widetilde{\textsc{Formula}}$.
$(\Longleftarrow)$ Suppose $\exists\, \theta_1 \in \gamma(\widetilde{\phi}_1),\ \theta_2 \in \gamma(\widetilde{\phi}_2)\ .\ \theta_1 \Rightarrow \theta_2$.

Case 47 ($\widetilde{\phi}_1 = \theta_1$). Since $\cdot \Rightarrow \cdot$ is reflexive and $\theta_2 \in \gamma(\widetilde{\phi}_2)$, by the definition of γ, we get $\theta_2 \Rightarrow \mathsf{static}(\widetilde{\phi}_2)$.

Then by the transitivity of $\cdot \Rightarrow \cdot$, we get $\theta_1 \Rightarrow \mathsf{static}(\widetilde{\phi}_2)$. Therefore, $\widetilde{\phi}_1 \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi}_2$ by $\widetilde{\textsc{ImplStatic}}$.

Case 48 ($\widetilde{\phi}_1 = ?*\phi_1$ and $\phi_1 \in \textsc{SatFormula}$). Since $\theta_1 \in \gamma(?*\phi_1)$, by the definition of γ, $\theta_1 \Rightarrow \phi_1$ and $\theta_1 \in \textsc{SatFormula}$.

Since $\cdot \Rightarrow \cdot$ is reflexive and $\theta_2 \in \gamma(\widetilde{\phi}_2)$, by the definition of γ, we get $\theta_2 \Rightarrow \mathsf{static}(\widetilde{\phi}_2)$. Then by the transitivity of $\cdot \Rightarrow \cdot$, we get $\theta_1 \Rightarrow \mathsf{static}(\widetilde{\phi}_2)$.

Therefore, $\widetilde{\phi}_1 \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi}_2$ by $\widetilde{\textsc{ImplGrad}}$.

Case 49 *($\widetilde{\phi}_1 = ? * \phi_1$ and $\phi_1 \notin$ SATFORMULA).* Contradiction, because $\theta_1 \in \gamma(\widetilde{\phi}_1) = undefined$.

($\Longrightarrow$) Suppose $\widetilde{\phi}_1 \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi}_2$.

Case 50 *($\widetilde{\phi}_1 = \theta_1$).* Then $\theta_1 \Rightarrow \mathsf{static}(\widetilde{\phi}_2)$ by inversion and $\theta_1 \in \gamma(\widetilde{\phi}_1)$.
If $\widetilde{\phi}_2 = \theta_2$,
then $\theta_1 \in \gamma(\widetilde{\phi}_1)$, $\theta_2 \in \gamma(\widetilde{\phi}_2)$, and $\theta_1 \Rightarrow \theta_2$.
If $\widetilde{\phi}_2 = ? * \phi_2$ and $\phi_2 \in$ SATFORMULA,
then if $\theta_1 \in$ SATFORMULA, $\theta_1 \in \gamma(\widetilde{\phi}_1)$, $\theta_1 \in \gamma(\widetilde{\phi}_2)$, and $\theta_1 \Rightarrow \theta_1$ (by reflexivity of $\cdot \Rightarrow \cdot$).
Otherwise, if $\theta_1 \notin$ SATFORMULA, $\theta_1 \in \gamma(\widetilde{\phi}_1)$ and $\theta_2 \in \gamma(\widetilde{\phi}_2)$. $\theta_1 \Rightarrow \theta_2$ ($[\![\theta_1]\!] = \emptyset \subseteq [\![\phi_2]\!]$), where θ_2 is created by adding the missing permissions of ϕ_2 to self-frame ϕ_2 to ϕ_2 via the non-separating conjunction.
If $\widetilde{\phi}_2 = ? * \phi_2$ and $\phi_2 \notin$ SATFORMULA,
then if $\theta_1 \in$ SATFORMULA, we have a contradiction. $\theta_1 \Rightarrow \mathsf{static}(\widetilde{\phi}_2)$ gives $[\![\theta_1]\!] \neq \emptyset \subseteq [\![\mathsf{static}(\widetilde{\phi}_2)]\!] = \emptyset$.
Otherwise, if $\theta_1 \notin$ SATFORMULA, this case is never used in any proof where it is necessary. Always, $\phi_2 \in$ SATFORMULA.

Case 51 *($\widetilde{\phi}_1 = ? * \phi_1$ and $\phi_1 \in$ SATFORMULA).* Then $\exists \,.\, \theta \in$ SATFORMULA, $\theta \Rightarrow \phi_1$, $\theta \Rightarrow \mathsf{static}(\widetilde{\phi}_2)$ by inversion. Also, $\theta \in \gamma(\widetilde{\phi}_1)$.
If $\widetilde{\phi}_2 = \theta_2$,
then $\theta \in \gamma(\widetilde{\phi}_1)$, $\theta_2 \in \gamma(\widetilde{\phi}_2)$, and $\theta_1 \Rightarrow \theta_2$.
If $\widetilde{\phi}_2 = ? * \phi_2$ and $\phi_2 \in$ SATFORMULA,
then $\theta \in \gamma(\widetilde{\phi}_1)$, $\theta \in \gamma(\widetilde{\phi}_2)$, and $\theta \Rightarrow \theta$ (by reflexivity of $\cdot \Rightarrow \cdot$).
If $\widetilde{\phi}_2 = ? * \phi_2$ and $\phi_2 \notin$ SATFORMULA,
then we have a contradiction. $\theta \Rightarrow \mathsf{static}(\widetilde{\phi}_2)$ gives $[\![\theta]\!] \neq \emptyset \subseteq [\![\mathsf{static}(\widetilde{\phi}_2)]\!] = \emptyset$.

Case 52 *($\widetilde{\phi}_1 = ? * \phi_1$ and $\phi_1 \notin$ SATFORMULA).* Then $\exists \,.\, \theta \in$ SATFORMULA, $\theta \Rightarrow \phi_1$, $\theta \Rightarrow \mathsf{static}(\widetilde{\phi}_2)$ by inversion. This results in a contradiction, because $\theta \Rightarrow \phi_1$ gives $[\![\theta]\!] \neq \emptyset \subseteq [\![\phi_1]\!] = \emptyset$.

5 GVL$_{\text{RP}}$ Soundness (Non-optimized Semantics)

5.1 Progress

Claim *(GVL$_{RP}$ Progress).* If $\psi \in$ STATE is a valid state and
$\psi \notin \{\langle H, \langle \rho, \pi, skip\rangle \cdot nil\rangle \mid H \in \text{HEAP}, \rho \in \text{ENV}, \pi \in \text{DYNFPRINT}\}$
then $\psi \mathrel{\widetilde{\longrightarrow}} \psi'$ for some $\psi' \in$ STATE or $\psi \mathrel{\widetilde{\longrightarrow}}$ **error**.

Proof. Suppose $\psi = \langle H, \langle \rho_n, \pi_n, s_n \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \in \textsc{State}$ such that ψ is a valid state and $\psi \notin \{\langle H, \langle \rho, \pi, skip \rangle \cdot nil \rangle \mid H \in \textsc{Heap}, \rho \in \textsc{Env}, \pi \in \textsc{DynFprint}\}$.

Since ψ is a valid state, by definition we get that $s_n = s_{n_1}$; `skip` for some $s_{n_1} \in$ $\textsc{Stmt}$ or $s_n = \texttt{skip}$ and $s_i = s_{i_1}$; s_{i_2} for some $s_{i_1}, s_{i_2} \in \textsc{Stmt}$ such that s_{i_1} is a method call or while loop statement for all $1 \leq i < n$.

If $s_n = \texttt{skip}$, then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip} \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

If $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip} \rangle \cdot \langle \rho_{n-1}, \pi_{n-1}, y := z.m(\overline{x})\ ;\ s_{n-1_2} \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$, then

If $\langle H, \rho_n, \pi_n \rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpost}(m), \mathsf{body}_\mu \rangle$ holds. Then by $\textsc{SsCallFinish}$,

$\psi \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho_{n-1}[y \mapsto \rho_n(\texttt{result})], \pi_n \cup \pi_{n-1}, s_{n-1_2} \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$.

If $\langle H, \rho_n, \pi_n \rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpost}(m), \mathsf{body}_\mu \rangle$ does not hold. Then by $\textsc{SsCallFinishError}$, $\psi \mathrel{\widetilde{\longrightarrow}}$ **error**.

Either way, $\psi \mathrel{\widetilde{\longrightarrow}} \psi'$ or $\psi \mathrel{\widetilde{\longrightarrow}}$ **error**.

If $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip} \rangle \cdot \langle \rho_{n-1}, \pi_{n-1}, \texttt{while (e) inv } \widetilde{\phi}\ \{\ \texttt{s}'\ \}\ ;\ s_{n-1_2} \rangle \cdot ... \cdot$ $\langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$, then by $\textsc{SsWhileFinish}$,

$\psi \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho_n, \pi_n \cup \pi_{n-1}, \texttt{while (e) inv } \widetilde{\phi}\ \{\ \texttt{s}'\ \}\ ;\ s_{n-1_2} \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$. Then clearly, $\psi \mathrel{\widetilde{\longrightarrow}} \psi'$ or $\psi \mathrel{\widetilde{\longrightarrow}}$ **error**.

If $\psi = \langle H, \langle \rho_n, \pi_n, skip \rangle \cdot nil \rangle$, then contradiction because we assumed $\psi \notin$ $\{\langle H', \langle \rho', \pi', skip \rangle \cdot nil \rangle \mid H' \in \textsc{Heap}, \rho' \in \textsc{Env}, \pi' \in \textsc{DynFprint}\}$.

If $s_n = s_{n_1}$; `skip` for some $s_{n_1} \in \textsc{Stmt}$, then by lemma 3, we get $s_n = s_h$; s_t such that s_h is not a sequence statement and $s_t = \texttt{skip}$ or s_{t_1} ; `skip` where $s_{n_1} = s_h$; s_{t_1}.

We will show $\psi \mathrel{\widetilde{\longrightarrow}} \psi'$ for some $\psi' \in \textsc{State}$ or $\psi \mathrel{\widetilde{\longrightarrow}}$ **error** by casing on s_h.

Case 53 ($s_h = \texttt{skip}$). Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip};\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$ and by $\textsc{SsSkip}$, $\langle H, \langle \rho_n, \pi_n, \texttt{skip};\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho_n, \pi_n, s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle =$ ψ'. Then clearly $\psi \mathrel{\widetilde{\longrightarrow}} \psi'$ or $\psi \mathrel{\widetilde{\longrightarrow}}$ **error**.

Case 54 ($s_h = \texttt{T x}$). Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{T x};\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$ and by $\textsc{SsDeclare}$, $\langle H, \langle \rho_n, \pi_n, \texttt{T x};\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho_n, \pi_n, s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle =$ ψ'. Then clearly $\psi \mathrel{\widetilde{\longrightarrow}} \psi'$ or $\psi \mathrel{\widetilde{\longrightarrow}}$ **error**.

Case 55 ($s_h = \texttt{assert } \phi$). Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{assert } \phi;\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

If $\langle H, \rho_n, \pi_n \rangle \mathrel{\widetilde{\vDash}} \langle ? * \phi, \mathsf{body}_\mu \rangle$ holds. Then by $\textsc{SsAssert}$,

$\langle H, \langle \rho_n, \pi_n, \texttt{assert } \phi;\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \mathrel{\widetilde{\longrightarrow}}$
$\langle H, \langle \rho_n, \pi_n, s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$.

If $\langle H, \rho_n, \pi_n \rangle \mathrel{\widetilde{\vDash}} \langle ? * \phi, \mathsf{body}_\mu \rangle$ does not hold. Then by $\textsc{SsAssertError}$, $\psi \mathrel{\widetilde{\longrightarrow}}$ **error**.
Either way, $\psi \mathrel{\widetilde{\longrightarrow}} \psi'$ or $\psi \mathrel{\widetilde{\longrightarrow}}$ **error**.

Case 56 ($s_h = \texttt{x.f := y}$). Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{x.f := y};\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

Also, we are operating over well-typed programs so $H, \rho_n \vdash y \Downarrow v$.

If $\langle H, \rho_n, \pi_n \rangle \vDash_E \textbf{acc}(\texttt{x.f})$ holds. Then by $\textsc{SsFAssign}$,

$\langle H, \langle \rho_n, \pi_n, \texttt{x.f := y};\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \mathrel{\widetilde{\longrightarrow}}$
$\langle H[o \mapsto [f \mapsto v]], \langle \rho_n, \pi_n, s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$.

If $\langle H, \rho_n, \pi_n \rangle \vDash_E$ **acc**(x.f) does not hold. Then by SsFAssignError, $\psi \widetilde{\longrightarrow}$ **error**.
Either way, $\psi \widetilde{\longrightarrow} \psi'$ or $\psi \widetilde{\longrightarrow}$ **error**.

Case 57 $(s_h = x := e)$. Then $\psi = \langle H, \langle \rho_n, \pi_n, x := e;\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

Since we are operating over well-typed programs, $H, \rho_n \vdash e \Downarrow v$.

If $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$ holds. Then by SsAssign,

$\langle H, \langle \rho_n, \pi_n, x := e;\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \widetilde{\longrightarrow}$
$\langle H, \langle \rho_n[x \mapsto v], \pi_n, s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$.

If $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$ does not hold. Then by SsAssignError, $\psi \widetilde{\longrightarrow}$ **error**.
Either way, $\psi \widetilde{\longrightarrow} \psi'$ or $\psi \widetilde{\longrightarrow}$ **error**.

Case 58 $(s_h = \texttt{x := new C})$. Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{x := new C};\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

Let $o \in$ Loc and $o \notin \mathsf{dom}(H)$.

Thus, by SsAlloc, $\langle H, \langle \rho_n, \pi_n, \texttt{x := new C};\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \widetilde{\longrightarrow}$
$\langle H', \langle \rho_n[x \mapsto o], \pi_n \cup \overline{\langle o, f_i \rangle}, s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$ where $\mathsf{fields}(C) = \overline{T_i\ \ f_i\,;}$ and $H' = H[o \mapsto [\overline{f_i \mapsto \mathsf{defaultValue}(T_i)}]]$.

Then clearly $\psi \widetilde{\longrightarrow} \psi'$ or $\psi \widetilde{\longrightarrow}$ **error**.

Case 59 $(s_h = \texttt{if (e) \{}\mathtt{s}_1'\texttt{\} else \{}\mathtt{s}_2'\texttt{\}})$. Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{if (e) \{}\mathtt{s}_1'\texttt{\} else \{}\mathtt{s}_2'\texttt{\}};\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$.

We operate over well-typed programs, so we have $H, \rho_n \vdash e \Downarrow v$ where $v \in \{\texttt{true}, \texttt{false}\}$.

If $H, \rho_n \vdash e \Downarrow \texttt{true}$ and $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$ holds, then by SsIfTrue,

$\langle H, \langle \rho_n, \pi_n, \texttt{if (e) \{}\mathtt{s}_1'\texttt{\} else \{}\mathtt{s}_2'\texttt{\}};\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \widetilde{\longrightarrow}$
$\langle H, \langle \rho_n, \pi_n, s_1';\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$.

If $H, \rho_n \vdash e \Downarrow \texttt{false}$ and $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$ holds, then by SsIfFalse,

$\langle H, \langle \rho_n, \pi_n, \texttt{if (e) \{}\mathtt{s}_1'\texttt{\} else \{}\mathtt{s}_2'\texttt{\}};\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \widetilde{\longrightarrow}$
$\langle H, \langle \rho_n, \pi_n, s_2';\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$.

If $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$ does not hold. Then by SsIfError, $\psi \widetilde{\longrightarrow}$ **error**.
In all cases, $\psi \widetilde{\longrightarrow} \psi'$ or $\psi \widetilde{\longrightarrow}$ **error**.

Case 60 $(s_h = y := z.m(\overline{x}))$.

The proof of this case can be found here: extra-proofs/gvlrp-progress-methodcall.pdf.

Case 61 $(s_h = \texttt{while (e) inv } \widetilde{\phi}_{\mathtt{i}}\ \texttt{\{ r \}})$.

The proof of this case can be found here: extra-proofs/gvlrp-progress-whileloop.pdf.

Case 62 $(s_h = \texttt{fold}\ p(\overline{e}))$. Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{fold}\ p(\overline{e});\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$ and by SsFold,

$\langle H, \langle \rho_n, \pi_n, \texttt{fold}\ p(\overline{e});\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$. Then clearly $\psi \widetilde{\longrightarrow} \psi'$ or $\psi \widetilde{\longrightarrow}$ **error**.

Case 63 $(s_h = \texttt{unfold}\ p(\overline{e}))$. Then $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{unfold}\ p(\overline{e});\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle$ and by SsUnfold,

$\langle H, \langle \rho_n, \pi_n, \texttt{unfold}\ p(\overline{e});\ s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \longrightarrow \langle H, \langle \rho_n, \pi_n, s_t \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'$. Then clearly $\psi \widetilde{\longrightarrow} \psi'$ or $\psi \widetilde{\longrightarrow}$ **error**.

5.2 Preservation

Claim (GVL$_{RP}$ Preservation). If ψ is a valid state and $\psi \xrightarrow{\sim} \psi'$ for some $\psi' \in$ STATE then ψ' is a valid state.

Proof. Suppose $\psi \in$ STATE such that ψ is a valid state and $\psi \xrightarrow{\sim} \psi'$ for some $\psi' \in$ STATE.

We will show ψ' is a valid state by case analysis on $\psi \xrightarrow{\sim} \psi'$.

*Case 64 (*SSSKIP*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip};\ s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim} \langle H, \langle \rho_n, \pi_n, s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$.

Since ψ is a valid state, by definition we get that

1) $\texttt{skip};\ s = s';\ \texttt{skip}$ for some $s' \in$ STMT,

2) $s_i = s_i';\ \texttt{skip}$ for some $s_i' \in$ STMT for all $1 \leq i < n$, and

3) $s_i = s_i^1;\ s_i^2$ for some $s_i^1,\ s_i^2 \in$ STMT where s_i^1 is a method call or while loop statement for all $1 \leq i < n$.

Then clearly,

a) s is clearly $\texttt{skip}$ or $s'';\ \texttt{skip}$ for some $s'' \in$ STMT.

Therefore, by a), 2), and 3) we get that ψ' is a valid state.

*Case 65 (*SSDECLARE*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{T x};\ s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim} \langle H, \langle \rho_n, \pi_n, s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$.

Since ψ is a valid state, by definition we get that

1) $\texttt{T x};\ s = s';\ \texttt{skip}$ for some $s' \in$ STMT,

2) $s_i = s_i';\ \texttt{skip}$ for some $s_i' \in$ STMT for all $1 \leq i < n$, and

3) $s_i = s_i^1;\ s_i^2$ for some $s_i^1,\ s_i^2 \in$ STMT where s_i^1 is a method call or while loop statement for all $1 \leq i < n$.

Then clearly,

a) s is clearly $\texttt{skip}$ or $s'';\ \texttt{skip}$ for some $s'' \in$ STMT.

Therefore, by a), 2), and 3) we get that ψ' is a valid state.

*Case 66 (*SSASSERT*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{assert}\ \phi';\ s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim} \langle H, \langle \rho_n, \pi_n, s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$ and $\langle H, \rho_n, \pi_n\rangle \mathrel{\widetilde{\vDash}} \langle ? * \phi', \mathsf{body}_\mu\rangle$.

Since ψ is a valid state, by definition we get that

1) $\texttt{assert}\ \phi';\ s = s';\ \texttt{skip}$ for some $s' \in$ STMT,

2) $s_i = s_i';\ \texttt{skip}$ for some $s_i' \in$ STMT for all $1 \leq i < n$, and

3) $s_i = s_i^1;\ s_i^2$ for some $s_i^1,\ s_i^2 \in$ STMT where s_i^1 is a method call or while loop statement for all $1 \leq i < n$.

Then clearly,

a) s is clearly $\texttt{skip}$ or $s'';\ \texttt{skip}$ for some $s'' \in$ STMT.

Therefore, by a), 2), and 3) we get that ψ' is a valid state.

*Case 67 (*SSFASSIGN*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{x.f := y};\ s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim} \langle H[o \mapsto [f \mapsto v]], \langle \rho_n, \pi_n, s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$, $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathbf{acc}(\texttt{x.f})$, and $H, \rho_n \vdash y \Downarrow v$.

Since ψ is a valid state, by definition we get that

1) `x.f := y`; $s = s'$; `skip` for some $s' \in$ STMT,

2) $s_i = s_i'$; `skip` for some $s_i' \in$ STMT for all $1 \le i < n$, and

3) $s_i = s_i^1$; s_i^2 for some s_i^1, $s_i^2 \in$ STMT where s_i^1 is a method call or while loop statement for all $1 \le i < n$.

Then clearly,

a) s is clearly `skip` or s''; `skip` for some $s'' \in$ STMT.

Therefore, by a), 2), and 3) we get that ψ' is a valid state.

*Case 68 (*SSASSIGN*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, x := e;\ s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim} \langle H, \langle \rho_n[x \mapsto v], \pi_n, s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$, $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{acc}(e)$, and $H, \rho_n \vdash e \Downarrow v$.

Since ψ is a valid state, by definition we get that

1) $x := e$; $s = s'$; `skip` for some $s' \in$ STMT,

2) $s_i = s_i'$; `skip` for some $s_i' \in$ STMT for all $1 \le i < n$, and

3) $s_i = s_i^1$; s_i^2 for some s_i^1, $s_i^2 \in$ STMT where s_i^1 is a method call or while loop statement for all $1 \le i < n$.

Then clearly,

a) s is `skip` or s''; `skip` for some $s'' \in$ STMT.

Therefore, by a), 2), and 3) we get that ψ' is a valid state.

*Case 69 (*SSALLOC*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{x := new C;}\ s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim} \langle H', \langle \rho_n[x \mapsto o], \pi_n \cup \overline{\langle o, f_i\rangle}, s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$ where $o \notin \mathsf{dom}(H)$, $\mathsf{fields}(C) = \overline{T_i\ \ f_i\ ;}$, and $H' = H[o \mapsto [\overline{f_i \mapsto \mathsf{defaultValue}(T_i)}]]$.

Since ψ is a valid state, by definition we get that

1) `x := new C;` $s = s'$; `skip` for some $s' \in$ STMT,

2) $s_i = s_i'$; `skip` for some $s_i' \in$ STMT for all $1 \le i < n$, and

3) $s_i = s_i^1$; s_i^2 for some s_i^1, $s_i^2 \in$ STMT where s_i^1 is a method call or while loop statement for all $1 \le i < n$.

Then clearly,

a) s is `skip` or s''; `skip` for some $s'' \in$ STMT.

Therefore, by a), 2), and 3) we get that ψ' is a valid state.

*Case 70 (*SSIFTRUE*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\}};\ s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim} \langle H, \langle \rho_n, \pi_n, r_1;\ s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$, $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{acc}(e)$, and $H, \rho_n \vdash e \Downarrow \texttt{true}$.

Since ψ is a valid state, by definition we get that

1) $\texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\}}$; $s = s'$; `skip` for some $s' \in$ STMT,

2) $s_i = s_i'$; `skip` for some $s_i' \in$ STMT for all $1 \le i < n$, and

3) $s_i = s_i^1$; s_i^2 for some s_i^1, $s_i^2 \in$ STMT where s_i^1 is a method call or while loop statement for all $1 \le i < n$.

Then clearly,

a) s is `skip` or s''; `skip` for some $s'' \in$ STMT, and so r_1; $s = r_1$; `skip` or r_1; $s = (r_1;\ s'')$; `skip` for some $s'' \in$ STMT.

Therefore, by a), 2), and 3) we get that ψ' is a valid state.

*Case 71 (*SSIFFALSE*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\}};\ s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim} \langle H, \langle \rho_n, \pi_n, r_2;\ s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$, $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{acc}(e)$, and $H, \rho_n \vdash e \Downarrow \texttt{false}$.

Since ψ is a valid state, by definition we get that

1) $\texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\}}\,;\ s = s'\,;\ \texttt{skip}$ for some $s' \in \textsc{Stmt}$,

2) $s_i = s'_i\,;\ \texttt{skip}$ for some $s'_i \in \textsc{Stmt}$ for all $1 \le i < n$, and

3) $s_i = s^1_i\,;\ s^2_i$ for some $s^1_i,\ s^2_i \in \textsc{Stmt}$ where s^1_i is a method call or while loop statement for all $1 \le i < n$.

Then clearly,

a) s is $\texttt{skip}$ or $s''\,;\ \texttt{skip}$ for some $s'' \in \textsc{Stmt}$, and so $r_2\,;\ s = r_2\,;\ \texttt{skip}$ or $r_2\,;\ s = (r_2\,;\ s'')\,;\ \texttt{skip}$ for some $s'' \in \textsc{Stmt}$.

Therefore, by a), 2), and 3) we get that ψ' is a valid state.

*Case 72 (*SsCall*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, y := z.m(\overline{x})\,;\ s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim} \langle H, \langle \rho'_n, \pi'_n, \mathsf{mbody}(m)\,;\ \texttt{skip}\rangle \cdot \langle \rho_n, \pi_n \setminus \pi'_n, y := z.m(\overline{x})\,;\ s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$, $H, \rho_n \vdash z \Downarrow o$, $H, \rho_n \vdash \overline{x} \Downarrow \overline{v}$,
$\rho'_n = [\texttt{this} \mapsto o, \overline{\mathsf{mparam}(m) \mapsto v}, \overline{\texttt{old}(\mathsf{mparam}(m)) \mapsto v}]$, $\pi'_n = \lfloor \mathsf{mpre}(m) \rfloor_{\pi_n, H, \rho'_n}$, $\pi'_n \subseteq \pi_n$, and $\langle H, \rho'_n, \pi'_n\rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpre}(m), \mathsf{body}_\mu\rangle$.

Since ψ is a valid state, by definition we get that

1) $y := z.m(\overline{x})\,;\ s = s'\,;\ \texttt{skip}$ for some $s' \in \textsc{Stmt}$,

2) $s_i = s'_i\,;\ \texttt{skip}$ for some $s'_i \in \textsc{Stmt}$ for all $1 \le i < n$, and

3) $s_i = s^1_i\,;\ s^2_i$ for some $s^1_i,\ s^2_i \in \textsc{Stmt}$ where s^1_i is a method call or while loop statement for all $1 \le i < n$.

Then clearly,

a) $\mathsf{mbody}(m)\,;\ \texttt{skip} = r\,;\ \texttt{skip}$ for some $r \in \textsc{Stmt}$, and

b) $y := z.m(\overline{x})\,;\ s = r_1\,;\ r_2$ for $r_1, r_2 \in \textsc{Stmt}$ where $r_1 = y := z.m(\overline{x})$ and $r_2 = s$.

Therefore, by a), 1), 2), b), and 3) we get that ψ' is a valid state.

*Case 73 (*SsCallFinish*).* We have
$\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip}\rangle \cdot \langle \rho_{n-1}, \pi_{n-1}, y := z.m(\overline{x})\,;\ s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim}$
$\langle H, \langle \rho_{n-1}[y \mapsto \rho_n(\texttt{result})], \pi_n \cup \pi_{n-1}, s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$, and
$\langle H, \rho_n, \pi_n\rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpost}(m), \mathsf{body}_\mu\rangle$.

Since ψ is a valid state, by definition we get that

1) $s_n = \texttt{skip}$,

2) $y := z.m(\overline{x})\,;\ s = s'\,;\ \texttt{skip}$ for some $s' \in \textsc{Stmt}$ and $s_i = s'_i\,;\ \texttt{skip}$ for some $s'_i \in \textsc{Stmt}$ for all $1 \le i < n-1$, and

3) $y := z.m(\overline{x})\,;\ s = s^1_{n-1}\,;\ s^2_{n-1}$ for some $s^1_{n-1}, s^2_{n-2} \in \textsc{Stmt}$ where $s^1_{n-1} = y := z.m(\overline{x})$, $s^2_{n-1} = s$ and $s_i = s^1_i\,;\ s^2_i$ for some $s^1_i,\ s^2_i \in \textsc{Stmt}$ where s^1_i is a method call or while loop statement for all $1 \le i < n-1$.

Then clearly,

a) s is $\texttt{skip}$ or $s''\,;\ \texttt{skip}$ for some $s'' \in \textsc{Stmt}$.

Therefore, by a), 2), and 3) we get that ψ' is a valid state.

*Case 74 (*SsWhileFalse*).* We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{while (e) inv}\ \widetilde{\phi}_{\texttt{i}}\ \texttt{\{ r \}}\,;\ s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim} \langle H, \langle \rho_n, \pi_n, s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$, $\langle H, \rho_n, \pi_n\rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_i, \mathsf{body}_\mu\rangle$, $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{acc}(e)$, and $H, \rho_n \vdash e \Downarrow \texttt{false}$.

Since ψ is a valid state, by definition we get that

1) $\texttt{while (e) inv}\ \widetilde{\phi}_{\texttt{i}}\ \texttt{\{ r \}}\,;\ s = s'\,;\ \texttt{skip}$ for some $s' \in \textsc{Stmt}$,

2) $s_i = s'_i$; skip for some $s'_i \in$ STMT for all $1 \le i < n$, and

3) $s_i = s^1_i$; s^2_i for some $s^1_i,\ s^2_i \in$ STMT where s^1_i is a method call or while loop statement for all $1 \le i < n$.

Then clearly,

a) s is `skip` or s''; `skip` for some $s'' \in$ STMT.

Therefore, by a), 2), and 3) we get that ψ' is a valid state.

Case 75 (SSWHILETRUE). We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}} \texttt{ \{ r \}; } s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{} \langle H, \langle \rho_n, \pi'_n, r\texttt{; skip}\rangle \cdot \langle \rho_n, \pi_n \setminus \pi'_n, \texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}} \texttt{ \{ r \}; } s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$,$\langle H, \rho_n, \pi_n\rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_i, \mathsf{body}_\mu\rangle$, $\langle H, \rho_n, \pi_n\rangle \vDash_E \mathsf{acc}(e)$, $H, \rho_n \vdash e \Downarrow \texttt{true}$, and $\pi'_n = \lfloor \widetilde{\phi}_i \rfloor_{\pi_n, H, \rho_n}$.

Since ψ is a valid state, by definition we get that

1) `while (e) inv` $\widetilde{\phi}_{\texttt{i}}$ `{ r }`; $s = s'$; `skip` for some $s' \in$ STMT,

2) $s_i = s'_i$; `skip` for some $s'_i \in$ STMT for all $1 \le i < n$, and

3) $s_i = s^1_i$; s^2_i for some $s^1_i,\ s^2_i \in$ STMT where s^1_i is a method call or while loop statement for all $1 \le i < n$.

Then clearly,

a) `while (e) inv` $\widetilde{\phi}_{\texttt{i}}$ `{ r }`; $s = r_1$; r_2 for $r_1, r_2 \in$ STMT where $r_1 =$ `while (e) inv` $\widetilde{\phi}_{\texttt{i}}$ `{ r }` and $r_2 = s$.

Therefore, by 1), 2), a), and 3) we get that ψ' is a valid state.

Case 76 (SSWHILEFINISH). We have
$\psi = \langle H, \langle \rho_n, \pi_n, \texttt{skip}\rangle \cdot \langle \rho_{n-1}, \pi_{n-1}, \texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}} \texttt{ \{ r \}; } s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{}$
$\langle H, \langle \rho_n, \pi_n \cup \pi_{n-1}, \texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}} \texttt{ \{ r \}; } s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$.

Since ψ is a valid state, by definition we get that

1) $s_n =$ `skip`,

2) `while (e) inv` $\widetilde{\phi}_{\texttt{i}}$ `{ r }`; $s = s'$; `skip` for some $s' \in$ STMT and $s_i = s'_i$; `skip` for some $s'_i \in$ STMT for all $1 \le i < n-1$, and

3) `while (e) inv` $\widetilde{\phi}_{\texttt{i}}$ `{ r }`; $s = s^1_{n-1}$; s^2_{n-1} for some $s^1_{n-1}, s^2_{n-2} \in$ STMT where $s^1_{n-1} =$ `while (e) inv` $\widetilde{\phi}_{\texttt{i}}$ `{ r }`, $s^2_{n-1} = s$ and $s_i = s^1_i$; s^2_i for some $s^1_i,\ s^2_i \in$ STMT where s^1_i is a method call or while loop statement for all $1 \le i < n-1$.

Therefore, by 2) and 3) we get that ψ' is a valid state.

Case 77 (SSFOLD). We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{fold } p(\overline{e})\texttt{; } s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{} \langle H, \langle \rho_n, \pi_n, s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$.

Since ψ is a valid state, by definition we get that

1) `fold` $p(\overline{e})$; $s = s'$; `skip` for some $s' \in$ STMT,

2) $s_i = s'_i$; `skip` for some $s'_i \in$ STMT for all $1 \le i < n$, and

3) $s_i = s^1_i$; s^2_i for some $s^1_i,\ s^2_i \in$ STMT where s^1_i is a method call or while loop statement for all $1 \le i < n$.

Then clearly,

a) s is `skip` or s''; `skip` for some $s'' \in$ STMT.

Therefore, by a), 2), and 3) we get that ψ' is a valid state.

Case 78 (SSUNFOLD). We have $\psi = \langle H, \langle \rho_n, \pi_n, \texttt{unfold}\ p(\overline{e});\ s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim} \langle H, \langle \rho_n, \pi_n, s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$.

Since ψ is a valid state, by definition we get that

1) $\texttt{unfold}\ p(\overline{e});\ s = s';\ \texttt{skip}$ for some $s' \in$ STMT,

2) $s_i = s_i';\ \texttt{skip}$ for some $s_i' \in$ STMT for all $1 \leq i < n$, and

3) $s_i = s_i^1;\ s_i^2$ for some $s_i^1,\ s_i^2 \in$ STMT where s_i^1 is a method call or while loop statement for all $1 \leq i < n$.

Then clearly,

a) s is $\texttt{skip}$ or $s'';\ \texttt{skip}$ for some $s'' \in$ STMT.

Therefore, by a), 2), and 3) we get that ψ' is a valid state.

6 GVL_{RP} Static Gradual Guarantee (Non-optimized Semantics)

The proof of the static gradual guarantee can be found here: extra-proofs/gvlrp-sgg.pdf.

7 GVL_{RP} Dynamic Gradual Guarantee (Non-optimized Semantics)

Proposition 1 (Dynamic gradual guarantee of verification).
Let $p_1, p_2 \in$ PROGRAM such that $p_1 \sqsubseteq_d p_2$, and $\psi_1, \psi_2 \in$ STATE such that $\psi_1 \lesssim \psi_2$.
If $\psi_1 \xrightarrow{\sim}_{p_1} \psi_1'$ then $\psi_2 \xrightarrow{\sim}_{p_2} \psi_2'$, with $\psi_1' \lesssim \psi_2'$.

Proof. Suppose $p_1, p_2 \in$ PROGRAM and $\psi_1, \psi_2 \in$ STATE such that $p_1 \sqsubseteq_d p_2$, $\psi_1 \lesssim \psi_2$, and $\psi_1 \xrightarrow{\sim}_{p_1} \psi_1'$.

Then, let $\psi_1 = \langle H, \langle \rho_n, \pi_n, s_n\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle$. Since $\psi_1 \lesssim \psi_2$, we get $\psi_2 = \langle H, \langle \rho_n, \pi_n', s_n'\rangle \cdot ... \cdot \langle \rho_1, \pi_1', s_1'\rangle \cdot \mathsf{nil}\rangle$ where $s_m \sqsubseteq s_m'$ for $1 \leq m \leq n$ and $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi_i'$ for $1 \leq m \leq n$.

We will show $\psi_2 \xrightarrow{\sim}_{p_2} \psi_2'$, with $\psi_1' \lesssim \psi_2'$ for some $\psi_2' \in$ STATE by case analysis on $\psi_1 \xrightarrow{\sim}_{p_1} \psi_1'$.

Case 79 (SSSKIP). We have $\psi_1 = \langle H, \langle \rho_n, \pi_n, \texttt{skip};\ s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim}_{p_1} \langle H, \langle \rho_n, \pi_n, s\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'$.

Since $\texttt{skip};\ s \sqsubseteq s_n'$, then by definition we get $s_n' = s_{n_1}';\ s_{n_2}'$ where $\texttt{skip} \sqsubseteq s_{n_1}'$ and $s \sqsubseteq s_{n_2}'$. Then $\texttt{skip} \sqsubseteq s_{n_1}'$ gives $s_{n_1}' = \texttt{skip}$, and so $s_n' = \texttt{skip};\ s_{n_2}'$.

Therefore, by SSSKIP, we have $\psi_2 = \langle H, \langle \rho_n, \pi_n', \texttt{skip};\ s_{n_2}'\rangle \cdot ... \cdot \langle \rho_1, \pi_1', s_1'\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim}_{p_2} \langle H, \langle \rho_n, \pi_n', s_{n_2}'\rangle \cdot ... \cdot \langle \rho_1, \pi_1', s_1'\rangle \cdot \mathsf{nil}\rangle = \psi_2'$.

Finally, since

1) $s \sqsubseteq s_{n_2}'$,

2) $s_m \sqsubseteq s_m'$ for $1 \leq m < n$, and

3) $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi_i'$ for $1 \leq m \leq n$,

we clearly have $\psi_1' \lesssim \psi_2'$.

Then clearly, $\psi_2 \xrightarrow{\sim}_{p_2} \psi_2'$, with $\psi_1' \lesssim \psi_2'$.

Case 80 (SsDeclare). We have $\psi_1 = \langle H, \langle \rho_n, \pi_n, \texttt{T x; } s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \widetilde{\longrightarrow}_{p_1} \langle H, \langle \rho_n, \pi_n, s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi_1'$.

Since $\texttt{T x; } s \sqsubseteq s_n'$, then by definition we get $s_n' = s_{n_1}' \texttt{ ; } s_{n_2}'$ where $\texttt{T x} \sqsubseteq s_{n_1}'$ and $s \sqsubseteq s_{n_2}'$. Then $\texttt{T x} \sqsubseteq s_{n_1}'$ gives $s_{n_1}' = \texttt{T x}$, and so $s_n' = \texttt{T x; } s_{n_2}'$.

Therefore, by SsDeclare, we have $\psi_2 = \langle H, \langle \rho_n, \pi_n', \texttt{T x; } s_{n_2}'\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1', s_1'\rangle \cdot \mathsf{nil}\rangle \widetilde{\longrightarrow}_{p_2} \langle H, \langle \rho_n, \pi_n', s_{n_2}'\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1', s_1'\rangle \cdot \mathsf{nil}\rangle = \psi_2'$.

Finally, since
1) $s \sqsubseteq s_{n_2}'$,
2) $s_m \sqsubseteq s_m'$ for $1 \le m < n$, and
3) $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi_i'$ for $1 \le m \le n$,
we clearly have $\psi_1' \lesssim \psi_2'$.

Then clearly, $\psi_2 \widetilde{\longrightarrow}_{p_2} \psi_2'$, with $\psi_1' \lesssim \psi_2'$.

Case 81 (SsAssert). We have $\psi_1 = \langle H, \langle \rho_n, \pi_n, \texttt{assert } \phi' \texttt{; } s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \widetilde{\longrightarrow}_{p_1} \langle H, \langle \rho_n, \pi_n, s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi_1'$ and $\langle H, \rho_n, \pi_n\rangle \mathrel{\widetilde{\models}} \langle ?*\phi', \mathsf{body}_{\mu_{p_1}}\rangle$.

Since $\texttt{assert } \phi' \texttt{; } s \sqsubseteq s_n'$, then by definition we get $s_n' = s_{n_1}' \texttt{ ; } s_{n_2}'$ where $\texttt{assert } \phi' \sqsubseteq s_{n_1}'$ and $s \sqsubseteq s_{n_2}'$. Then $\texttt{assert } \phi' \sqsubseteq s_{n_1}'$ gives $s_{n_1}' = \texttt{assert } \phi'$, and so $s_n' = \texttt{assert } \phi' \texttt{; } s_{n_2}'$.

Since $p_1 \sqsubseteq_d p_2$, we get that $\mathsf{dom}(\mathsf{body}_{\mu_{p_1}}) = \mathsf{dom}(\mathsf{body}_{\mu_{p_2}})$ and $\forall p \in \mathsf{dom}(\mathsf{body}_{\mu_{p_1}})\,.\,\forall \overline{tmp} \in \textsc{Var}^*\,.\,\mathsf{body}_{\mu_{p_1}}(p)(\overline{tmp}) \sqsubseteq \mathsf{body}_{\mu_{p_2}}(p)(\overline{tmp})$.

Then, by lemma 13 on
$\mathsf{dom}(\mathsf{body}_{\mu_{p_1}}) = \mathsf{dom}(\mathsf{body}_{\mu_{p_2}}), \forall p \in \mathsf{dom}(\mathsf{body}_{\mu_{p_1}})\,.\,\forall \overline{tmp} \in \textsc{Var}^*\,.\,\mathsf{body}_{\mu_{p_1}}(p)(\overline{tmp}) \sqsubseteq \mathsf{body}_{\mu_{p_2}}(p)(\overline{tmp})$ and
$\langle H, \rho_n, \pi_n\rangle \mathrel{\widetilde{\models}} \langle ? * \phi', \mathsf{body}_{\mu_{p_1}}\rangle$, we get $\langle H, \rho_n, \pi_n\rangle \mathrel{\widetilde{\models}} \langle ? * \phi', \mathsf{body}_{\mu_{p_2}}\rangle$.

Also, since $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi_i'$ for $1 \le m \le n$, we get $\pi_n \subseteq \pi_n'$.

By lemma 14 on $\pi_n \subseteq \pi_n'$ and $\langle H, \rho_n, \pi_n\rangle \mathrel{\widetilde{\models}} \langle ?*\phi', \mathsf{body}_{\mu_{p_2}}\rangle$, we get $\langle H, \rho_n, \pi_n'\rangle \mathrel{\widetilde{\models}} \langle ?*\phi', \mathsf{body}_{\mu_{p_2}}\rangle$.

Therefore, since $\langle H, \rho_n, \pi_n'\rangle \mathrel{\widetilde{\models}} \langle ? * \phi', \mathsf{body}_{\mu_{p_2}}\rangle$, by SsAssert, we have
$\psi_2 = \langle H, \langle \rho_n, \pi_n', \texttt{assert } \phi' \texttt{; } s_{n_2}'\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1', s_1'\rangle \cdot \mathsf{nil}\rangle \widetilde{\longrightarrow}_{p_2}$
$\langle H, \langle \rho_n, \pi_n', s_{n_2}'\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1', s_1'\rangle \cdot \mathsf{nil}\rangle = \psi_2'$.

Finally, since
1) $s \sqsubseteq s_{n_2}'$,
2) $s_m \sqsubseteq s_m'$ for $1 \le m < n$, and
3) $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi_i'$ for $1 \le m \le n$,
we clearly have $\psi_1' \lesssim \psi_2'$.

Then clearly, $\psi_2 \widetilde{\longrightarrow}_{p_2} \psi_2'$, with $\psi_1' \lesssim \psi_2'$.

Case 82 (SsFAssign). We have $\psi_1 = \langle H, \langle \rho_n, \pi_n, \texttt{x.f := y; } s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \widetilde{\longrightarrow}_{p_1} \langle H[o \mapsto [f \mapsto v]], \langle \rho_n, \pi_n, s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi_1', \langle H, \rho_n, \pi_n\rangle \models_E \mathbf{acc}(\texttt{x.f})$, and $H, \rho_n \vdash y \Downarrow v$.

Since x.f := y; $s \sqsubseteq s'_n$, then by definition we get $s'_n = s'_{n_1}$; s'_{n_2} where x.f := y $\sqsubseteq s'_{n_1}$ and $s \sqsubseteq s'_{n_2}$. Then x.f := y $\sqsubseteq s'_{n_1}$ gives s'_{n_1} = x.f := y, and so s'_n = x.f := y; s'_{n_2}.

Since $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$, we get $\pi_n \subseteq \pi'_n$.

By lemma 1 on $\pi_n \subseteq \pi'_n$ and $\langle H, \rho_n, \pi_n \rangle \vDash_E$ **acc**(x.f), we get $\langle H, \rho_n, \pi'_n \rangle \vDash_E$ **acc**(x.f).

Therefore, since $\langle H, \rho_n, \pi'_n \rangle \vDash_E$ **acc**(x.f) and $H, \rho_n \vdash y \Downarrow v$, by SsFAssign, we have

$\psi_2 = \langle H, \langle \rho_n, \pi'_n, \texttt{x.f := y; } s'_{n_2} \rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_2}$
$\langle H[o \mapsto [f \mapsto v]], \langle \rho_n, \pi'_n, s'_{n_2} \rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_2$.

Finally, since

1) $s \sqsubseteq s'_{n_2}$,
2) $s_m \sqsubseteq s'_m$ for $1 \leq m < n$, and
3) $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$,

we clearly have $\psi'_1 \lesssim \psi'_2$.

Then clearly, $\psi_2 \xrightarrow{\sim}_{p_2} \psi'_2$, with $\psi'_1 \lesssim \psi'_2$.

*Case 83 (*SsAssign*).* We have $\psi_1 = \langle H, \langle \rho_n, \pi_n, x := e; s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_1} \langle H, \langle \rho_n[x \mapsto v], \pi_n, s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_1$, $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$, and $H, \rho_n \vdash e \Downarrow v$.

Since $x := e;\ s \sqsubseteq s'_n$, then by definition we get $s'_n = s'_{n_1}$; s'_{n_2} where $x := e \sqsubseteq s'_{n_1}$ and $s \sqsubseteq s'_{n_2}$. Then $x := e \sqsubseteq s'_{n_1}$ gives $s'_{n_1} = x := e$, and so $s'_n = x := e;\ s'_{n_2}$.

Since $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$, we get $\pi_n \subseteq \pi'_n$.

By lemma 1 on $\pi_n \subseteq \pi'_n$ and $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$, we get $\langle H, \rho_n, \pi'_n \rangle \vDash_E \mathsf{acc}(e)$.

Therefore, since $\langle H, \rho_n, \pi'_n \rangle \vDash_E \mathsf{acc}(e)$ and $H, \rho_n \vdash e \Downarrow v$, by SsAssign, we have

$\psi_2 = \langle H, \langle \rho_n, \pi'_n, x := e;\ s'_{n_2} \rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_2}$
$\langle H, \langle \rho_n[x \mapsto v], \pi'_n, s'_{n_2} \rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_2$.

Finally, since

1) $s \sqsubseteq s'_{n_2}$,
2) $s_m \sqsubseteq s'_m$ for $1 \leq m < n$, and
3) $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$,

we clearly have $\psi'_1 \lesssim \psi'_2$.

Then clearly, $\psi_2 \xrightarrow{\sim}_{p_2} \psi'_2$, with $\psi'_1 \lesssim \psi'_2$.

*Case 84 (*SsAlloc*).* We have $\psi_1 = \langle H, \langle \rho_n, \pi_n, \texttt{x := new C; } s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_1} \langle H', \langle \rho_n[x \mapsto o], \pi_n \cup \overline{\langle o, f_i \rangle}, s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_1$ where $o \notin \mathsf{dom}(H)$, $\mathsf{fields}(C) = \overline{T_i\ \ f_i\ ;}$, and $H' = H[o \mapsto \overline{[f_i \mapsto \mathsf{defaultValue}(T_i)]}]$.

Since x := new C; $s \sqsubseteq s'_n$, then by definition we get $s'_n = s'_{n_1}$; s'_{n_2} where x := new C $\sqsubseteq s'_{n_1}$ and $s \sqsubseteq s'_{n_2}$. Then x := new C $\sqsubseteq s'_{n_1}$ gives s'_{n_1} = x := new C, and so s'_n = x := new C; s'_{n_2}.

Therefore, since $o \notin \mathsf{dom}(H)$, $\mathsf{fields}(C) = \overline{T_i \ f_i \ ;}$, and $H' = H[o \mapsto [\overline{f_i \mapsto \mathsf{defaultValue}(T_i)}]]$, by SsAlloc, we have

$\psi_2 = \langle H, \langle \rho_n, \pi'_n, \texttt{x := new C;}\ s'_{n_2} \rangle \cdot ... \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_2}$
$\langle H', \langle \rho_n[x \mapsto o], \pi'_n \cup \overline{\langle o, f_i \rangle}, s'_{n_2} \rangle \cdot ... \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_2$.

Also, since $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$, we get $\pi_n \cup \bigcup_{i=m}^{n-1} \pi_i \subseteq \pi'_n \cup \bigcup_{i=m}^{n-1} \pi'_i$ for $1 \leq m \leq n$.

Finally, since
1) $s \sqsubseteq s'_{n_2}$,
2) $s_m \sqsubseteq s'_m$ for $1 \leq m < n$, and
3) $\pi_n \cup \overline{\langle o, f_i \rangle} \cup \bigcup_{i=m}^{n-1} \pi_i \subseteq \pi'_n \cup \overline{\langle o, f_i \rangle} \cup \bigcup_{i=m}^{n-1} \pi'_i$ for $1 \leq m \leq n$

we clearly have $\psi'_1 \lesssim \psi'_2$.

Then clearly, $\psi_2 \xrightarrow{\sim}_{p_2} \psi'_2$, with $\psi'_1 \lesssim \psi'_2$.

Case 85 (SsIfTrue). We have $\psi_1 = \langle H, \langle \rho_n, \pi_n, \texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\};}\ s \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_1} \langle H, \langle \rho_n, \pi_n, r_1\ ;\ s \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_1$, $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$, and $H, \rho_n \vdash e \Downarrow \texttt{true}$.

Since $\texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\};}\ s \sqsubseteq s'_n$, then by definition we get $s'_n = s'_{n_1}\ ;\ s'_{n_2}$ where $\texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\}} \sqsubseteq s'_{n_1}$ and $s \sqsubseteq s'_{n_2}$. Then $\texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\}} \sqsubseteq s'_{n_1}$ gives $s'_{n_1} = \texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\}}$, and so $s'_n = \texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\};}\ s'_{n_2}$.

Since $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$, we get $\pi_n \subseteq \pi'_n$.

By lemma 1 on $\pi_n \subseteq \pi'_n$ and $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$, we get $\langle H, \rho_n, \pi'_n \rangle \vDash_E \mathsf{acc}(e)$.

Therefore, since $\langle H, \rho_n, \pi'_n \rangle \vDash_E \mathsf{acc}(e)$ and $H, \rho_n \vdash e \Downarrow \texttt{true}$, by SsIfTrue, we have

$\psi_2 = \langle H, \langle \rho_n, \pi'_n, \texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\};}\ s'_{n_2} \rangle \cdot ... \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_2}$
$\langle H, \langle \rho_n, \pi'_n, r_1\ ;\ s'_{n_2} \rangle \cdot ... \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_2$.

Finally, since
1) $r_1 \sqsubseteq r_1$ and $s \sqsubseteq s'_{n_2}$ gives $r_1\ ;\ s \sqsubseteq r_1\ ;\ s'_{n_2}$,
2) $s_m \sqsubseteq s'_m$ for $1 \leq m < n$, and
3) $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$,

we clearly have $\psi'_1 \lesssim \psi'_2$.

Then clearly, $\psi_2 \xrightarrow{\sim}_{p_2} \psi'_2$, with $\psi'_1 \lesssim \psi'_2$.

Case 86 (SsIfFalse). We have $\psi_1 = \langle H, \langle \rho_n, \pi_n, \texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\};}\ s \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_1} \langle H, \langle \rho_n, \pi_n, r_2\ ;\ s \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_1$, $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$, and $H, \rho_n \vdash e \Downarrow \texttt{false}$.

Since $\texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\};}\ s \sqsubseteq s'_n$, then by definition we get $s'_n = s'_{n_1}\ ;\ s'_{n_2}$ where $\texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\}} \sqsubseteq s'_{n_1}$ and $s \sqsubseteq s'_{n_2}$. Then $\texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\}} \sqsubseteq s'_{n_1}$ gives $s'_{n_1} = \texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\}}$, and so $s'_n = \texttt{if (e) \{r}_1\texttt{\} else \{r}_2\texttt{\};}\ s'_{n_2}$.

Since $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$, we get $\pi_n \subseteq \pi'_n$.

By lemma 1 on $\pi_n \subseteq \pi'_n$ and $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$, we get $\langle H, \rho_n, \pi'_n \rangle \vDash_E \mathsf{acc}(e)$.

Therefore, since $\langle H, \rho_n, \pi'_n \rangle \vDash_E \mathsf{acc}(e)$ and $H, \rho_n \vdash e \Downarrow \mathtt{false}$, by SsIfFalse, we have

$\psi_2 = \langle H, \langle \rho_n, \pi'_n, \mathtt{if}\ (\mathtt{e})\ \{\mathtt{r}_1\}\ \mathtt{else}\ \{\mathtt{r}_2\}\ ;\ s'_{n_2} \rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle \widetilde{\longrightarrow}_{p_2}$
$\langle H, \langle \rho_n, \pi'_n, r_2\ ;\ s'_{n_2} \rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_2$.

Finally, since

1) $r_2 \sqsubseteq r_2$ and $s \sqsubseteq s'_{n_2}$ gives $r_2\ ;\ s \sqsubseteq r_2\ ;\ s'_{n_2}$,

2) $s_m \sqsubseteq s'_m$ for $1 \leq m < n$, and

3) $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$,

we clearly have $\psi'_1 \lesssim \psi'_2$.

Then clearly, $\psi_2 \widetilde{\longrightarrow}_{p_2} \psi'_2$, with $\psi'_1 \lesssim \psi'_2$.

*Case 87 (*SsCall*).* We have $\psi_1 = \langle H, \langle \rho_n, \pi_n, y := z.m(\overline{x})\ ;\ s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \widetilde{\longrightarrow}_{p_1} \langle H, \langle \rho_n^{p_1}, \pi_n^{p_1}, \mathsf{mbody}(m)_{p_1}\ ;\ \mathtt{skip} \rangle \cdot \langle \rho_n, \pi_n \setminus \pi_n^{p_1}, y := z.m(\overline{x})\ ;\ s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_1$, $\overline{H, \rho_n \vdash z \Downarrow o}$, $\overline{H, \rho_n \vdash x \Downarrow v}$,
$\rho_n^{p_1} = [\mathtt{this} \mapsto o, \overline{\mathsf{mparam}(m)_{p_1} \mapsto v}, \overline{\mathtt{old}(\mathsf{mparam}(m)_{p_1}) \mapsto v}]$, $\pi_n^{p_1} = \lfloor \mathsf{mpre}(m)_{p_1} \rfloor^{p_1}_{\pi_n, H, \rho_n^{p_1}}$,
$\pi_n^{p_1} \subseteq \pi_n$, and $\langle H, \rho_n^{p_1}, \pi_n^{p_1} \rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpre}(m)_{p_1}, \mathsf{body}_{\mu_{p_1}} \rangle$.

Since $y := z.m(\overline{x})\ ;\ s \sqsubseteq s'_n$, then by definition we get $s'_n = s'_{n_1}\ ;\ s'_{n_2}$ where $y := z.m(\overline{x}) \sqsubseteq s'_{n_1}$ and $s \sqsubseteq s'_{n_2}$. Then $y := z.m(\overline{x}) \sqsubseteq s'_{n_1}$ gives $s'_{n_1} = y := z.m(\overline{x})$, and so $s'_n = y := z.m(\overline{x})\ ;\ s'_{n_2}$.

Since $p_1 \sqsubseteq_d p_2$, we get that $\mathsf{mbody}(m)_{p_1} \sqsubseteq \mathsf{mbody}(m)_{p_2}$, $\mathsf{mpre}(m)_{p_1} \sqsubseteq \mathsf{mpre}(m)_{p_2}$, $\mathsf{dom}(\mathsf{body}_{\mu_{p_1}}) = \mathsf{dom}(\mathsf{body}_{\mu_{p_2}})$, and $\forall p \in \mathsf{dom}(\mathsf{body}_{\mu_{p_1}})\,.\,\forall \overline{tmp} \in \textsc{Var}^*\,.\,\mathsf{body}_{\mu_{p_1}}(p)(\overline{tmp}) \sqsubseteq \mathsf{body}_{\mu_{p_2}}(p)(\overline{tmp})$, and $\rho_n^{p_2} = [\mathtt{this} \mapsto o, \overline{\mathsf{mparam}(m)_{p_2} \mapsto v}, \overline{\mathtt{old}(\mathsf{mparam}(m)_{p_2}) \mapsto v}] = \rho_n^{p_1}$.

$\langle H, \rho_n^{p_1}, \pi_n^{p_1} \rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpre}(m)_{p_1}, \mathsf{body}_{\mu_{p_1}} \rangle$ and $\rho_n^{p_1} = \rho_n^{p_2}$, gives $\langle H, \rho_n^{p_2}, \pi_n^{p_1} \rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpre}(m)_{p_1}, \mathsf{body}_{\mu_{p_1}} \rangle$.

Also, since $p_1 \sqsubseteq_d p_2$, $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$ gives $\pi_n \subseteq \pi'_n$, $\mathsf{mpre}(m)_{p_1} \sqsubseteq \mathsf{mpre}(m)_{p_2}$, $\pi_n^{p_1} = \lfloor \mathsf{mpre}(m)_{p_1} \rfloor^{p_1}_{\pi_n, H, \rho_n^{p_1}} = \lfloor \mathsf{mpre}(m)_{p_1} \rfloor^{p_1}_{\pi_n, H, \rho_n^{p_2}}$, and $\pi_n^{p_1} \subseteq \pi_n$ by lemma 15, we get $\pi_n^{p_2} = \lfloor \mathsf{mpre}(m)_{p_2} \rfloor^{p_2}_{\pi'_n, H, \rho_n^{p_2}}$ and $\pi_n^{p_1} \subseteq \pi_n^{p_2}$.

Then, $\pi_n^{p_2} \subseteq \pi'_n$ since if $\mathsf{mpre}(m)_{p_2}$ is semantically precise and $p_1 \sqsubseteq_d p_2$, we get $\pi_n^{p_2} = \pi_n^{p_1} \subseteq \pi_n \subseteq \pi'_n$ and otherwise, $\pi_n^{p_2} = \pi'_n$.

By lemma 14 on $\pi_n^{p_1} \subseteq \pi_n^{p_2}$ and $\langle H, \rho_n^{p_2}, \pi_n^{p_1} \rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpre}(m)_{p_1}, \mathsf{body}_{\mu_{p_1}} \rangle$, we get
$\langle H, \rho_n^{p_2}, \pi_n^{p_2} \rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpre}(m)_{p_1}, \mathsf{body}_{\mu_{p_1}} \rangle$.

By lemma 12 on $\mathsf{mpre}(m)_{p_1} \sqsubseteq \mathsf{mpre}(m)_{p_2}$ and $\langle H, \rho_n^{p_2}, \pi_n^{p_2} \rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpre}(m)_{p_1}, \mathsf{body}_{\mu_{p_1}} \rangle$, we get $\langle H, \rho_n^{p_2}, \pi_n^{p_2} \rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpre}(m)_{p_2}, \mathsf{body}_{\mu_{p_1}} \rangle$.

Similarly, by lemma 13 on
$\mathsf{dom}(\mathsf{body}_{\mu_{p_1}}) = \mathsf{dom}(\mathsf{body}_{\mu_{p_2}}), \forall p \in \mathsf{dom}(\mathsf{body}_{\mu_{p_1}})\,.\,\forall \overline{tmp} \in \text{VAR}^*\,.\,\mathsf{body}_{\mu_{p_1}}(p)(\overline{tmp}) \sqsubseteq \mathsf{body}_{\mu_{p_2}}(p)(\overline{tmp})$, and
$\langle H, \rho_n^{p_2}, \pi_n^{p_2}\rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpre}(m)_{p_2}, \mathsf{body}_{\mu_{p_1}}\rangle$, we get $\langle H, \rho_n^{p_2}, \pi_n^{p_2}\rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpre}(m)_{p_2}, \mathsf{body}_{\mu_{p_2}}\rangle$.

Therefore, since $\langle H, \rho_n^{p_2}, \pi_n^{p_2}\rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpre}(m)_{p_2}, \mathsf{body}_{\mu_{p_2}}\rangle$, $\pi_n^{p_2} = \lfloor \mathsf{mpre}(m)_{p_2} \rfloor^{p_2}_{\pi_n', H, \rho_n^{p_2}}$, and $\pi_n^{p_2} \subseteq \pi_n'$, by SSCALL, we have
$\psi_2 = \langle H, \langle \rho_n, \pi_n', y := z.m(\overline{x})\ ;\ s_{n_2}'\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1', s_1'\rangle \cdot \mathsf{nil}\rangle \mathrel{\widetilde{\longrightarrow}}_{p_2}$

$\langle H, \langle \rho_n^{p_2}, \pi_n^{p_2}, \mathsf{mbody}(m)_{p_2}\ ;\ \mathtt{skip}\rangle \cdot \langle \rho_n, \pi_n' \setminus \pi_n^{p_2}, y := z.m(\overline{x})\ ;\ s_{n_2}'\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1', s_1'\rangle \cdot \mathsf{nil}\rangle = \psi_2'$

We have $\pi_n^{p_1} \subseteq \pi_n^{p_2}$ and $\pi_n \subseteq \pi_n'$. Also, $\pi_n \subseteq \pi_n^{p_1} \cup \pi_n \setminus \pi_n^{p_1}$ and $\pi_n' \subseteq \pi_n^{p_2} \cup \pi_n' \setminus \pi_n^{p_2}$.
Therefore, $\pi_n^{p_1} \cup \pi_n \setminus \pi_n^{p_1} \subseteq \pi_n^{p_2} \cup \pi_n' \setminus \pi_n^{p_2}$.

Also, since $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi_i'$ for $1 \le m \le n$, we get $\pi_n \cup \bigcup_{i=m}^{n-1} \pi_i \subseteq \pi_n' \cup \bigcup_{i=m}^{n-1} \pi_i'$ for $1 \le m \le n$.

Therefore, we have $(\pi_n^{p_1} \cup \pi_n \setminus \pi_n^{p_1}) \cup \bigcup_{i=m}^{n-1} \pi_i \subseteq (\pi_n^{p_2} \cup \pi_n' \setminus \pi_n^{p_2}) \cup \bigcup_{i=m}^{n-1} \pi_i'$ for $1 \le m \le n$.

Finally, since
1) $\rho_n^{p_1} = \rho_n^{p_2}$,
2) $s \sqsubseteq s_{n_2}'$ gives $y := z.m(\overline{x})\ ;\ s \sqsubseteq y := z.m(\overline{x})\ ;\ s_{n_2}'$ and $\mathsf{mbody}(m)_{p_1} \sqsubseteq \mathsf{mbody}(m)_{p_2}$ gives $\mathsf{mbody}(m)_{p_1}\ ;\ \mathtt{skip} \sqsubseteq \mathsf{mbody}(m)_{p_2}\ ;\ \mathtt{skip}$,
3) $s_m \sqsubseteq s_m'$ for $1 \le m < n-1$,
4) $(\pi_n^{p_1} \cup \pi_n \setminus \pi_n^{p_1}) \cup \bigcup_{i=m}^{n-1} \pi_i \subseteq (\pi_n^{p_2} \cup \pi_n' \setminus \pi_n^{p_2}) \cup \bigcup_{i=m}^{n-1} \pi_i'$ for $1 \le m \le n$, and $\pi_n^{p_1} \subseteq \pi_n^{p_2}$.
we clearly have $\psi_1' \lesssim \psi_2'$.

Then clearly, $\psi_2 \mathrel{\widetilde{\longrightarrow}}_{p_2} \psi_2'$, with $\psi_1' \lesssim \psi_2'$.

*Case 88 (*SSCALLFINISH*).* We have
$\psi_1 = \langle H, \langle \rho_n, \pi_n, \mathtt{skip}\rangle \cdot \langle \rho_{n-1}, \pi_{n-1}, y := z.m(\overline{x})\ ;\ s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \mathrel{\widetilde{\longrightarrow}}_{p_1}$
$\langle H, \langle \rho_{n-1}[y \mapsto \rho_n(\mathtt{result})], \pi_n \cup \pi_{n-1}, s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi_1'$, and
$\langle H, \rho_n, \pi_n\rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpost}(m)_{p_1}, \mathsf{body}_{\mu_{p_1}}\rangle$.

Since $\mathtt{skip} \sqsubseteq s_n'$, then by definition we get $s_n' = \mathtt{skip}$.

Also, since $y := z.m(\overline{x})\ ;\ s \sqsubseteq s_{n-1}'$, then by definition we get $s_{n-1}' = s_{n-1_1}'\ ;\ s_{n-1_2}'$ where $y := z.m(\overline{x}) \sqsubseteq s_{n-1_1}'$ and $s \sqsubseteq s_{n-1_2}'$. Then $y := z.m(\overline{x}) \sqsubseteq s_{n-1_1}'$ gives $s_{n-1_1}' = y := z.m(\overline{x})$, and so $s_{n-1}' = y := z.m(\overline{x})\ ;\ s_{n-1_2}'$.

Since $p_1 \sqsubseteq_d p_2$, we get that $\mathsf{mpost}(m)_{p_1} \sqsubseteq \mathsf{mpost}(m)_{p_2}$, $\mathsf{dom}(\mathsf{body}_{\mu_{p_1}}) = \mathsf{dom}(\mathsf{body}_{\mu_{p_2}})$, and $\forall p \in \mathsf{dom}(\mathsf{body}_{\mu_{p_1}})\,.\,\forall \overline{tmp} \in \text{VAR}^*\,.\,\mathsf{body}_{\mu_{p_1}}(p)(\overline{tmp}) \sqsubseteq \mathsf{body}_{\mu_{p_2}}(p)(\overline{tmp})$.

By lemma 12 on $\mathsf{mpost}(m)_{p_1} \sqsubseteq \mathsf{mpost}(m)_{p_2}$ and $\langle H, \rho_n, \pi_n\rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpost}(m)_{p_1}, \mathsf{body}_{\mu_{p_1}}\rangle$, we get $\langle H, \rho_n, \pi_n\rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpost}(m)_{p_2}, \mathsf{body}_{\mu_{p_1}}\rangle$.

Similarly, by lemma 13 on

$\mathsf{dom}(\mathsf{body}_{\mu_{p_1}}) = \mathsf{dom}(\mathsf{body}_{\mu_{p_2}}), \forall p \in \mathsf{dom}(\mathsf{body}_{\mu_{p_1}}) \,.\, \forall \overline{tmp} \in \text{VAR}^* \,.\, \mathsf{body}_{\mu_{p_1}}(p)(\overline{tmp}) \sqsubseteq \mathsf{body}_{\mu_{p_2}}(p)(\overline{tmp})$, and

$\langle H, \rho_n, \pi_n \rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpost}(m)_{p_2}, \mathsf{body}_{\mu_{p_1}} \rangle$, we get $\langle H, \rho_n, \pi_n \rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpost}(m)_{p_2}, \mathsf{body}_{\mu_{p_2}} \rangle$.

Finally, since $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$ gives $\pi_n \subseteq \pi'_n$, by lemma 14, we get $\langle H, \rho_n, \pi'_n \rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpost}(m)_{p_2}, \mathsf{body}_{\mu_{p_2}} \rangle$.

Therefore, since $\langle H, \rho_n, \pi'_n \rangle \mathrel{\widetilde{\vDash}} \langle \mathsf{mpost}(m)_{p_2}, \mathsf{body}_{\mu_{p_2}} \rangle$, by SsCallFinish, we have $\psi_2 = \langle H, \langle \rho_n, \pi'_n, \texttt{skip} \rangle \cdot \langle \rho_{n-1}, \pi'_{n-1}, y := z.m(\overline{x})\,;\, s'_{n-1_2} \rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_2} \langle H, \langle \rho_{n-1}[y \mapsto \rho_n(\texttt{result})], \pi'_n \cup \pi'_{n-1}, s'_{n-1_2} \rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_2$.

Also, since $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$, we get $(\pi_n \cup \pi_{n-1}) \cup \bigcup_{i=m}^{n-2} \pi_i \subseteq (\pi'_n \cup \pi'_{n-1}) \cup \bigcup_{i=m}^{n-2} \pi'_i$ for $1 \leq m \leq n-1$.

Finally, since

1) $s \sqsubseteq s'_{n-1_2}$,
2) $s_m \sqsubseteq s'_m$ for $1 \leq m < n-1$, and
3) $(\pi_n \cup \pi_{n-1}) \cup \bigcup_{i=m}^{n-2} \pi_i \subseteq (\pi'_n \cup \pi'_{n-1}) \cup \bigcup_{i=m}^{n-2} \pi'_i$ for $1 \leq m \leq n-1$

we clearly have $\psi'_1 \lesssim \psi'_2$.

Then clearly, $\psi_2 \xrightarrow{\sim}_{p_2} \psi'_2$, with $\psi'_1 \lesssim \psi'_2$.

Case 89 (SsWhileFalse). We have $\psi_1 = \langle H, \langle \rho_n, \pi_n, \texttt{while (e) inv } \widetilde{\phi}_{\mathtt{i}}^{\mathtt{p}_1} \texttt{ \{ r \}; } s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_1} \langle H, \langle \rho_n, \pi_n, s \rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_{p_1}$, $\langle H, \rho_n, \pi_n \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_i^{p_1}, \mathsf{body}_{\mu_{p_1}} \rangle$, $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$, and $H, \rho_n \vdash e \Downarrow \texttt{false}$.

Since $\texttt{while (e) inv } \widetilde{\phi}_{\mathtt{i}}^{\mathtt{p}_1} \texttt{ \{ r \}; } s \sqsubseteq s'_n$, then by definition we get $s'_n = s'_{n_1}\,;\, s'_{n_2}$ where $\texttt{while (e) inv } \widetilde{\phi}_{\mathtt{i}}^{\mathtt{p}_1} \texttt{ \{ r \}} \sqsubseteq s'_{n_1}$ and $s \sqsubseteq s'_{n_2}$. Then $\texttt{while (e) inv } \widetilde{\phi}_{\mathtt{i}}^{\mathtt{p}_1} \texttt{ \{ r \}} \sqsubseteq s'_{n_1}$ gives $s'_{n_1} = \texttt{while (e) inv } \widetilde{\phi}_{\mathtt{i}}^{\mathtt{p}_2} \texttt{ \{ r \}}$, where $\widetilde{\phi}_i^{p_1} \sqsubseteq \widetilde{\phi}_i^{p_2}$, and so $s'_n = \texttt{while (e) inv } \widetilde{\phi}_{\mathtt{i}}^{\mathtt{p}_2} \texttt{ \{ r \}; } s'_{n_2}$.

Since $p_1 \sqsubseteq_d p_2$, we get that $\mathsf{dom}(\mathsf{body}_{\mu_{p_1}}) = \mathsf{dom}(\mathsf{body}_{\mu_{p_2}})$ and $\forall p \in \mathsf{dom}(\mathsf{body}_{\mu_{p_1}}) \,.\, \forall \overline{tmp} \in \text{VAR}^* \,.\, \mathsf{body}_{\mu_{p_1}}(p)(\overline{tmp}) \sqsubseteq \mathsf{body}_{\mu_{p_2}}(p)(\overline{tmp})$.

By lemma 12 on $\widetilde{\phi}_i^{p_1} \sqsubseteq \widetilde{\phi}_i^{p_2}$ and $\langle H, \rho_n, \pi_n \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_i^{p_1}, \mathsf{body}_{\mu_{p_1}} \rangle$, we get $\langle H, \rho_n, \pi_n \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_i^{p_2}, \mathsf{body}_{\mu_{p_1}} \rangle$.

Similarly, by lemma 13 on
$\mathsf{dom}(\mathsf{body}_{\mu_{p_1}}) = \mathsf{dom}(\mathsf{body}_{\mu_{p_2}}), \forall p \in \mathsf{dom}(\mathsf{body}_{\mu_{p_1}}) \,.\, \forall \overline{tmp} \in \text{VAR}^* \,.\, \mathsf{body}_{\mu_{p_1}}(p)(\overline{tmp}) \sqsubseteq \mathsf{body}_{\mu_{p_2}}(p)(\overline{tmp})$, and

$\langle H, \rho_n, \pi_n \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_i^{p_2}, \mathsf{body}_{\mu_{p_1}} \rangle$, we get $\langle H, \rho_n, \pi_n \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_i^{p_2}, \mathsf{body}_{\mu_{p_2}} \rangle$.

Finally, since $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$ gives $\pi_n \subseteq \pi'_n$, by lemma 14, we get $\langle H, \rho_n, \pi'_n \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_i^{p_2}, \mathsf{body}_{\mu_{p_2}} \rangle$.

By lemma 1 on $\pi_n \subseteq \pi'_n$ and $\langle H, \rho_n, \pi_n \rangle \vDash_E \mathsf{acc}(e)$, we get $\langle H, \rho_n, \pi'_n \rangle \vDash_E \mathsf{acc}(e)$.

Therefore, since $\langle H, \rho_n, \pi'_n \rangle \;\widetilde{\vDash}\; \langle \widetilde{\phi}_i^{p_2}, \mathsf{body}_{\mu_{p_2}} \rangle$, $\langle H, \rho_n, \pi'_n \rangle \vDash_E \mathsf{acc}(e)$, and $H, \rho_n \vdash e \Downarrow \texttt{false}$, by SsWhileFalse, we have
$\psi_2 = \langle H, \langle \rho_n, \pi'_n, \texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}}^{\texttt{p}_2} \texttt{ \{ r \}; } s'_{n_2} \rangle \cdot ... \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_2}$
$\langle H, \langle \rho_n, \pi'_n, s'_{n_2} \rangle \cdot ... \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_2$.

Finally, since

1) $s \sqsubseteq s'_{n_2}$,

2) $s_m \sqsubseteq s'_m$ for $1 \leq m < n$, and

3) $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$

we clearly have $\psi'_1 \lesssim \psi'_2$.

Then clearly, $\psi_2 \xrightarrow{\sim}_{p_2} \psi'_2$, with $\psi'_1 \lesssim \psi'_2$.

*Case 90 (*SsWhileTrue*).* The proof of this case is similar in structure to SsCall.

*Case 91 (*SsWhileFinish*).* We have
$\psi_1 = \langle H, \langle \rho_n, \pi_n, \texttt{skip} \rangle \cdot \langle \rho_{n-1}, \pi_{n-1}, \texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}}^{\texttt{p}_1} \texttt{ \{ r \}; } s \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_1} \langle H, \langle \rho_n, \pi_n \cup \pi_{n-1}, \texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}}^{\texttt{p}_1} \texttt{ \{ r \}; } s \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_1$.

Since $\texttt{skip} \sqsubseteq s'_n$, then by definition we get $s'_n = \texttt{skip}$.

Also, since $\texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}}^{\texttt{p}_1} \texttt{ \{ r \}; } s \sqsubseteq s'_{n-1}$, then by definition we get $s'_{n-1} = s'_{n-1_1} ; s'_{n-1_2}$ where $\texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}}^{\texttt{p}_1} \texttt{ \{ r \}} \sqsubseteq s'_{n-1_1}$ and $s \sqsubseteq s'_{n-1_2}$. Then $\texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}}^{\texttt{p}_1} \texttt{ \{ r \}} \sqsubseteq s'_{n-1_1}$ gives $s'_{n-1_1} = \texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}}^{\texttt{p}_2} \texttt{ \{ r \}}$ where $\widetilde{\phi}_i^{p_1} \sqsubseteq \widetilde{\phi}_i^{p_2}$,
and so $s'_{n-1} = \texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}}^{\texttt{p}_2} \texttt{ \{ r \}; } s'_{n-1_2}$.

Therefore, by SsWhileFinish, we have
$\psi_2 = \langle H, \langle \rho_n, \pi'_n, \texttt{skip} \rangle \cdot \langle \rho_{n-1}, \pi'_{n-1}, \texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}}^{\texttt{p}_2} \texttt{ \{ r \}; } s'_{n-1_2} \rangle \cdot ... \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_2}$
$\langle H, \langle \rho_n, \pi'_n \cup \pi'_{n-1}, \texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}}^{\texttt{p}_2} \texttt{ \{ r \}; } s'_{n-1_2} \rangle \cdot ... \cdot \langle \rho_1, \pi'_1, s'_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_2$.

Also, since $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$, we get $(\pi_n \cup \pi_{n-1}) \cup \bigcup_{i=m}^{n-2} \pi_i \subseteq (\pi'_n \cup \pi'_{n-1}) \cup \bigcup_{i=m}^{n-2} \pi'_i$ for $1 \leq m \leq n-1$.

Finally, since

1) $\texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}}^{\texttt{p}_1} \texttt{ \{ r \}; } s = s_{n-1} \sqsubseteq s'_{n-1} = \texttt{while (e) inv } \widetilde{\phi}_{\texttt{i}}^{\texttt{p}_2} \texttt{ \{ r \}; } s'_{n-1_2}$ and ,

2) $s_m \sqsubseteq s'_m$ for $1 \leq m < n-1$, and

3) $(\pi_n \cup \pi_{n-1}) \cup \bigcup_{i=m}^{n-2} \pi_i \subseteq (\pi'_n \cup \pi'_{n-1}) \cup \bigcup_{i=m}^{n-2} \pi'_i$ for $1 \leq m \leq n-1$

we clearly have $\psi'_1 \lesssim \psi'_2$.

Then clearly, $\psi_2 \xrightarrow{\sim}_{p_2} \psi'_2$, with $\psi'_1 \lesssim \psi'_2$.

*Case 92 (*SsFold*).* We have $\psi_1 = \langle H, \langle \rho_n, \pi_n, \texttt{fold } p(\overline{e}) \texttt{; } s \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle \xrightarrow{\sim}_{p_1} \langle H, \langle \rho_n, \pi_n, s \rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1 \rangle \cdot \mathsf{nil} \rangle = \psi'_1$.

Since $\texttt{fold}\ p(\overline{e});\ s \sqsubseteq s'_n$, then by definition we get $s'_n = s'_{n_1}\ ;\ s'_{n_2}$ where $\texttt{fold}\ p(\overline{e}) \sqsubseteq s'_{n_1}$ and $s \sqsubseteq s'_{n_2}$. Then $\texttt{fold}\ p(\overline{e}) \sqsubseteq s'_{n_1}$ gives $s'_{n_1} = \texttt{fold}\ p(\overline{e})$ or $s'_{n_1} = \texttt{skip}$, and so $s'_n = \texttt{fold}\ p(\overline{e});\ s'_{n_2}$ or $s'_n = \texttt{skip};\ s'_{n_2}$.

If $s'_n = \texttt{fold}\ p(\overline{e});\ s'_{n_2}$, then

by SsFold, we have $\psi_2 = \langle H, \langle \rho_n, \pi'_n, \texttt{fold}\ p(\overline{e});\ s'_{n_2}\rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim}_{p_2} \langle H, \langle \rho_n, \pi'_n, s'_{n_2}\rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1\rangle \cdot \mathsf{nil}\rangle = \psi'_2$.

If $s'_n = \texttt{skip};\ s'_{n_2}$, then

by SsSkip, we have $\psi_2 = \langle H, \langle \rho_n, \pi'_n, \texttt{skip};\ s'_{n_2}\rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim}_{p_2} \langle H, \langle \rho_n, \pi'_n, s'_{n_2}\rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1\rangle \cdot \mathsf{nil}\rangle = \psi'_2$.

In either case, $\psi_2 \xrightarrow{\sim}_{p_2} \langle H, \langle \rho_n, \pi'_n, s'_{n_2}\rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1\rangle \cdot \mathsf{nil}\rangle = \psi'_2$.

Finally, since

1) $s \sqsubseteq s'_{n_2}$,

2) $s_m \sqsubseteq s'_m$ for $1 \leq m < n$, and

3) $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$,

we clearly have $\psi'_1 \lesssim \psi'_2$.

Then clearly, $\psi_2 \xrightarrow{\sim}_{p_2} \psi'_2$, with $\psi'_1 \lesssim \psi'_2$.

*Case 93 (*SsUnfold*).* We have $\psi_1 = \langle H, \langle \rho_n, \pi_n, \texttt{unfold}\ p(\overline{e});\ s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim}_{p_1} \langle H, \langle \rho_n, \pi_n, s\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle = \psi'_1$.

Since $\texttt{unfold}\ p(\overline{e});\ s \sqsubseteq s'_n$, then by definition we get $s'_n = s'_{n_1}\ ;\ s'_{n_2}$ where $\texttt{unfold}\ p(\overline{e}) \sqsubseteq s'_{n_1}$ and $s \sqsubseteq s'_{n_2}$. Then $\texttt{unfold}\ p(\overline{e}) \sqsubseteq s'_{n_1}$ gives $s'_{n_1} = \texttt{unfold}\ p(\overline{e})$ or $s'_{n_1} = \texttt{skip}$, and so $s'_n = \texttt{unfold}\ p(\overline{e});\ s'_{n_2}$ or $s'_n = \texttt{skip};\ s'_{n_2}$.

If $s'_n = \texttt{unfold}\ p(\overline{e});\ s'_{n_2}$, then

by SsUnfold, we have $\psi_2 = \langle H, \langle \rho_n, \pi'_n, \texttt{unfold}\ p(\overline{e});\ s'_{n_2}\rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim}_{p_2} \langle H, \langle \rho_n, \pi'_n, s'_{n_2}\rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1\rangle \cdot \mathsf{nil}\rangle = \psi'_2$.

If $s'_n = \texttt{skip};\ s'_{n_2}$, then

by SsSkip, we have $\psi_2 = \langle H, \langle \rho_n, \pi'_n, \texttt{skip};\ s'_{n_2}\rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1\rangle \cdot \mathsf{nil}\rangle \xrightarrow{\sim}_{p_2} \langle H, \langle \rho_n, \pi'_n, s'_{n_2}\rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1\rangle \cdot \mathsf{nil}\rangle = \psi'_2$.

In either case, $\psi_2 \xrightarrow{\sim}_{p_2} \langle H, \langle \rho_n, \pi'_n, s'_{n_2}\rangle \cdot \ldots \cdot \langle \rho_1, \pi'_1, s'_1\rangle \cdot \mathsf{nil}\rangle = \psi'_2$.

Finally, since

1) $s \sqsubseteq s'_{n_2}$,

2) $s_m \sqsubseteq s'_m$ for $1 \leq m < n$, and

3) $\bigcup_{i=m}^{n} \pi_i \subseteq \bigcup_{i=m}^{n} \pi'_i$ for $1 \leq m \leq n$,

we clearly have $\psi'_1 \lesssim \psi'_2$.

Then clearly, $\psi_2 \xrightarrow{\sim}_{p_2} \psi'_2$, with $\psi'_1 \lesssim \psi'_2$.

8 $\mathrm{GVL_{RP}}$ Lemmas

Lemma 12 (Gradual Eval Preserved Reduced Precision). *If* $\langle H,\rho,\pi\rangle \mathrel{\widetilde{\vDash}} \langle\widetilde{\phi},\widetilde{\mathsf{body}}_\Delta\rangle$ *and* $\widetilde{\phi}\sqsubseteq\widetilde{\phi}'$*, then* $\langle H,\rho,\pi\rangle \mathrel{\widetilde{\vDash}} \langle\widetilde{\phi}',\widetilde{\mathsf{body}}_\Delta\rangle$.

Proof. Let $\langle H,\rho,\pi\rangle \in \textsc{Mem}$, $\widetilde{\phi},\widetilde{\phi}' \in \widetilde{\textsc{Formula}}$, and $\widetilde{\mathsf{body}}_\Delta \in \textsc{PredName} \rightarrow \textsc{Expr}^* \rightarrow \widetilde{\textsc{Formula}}$. $\langle H,\rho,\pi\rangle \mathrel{\widetilde{\vDash}} \langle\widetilde{\phi},\widetilde{\mathsf{body}}_\Delta\rangle$ and $\widetilde{\phi}\sqsubseteq\widetilde{\phi}'$.

Then, since $\langle H,\rho,\pi\rangle \mathrel{\widetilde{\vDash}} \langle\widetilde{\phi},\widetilde{\mathsf{body}}_\Delta\rangle$, by claim 3, we get that $\exists\,\langle\theta,\mathsf{body}_\Delta\rangle \in \gamma(\langle\widetilde{\phi},\widetilde{\mathsf{body}}_\Delta\rangle)$. $\langle H,\rho,\pi\rangle \vDash_E \langle\theta,\mathsf{body}_\Delta\rangle$.

Since $\langle\theta,\mathsf{body}_\Delta\rangle \in \gamma(\langle\widetilde{\phi},\widetilde{\mathsf{body}}_\Delta\rangle)$, we get that $\theta\in\gamma(\widetilde{\phi})$, $\mathsf{body}_\Delta \in \gamma(\widetilde{\mathsf{body}}_\Delta)$, and $\vdash_{\mathtt{frm}} \langle\theta,\mathsf{body}_\Delta\rangle$.

Also, by definition $\widetilde{\phi}\sqsubseteq\widetilde{\phi}'$ implies that $\gamma(\widetilde{\phi})\subseteq\gamma(\widetilde{\phi}')$. Therefore, $\theta\in\gamma(\widetilde{\phi})\subseteq\gamma(\widetilde{\phi}')$.

Since $\theta\in\gamma(\widetilde{\phi}')$, $\mathsf{body}_\Delta\in\gamma(\widetilde{\mathsf{body}}_\Delta)$, and $\vdash_{\mathtt{frm}} \langle\theta,\mathsf{body}_\Delta\rangle$, we get that $\langle\theta,\mathsf{body}_\Delta\rangle \in \gamma(\langle\widetilde{\phi}',\widetilde{\mathsf{body}}_\Delta\rangle)$

Since $\langle\theta,\mathsf{body}_\Delta\rangle \in \gamma(\langle\widetilde{\phi}',\widetilde{\mathsf{body}}_\Delta\rangle)$ and $\langle H,\rho,\pi\rangle \vDash_E \langle\theta,\mathsf{body}_\Delta\rangle$, we get by claim 3 that $\langle H,\rho,\pi\rangle \mathrel{\widetilde{\vDash}} \langle\widetilde{\phi}',\widetilde{\mathsf{body}}_\Delta\rangle$.

Lemma 13 (Gradual Eval Preserved Reduced Precision for Body Context). *If* $\mathsf{dom}(\widetilde{\mathsf{body}}_\Delta) = \mathsf{dom}(\widetilde{\mathsf{body}}'_\Delta)$*,* $\forall p\in\mathsf{dom}(\widetilde{\mathsf{body}}_\Delta)$. $\forall\overline{tmp}\in\textsc{Var}^*$. $\widetilde{\mathsf{body}}_\Delta(p)(\overline{tmp}) \sqsubseteq \widetilde{\mathsf{body}}'_\Delta(p)(\overline{tmp})$*, and* $\langle H,\rho,\pi\rangle \mathrel{\widetilde{\vDash}} \langle\widetilde{\phi},\widetilde{\mathsf{body}}_\Delta\rangle$*, then* $\langle H,\rho,\pi\rangle \mathrel{\widetilde{\vDash}} \langle\widetilde{\phi},\widetilde{\mathsf{body}}'_\Delta\rangle$.

Proof. Let $\langle H,\rho,\pi\rangle \in \textsc{Mem}$, $\widetilde{\phi} \in \widetilde{\textsc{Formula}}$, and $\widetilde{\mathsf{body}}_\Delta,\widetilde{\mathsf{body}}'_\Delta \in \textsc{PredName} \rightarrow \textsc{Expr}^* \rightarrow \widetilde{\textsc{Formula}}$. $\langle H,\rho,\pi\rangle \mathrel{\widetilde{\vDash}} \langle\widetilde{\phi},\widetilde{\mathsf{body}}_\Delta\rangle$, $\mathsf{dom}(\widetilde{\mathsf{body}}_\Delta) = \mathsf{dom}(\widetilde{\mathsf{body}}'_\Delta)$, and $\forall p\in\mathsf{dom}(\widetilde{\mathsf{body}}_\Delta)$. $\forall\overline{tmp}\in\textsc{Var}^*$. $\widetilde{\mathsf{body}}_\Delta(p)(\overline{tmp}) \sqsubseteq \widetilde{\mathsf{body}}'_\Delta(p)(\overline{tmp})$.

Then, since $\langle H,\rho,\pi\rangle \mathrel{\widetilde{\vDash}} \langle\widetilde{\phi},\widetilde{\mathsf{body}}_\Delta\rangle$, by claim 3, we get that $\exists\,\langle\theta,\mathsf{body}_\Delta\rangle \in \gamma(\langle\widetilde{\phi},\widetilde{\mathsf{body}}_\Delta\rangle)$. $\langle H,\rho,\pi\rangle \vDash_E \langle\theta,\mathsf{body}_\Delta\rangle$.

Since $\langle\theta,\mathsf{body}_\Delta\rangle \in \gamma(\langle\widetilde{\phi},\widetilde{\mathsf{body}}_\Delta\rangle)$, we get that $\theta\in\gamma(\widetilde{\phi})$, $\mathsf{body}_\Delta \in \gamma(\widetilde{\mathsf{body}}_\Delta)$, and $\vdash_{\mathtt{frm}} \langle\theta,\mathsf{body}_\Delta\rangle$.

$\mathsf{body}_\Delta\in\gamma(\widetilde{\mathsf{body}}_\Delta)$, gives $\mathsf{body}_\Delta = \lambda p_i\in\mathsf{dom}(\widetilde{\mathsf{body}}_\Delta)$. $\lambda\overline{e}\in\textsc{Expr}^*$. $\theta_{p_i}[\overline{e}/\overline{tmp_i}]$ such that $\forall p_i\in\mathsf{dom}(\widetilde{\mathsf{body}}_\Delta)$. $\theta_{p_i}\in\gamma(\widetilde{\mathsf{body}}_\Delta(p_i)(\overline{tmp_i}))$ and $\forall p_i\in\mathsf{dom}(\widetilde{\mathsf{body}}_\Delta)$. $\vdash_{\mathtt{frm}} \langle\mathsf{body}_\Delta(p_i)(\overline{tmp_i}),\mathsf{body}_\Delta\rangle$.

Since $\forall p\in\mathsf{dom}(\widetilde{\mathsf{body}}_\Delta)$. $\forall\overline{tmp}\in\textsc{Var}^*$. $\widetilde{\mathsf{body}}_\Delta(p)(\overline{tmp}) \sqsubseteq \widetilde{\mathsf{body}}'_\Delta(p)(\overline{tmp})$, we get $\forall p\in\mathsf{dom}(\widetilde{\mathsf{body}}_\Delta)$. $\forall\overline{tmp}\in\textsc{Var}^*$. $\gamma(\widetilde{\mathsf{body}}_\Delta(p)(\overline{tmp})) \subseteq \gamma(\widetilde{\mathsf{body}}'_\Delta(p)(\overline{tmp}))$.

Therefore, $\forall p_i\in\mathsf{dom}(\widetilde{\mathsf{body}}_\Delta)$. $\theta_{p_i}\in\gamma(\widetilde{\mathsf{body}}_\Delta(p_i)(\overline{tmp_i})) \subseteq \gamma(\widetilde{\mathsf{body}}'_\Delta(p_i)(\overline{tmp_i}))$. That is, $\forall p_i\in\mathsf{dom}(\widetilde{\mathsf{body}}_\Delta)$. $\theta_{p_i}\in\gamma(\widetilde{\mathsf{body}}'_\Delta(p_i)(\overline{tmp_i}))$.

Since $\mathsf{dom}(\widetilde{\mathsf{body}}_\Delta) = \mathsf{dom}(\widetilde{\mathsf{body}}'_\Delta)$, we get $\mathsf{body}_\Delta = \lambda p_i\in\mathsf{dom}(\widetilde{\mathsf{body}}'_\Delta)$. $\lambda\overline{e}\in\textsc{Expr}^*$. $\theta_{p_i}[\overline{e}/\overline{tmp_i}]$ such that $\forall p_i\in\mathsf{dom}(\widetilde{\mathsf{body}}'_\Delta)$. $\theta_{p_i}\in\gamma(\widetilde{\mathsf{body}}'_\Delta(p_i)(\overline{tmp_i}))$ and

$\forall p_i\in\mathsf{dom}(\widetilde{\mathsf{body}}'_\Delta)$. $\vdash_{\mathtt{frm}} \langle\mathsf{body}_\Delta(p_i)(\overline{tmp_i}),\mathsf{body}_\Delta\rangle$. Therefore, $\mathsf{body}_\Delta\in\gamma(\widetilde{\mathsf{body}}'_\Delta)$.

Since $\theta \in \gamma(\widetilde{\phi})$, $\mathsf{body}_\Delta \in \gamma(\widetilde{\mathsf{body}}'_\Delta)$, and $\vdash_{\mathtt{frm}} \langle \theta, \mathsf{body}_\Delta \rangle$, we get that $\langle \theta, \mathsf{body}_\Delta \rangle \in \gamma(\langle \widetilde{\phi}, \widetilde{\mathsf{body}}'_\Delta \rangle)$.

Since $\langle \theta, \mathsf{body}_\Delta \rangle \in \gamma(\langle \widetilde{\phi}, \widetilde{\mathsf{body}}'_\Delta \rangle)$ and $\langle H, \rho, \pi \rangle \vDash_E \langle \theta, \mathsf{body}_\Delta \rangle$, we get by claim 3 that $\langle H, \rho, \pi \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}, \widetilde{\mathsf{body}}'_\Delta \rangle$.

Lemma 14 (Gradual Eval Preserved Dynamic Footprint Larger). *If $\langle H, \rho, \pi \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}, \widetilde{\mathsf{body}}_\Delta \rangle$ and $\pi \subseteq \pi'$, then $\langle H, \rho, \pi' \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}, \widetilde{\mathsf{body}}_\Delta \rangle$.*

Proof. Let $\langle H, \rho, \pi \rangle \in \textsc{Mem}$, $\pi' \in \textsc{DynFprint}$, $\widetilde{\phi} \in \widetilde{\textsc{Formula}}$, and $\widetilde{\mathsf{body}}_\Delta \in \textsc{PredName} \rightarrow \textsc{Expr}^* \rightarrow \widetilde{\textsc{Formula}}$. $\langle H, \rho, \pi \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}, \widetilde{\mathsf{body}}_\Delta \rangle$ and $\pi \subseteq \pi'$.

Then, since $\langle H, \rho, \pi \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}, \widetilde{\mathsf{body}}_\Delta \rangle$, by claim 3, we get that $\exists\, \langle \theta, \mathsf{body}_\Delta \rangle \in \gamma(\langle \widetilde{\phi}, \widetilde{\mathsf{body}}_\Delta \rangle)$. $\langle H, \rho, \pi \rangle \vDash_E \langle \theta, \mathsf{body}_\Delta \rangle$.

Since $\langle H, \rho, \pi \rangle \vDash_E \langle \theta, \mathsf{body}_\Delta \rangle$ and $\pi \subseteq \pi'$, by lemma 16, we get $\langle H, \rho, \pi' \rangle \vDash_E \langle \theta, \mathsf{body}_\Delta \rangle$.

Finally, since $\langle \theta, \mathsf{body}_\Delta \rangle \in \gamma(\langle \widetilde{\phi}, \widetilde{\mathsf{body}}_\Delta \rangle)$ and $\langle H, \rho, \pi' \rangle \vDash_E \langle \theta, \mathsf{body}_\Delta \rangle$, we get by claim 3 that $\langle H, \rho, \pi' \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}, \widetilde{\mathsf{body}}_\Delta \rangle$.

Lemma 15 (Gradual Dynamic Footprint Calc Preserved Across Programs). *Let $p_1, p_1 \in \textsc{State}$ such that $p_1 \sqsubseteq_d p_2$. If $\lfloor \widetilde{\phi} \rceil^{p_1}_{\pi, H, \rho} = \pi_1$, $\widetilde{\phi} \sqsubseteq \widetilde{\phi}'$, and $\pi_1 \subseteq \pi \subseteq \pi'$, then $\lfloor \widetilde{\phi}' \rceil^{p_2}_{\pi', H, \rho} = \pi_2$ and $\pi_1 \subseteq \pi_2$.*

Proof. The proof of this lemma can be found here: extra-proofs/gvlrp-lemma-dynfpcalc-acrossprograms.pdf.

Lemma 16 (Equi Permissions Supersets & Formula Evaluation With Body Context). *If $\pi \subseteq \pi'$ and $\langle H, \rho, \pi \rangle \vDash_E \langle \phi, \mathsf{body}_\Delta \rangle$, then $\langle H, \rho, \pi' \rangle \vDash_E \langle \phi, \mathsf{body}_\Delta \rangle$.*

Proof. This proof is similar to the proof of lemma 1 (its SVL_{RP} counterpart without body_Δ), body_Δ is just carried around in this proof.

Lemma 17 (Permission Erasure Subset Preservation with Body Context). *If $\Pi \subseteq \Pi'$ and $\langle\langle\, \langle \Pi', \mathsf{body}_\Delta \rangle\, \rangle\rangle_H$ is defined, then $\langle\langle\, \langle \Pi, \mathsf{body}_\Delta \rangle\, \rangle\rangle_H$ is defined and $\langle\langle\, \langle \Pi, \mathsf{body}_\Delta \rangle\, \rangle\rangle_H \subseteq \langle\langle\, \langle \Pi', \mathsf{body}_\Delta \rangle\, \rangle\rangle_H$.*

Proof. This proof is similar to the proof of lemma 4 (its SVL_{RP} counterpart without body_Δ), body_Δ just replaces body_μ in the proof.

Lemma 18 (Formula Footprint & Evaluation with Body Context). *If $\langle\langle\, \langle \lfloor \phi \rfloor_{H,\rho}, \mathsf{body}_\Delta \rangle\, \rangle\rangle_H$ is defined, $\langle\langle\, \langle \lfloor \phi \rfloor_{H,\rho}, \mathsf{body}_\Delta \rangle\, \rangle\rangle_H = \pi'$, and $\langle H, \rho, \pi \rangle \vDash_E \langle \phi, \mathsf{body}_\Delta \rangle$, then $\langle H, \rho, \pi' \rangle \vDash_E \langle \phi, \mathsf{body}_\Delta \rangle$.*

Proof. This proof is similar to the proof of lemma 5 (its SVL_{RP} counterpart without body_Δ), body_Δ just replaces body_μ and is carried around in the proof. The lemmas referenced in the SVL_{RP} proof are replaced with their body_Δ containing counterparts.

Lemma 19 **(Acc(e) Eval Preserved Without Body Context).** *If* $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e), \mathsf{body}_\Delta \rangle$, *then* $\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e)$.

Proof. Let $\langle H, \rho, \pi \rangle \in$ MEM, $e \in$ EXPR, and $\mathsf{body}_\Delta \in$ PREDNAME $\rightarrow$ EXPR* $\rightarrow$ FORMULA . $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e), \mathsf{body}_\Delta \rangle$.

Then, we will prove $\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e)$, by induction on the syntax of e.

Case 94 *($e = v$).* Then, $\mathsf{acc}(e) = \mathsf{acc}(v) = \texttt{true}$ by definition. By axiom rule EVTRUE-EXPR, we get $\langle H, \rho, \pi \rangle \vDash_E \texttt{true}$.

Case 95 *($e = x$).* Then, $\mathsf{acc}(e) = \mathsf{acc}(x) = \texttt{true}$ by definition. By axiom rule EVTRUE-EXPR, we get $\langle H, \rho, \pi \rangle \vDash_E \texttt{true}$.

Case 96 *($e = e_1 \oplus e_2$).* Then, $\mathsf{acc}(e) = \mathsf{acc}(e_1 \oplus e_2) = \mathsf{acc}(e_1) \wedge \mathsf{acc}(e_2)$ by definition.

So, we have $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e_1) \wedge \mathsf{acc}(e_2), \mathsf{body}_\Delta \rangle$, and by inversion we get that $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e_1), \mathsf{body}_\Delta \rangle$ and $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e_2), \mathsf{body}_\Delta \rangle$.

Then, by the IH on $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e_1), \mathsf{body}_\Delta \rangle$ and $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e_2), \mathsf{body}_\Delta \rangle$, we get that $\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e_1)$ and $\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e_2)$.

Finally, by EVANDOP, we get $\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e_1) \wedge \mathsf{acc}(e_2)$.

Case 97 *($e = e_1 \odot e_2$).* Then, $\mathsf{acc}(e) = \mathsf{acc}(e_1 \odot e_2) = \mathsf{acc}(e_1) \wedge \mathsf{acc}(e_2)$ by definition.

So, we have $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e_1) \wedge \mathsf{acc}(e_2), \mathsf{body}_\Delta \rangle$, and by inversion we get that $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e_1), \mathsf{body}_\Delta \rangle$ and $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e_2), \mathsf{body}_\Delta \rangle$.

Then, by the IH on $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e_1), \mathsf{body}_\Delta \rangle$ and $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e_2), \mathsf{body}_\Delta \rangle$, we get that $\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e_1)$ and $\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e_2)$.

Finally, by EVANDOP, we get $\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e_1) \wedge \mathsf{acc}(e_2)$.

Case 98 *($e = e_1.f$).* Then, $\mathsf{acc}(e) = \mathsf{acc}(e_1.f) = \mathsf{acc}(e_1) \wedge \textbf{acc}(\mathtt{e_1.f})$ by definition.

So, we have $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e_1) \wedge \textbf{acc}(\mathtt{e_1.f}), \mathsf{body}_\Delta \rangle$, and by inversion we get that $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e_1), \mathsf{body}_\Delta \rangle$ and $\langle H, \rho, \pi \rangle \vDash_E \langle \textbf{acc}(\mathtt{e_1.f}), \mathsf{body}_\Delta \rangle$.

Then, by the IH on $\langle H, \rho, \pi \rangle \vDash_E \langle \mathsf{acc}(e_1), \mathsf{body}_\Delta \rangle$, we get that $\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e_1)$.

Also, by inversion on $\langle H, \rho, \pi \rangle \vDash_E \langle \textbf{acc}(\mathtt{e_1.f}), \mathsf{body}_\Delta \rangle$, we get that $H, \rho \vdash e_1 \Downarrow o$, $H, \rho \vdash e_1 . f \Downarrow v$, and $\langle o, f \rangle \in \pi$. By EVACC, we get that $\langle H, \rho, \pi \rangle \vDash_E \textbf{acc}(\mathtt{e_1.f})$.

Finally, by EVANDOP, we get $\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e_1) \wedge \textbf{acc}(\mathtt{e_1.f})$.