A DETAILS FOR SECTION 3

A.1 Generating Quantifier-Free Verification Conditions

The FWYB methodology described in previous sections shows that we can soundly reduce the problem of verifying programs with intrinsic specifications to the problem of verifying programs (with ghost code) with quantifier-free contracts. We then argue that we can reason with the latter using combinations of various quantifier-free theories including sets and maps with pointwise updates. In this section we detail some subtleties involved in the argument.

It is well-known that we can reason with scalar programs with quantifier-free contracts by generating quantifier-free verification conditions (which in turn can be handled by SMT solvers). However, this is not immediately clear for programs that dynamically manipulate heaps. In particular, commands such as allocation and function calls pose challenges in formulating quantifier-free verification conditions.

At a high level, our solution transforms the given heap program into a scalar program that explicitly encodes changes to the heap. Specifically, we show an encoding for the field mutation, allocation, and function call statements.

Modeling Field Mutation. As described earlier, we model the monadic maps and fields as updatable maps [18]. Formally, we introduce a map M_f (also called an *array* in SMT solvers like Z3 [17]) for every field/monadic map f. We then encode the commands for field lookup and mutation as map operations. For example, the mutation $x.f := y$ is encoded as $M_f[x] := y$.

Modeling Allocation. We model programs in a *safe* garbage-collected programming language[6]. We introduce a ghost global variable *Alloc* to model the allocated set of objects in the program. We then add several assumptions (i.e., assume statements) throughout the program. Specifically, we assume for every program parameter of type Object, the parameter itself as well as the values of the monadic maps of type Object/Set-of-Objects on the parameter are all contained in *Alloc*. For example, in the case of our running example (Example 3.4), we add the assumptions $x \in Alloc$ and $next(x) \neq nil \Rightarrow next(x) \in Alloc$. If we had a monadic map *hslist* corresponding to the heaplet of the sorted list, we would also add the assumption $hslist(x) \subseteq Alloc$. Similarly, whenever an object is dereferenced on a field of type Object/Set-of-Objects in the program, we add an assumption that the resulting value is contained in *Alloc*. Note that these are quantifier-free assumptions. They can be added soundly since they are valid under the semantics of the underlying language.

We then model allocation by introducing a new object to *Alloc* and ensure that the default values of the various fields on the newly allocated object belong to *Alloc*. These constraints can be expressed using a quantifier-free formula over maps.

Modeling Heap Change Across Function Calls. The main challenge in modeling function calls is to ensure the ability to do frame reasoning. To do this, we extend the programming language with a *modified set* annotation for methods. We require the modified set to be a term of type Set-of-Objects that is constructed using object variables in the current scope and monadic maps over them. In the case of our running example (Example 3.1), we would add a monadic map *hslist* of type Set-of-Objects corresponding to the heaplet of the sorted list and annotate the program with $hslist(x)$ as the modified set. Figure 5 shows the full version of sorted list insertion with the modified set annotation.

Given a modified set *Mod*, we model changes to the heap across a function call by introducing new maps corresponding to the various fields (including monadic maps) after the call. We then add assumptions that the values of the new maps are equal to the values of the maps before the call on all locations that do not belong to the modified set *Mod*. Although this constrains the maps on

[6]this means that dereferencing an allocated object can never yield an un-allocated object.

unboundedly many objects, it can be written without quantifiers by using pointwise operators on maps [18]. Formally, for a field f modeled as a map M_f, we introduce a new map M'_f and update M_f as:

$$M_f[x] := ite(x \in Mod,\ M'_f[x],\ M_f[x])$$

The above update can be expressed using pointwise operators as $M_f := ite(Mod, M'_f, M_f)$, where the *ite* operator is applied pointwise over the maps Mod, M_f, and M'_f. The value of the field f on an object x after the call will then be equal to $x.f$ before the call if x was not modified, and a *havoc*-ed value given by M'_f otherwise.

Program verifiers like Boogie [52] offer VC generation frameworks that are amenable to the modeling described in this section. Indeed, our implementation of the IDS/FWYB methodology described in Section 5.1 uses Boogie.

B PROOF OF THEOREM 3.6

We provide here the gist of a proof of Theorem 3.6, which states that the FWYB methodology is sound. We assume that there is only one method for simplicity.

Fix an intrinsic definition $(\mathcal{G}, LC, \varphi)$ where $\mathcal{G} = \{g_1, g_2 \ldots, g_l\}$ and $LC \equiv \forall z.\, \rho(z)$. Let M be a method and $P_{\mathcal{G},Br}$ be its body such that $\vdash_{\text{WB}} P_{\mathcal{G},Br}$. Let ψ_{pre} and ψ_{post} be quantifier-free formula that do not mention Br (but they can mention the maps in $\mathcal{G}$).

Then, if the following triple is valid:

$$\{\varphi \land \psi_{pre} \land Br = \emptyset\}\ P_{\mathcal{G},Br}\ \{\varphi \land \psi_{post} \land Br = \emptyset\}$$

then we must show that the following triple is valid:

$$\langle\, \exists\, g_1, g_2 \ldots, g_l.\, (LC \land \varphi \land \psi_{pre})\,\rangle\ P\ \langle\, \exists\, g_1, g_2 \ldots, g_l.\, (LC \land \varphi \land \psi_{post})\,\rangle$$

where P is the projection of $P_{\mathcal{G},Br}$ to user-level code.

The core of the proof is the argument that for well-behaved programs, if

$$\{\varphi \land \psi_{pre} \land Br = \emptyset\}\ P_{\mathcal{G},Br}\ \{\varphi \land \psi_{post} \land Br = \emptyset\}$$

is valid then

$$\langle\, (\forall z \notin Br.\, \rho) \land \varphi \land \psi_{pre} \land Br = \emptyset\,\rangle\ P_{\mathcal{G},Br}\ \langle (\forall z \notin Br.\, \rho) \land \varphi \land \psi_{post} \land Br = \emptyset\,\rangle$$

is valid.

If we show that the latter is valid, then we can rewrite it by removing Br, showing the validity of the following triple:

$$\langle\, LC \land \varphi \land \psi_{pre}\,\rangle\ P_{\mathcal{G},Br}\ \ LC \land \varphi \land \psi_{post}\,\rangle$$

Observe that the specifications now do not mention Br. We then apply Proposition 3.3 to obtain the validity of the desired intrinsic triple. Therefore, we dedicate the rest of the section to showing the above property of well-behaved programs.

The proof proceeds by an induction on the nesting depth of method calls in a trace of the program $P_{\mathcal{G},Br}$. We elide this level of induction here because it is routine. Importantly, given a particular execution of the program P, we must show that the claim holds, assuming it holds for all method calls occurring in the execution. We show this by structural induction on the proof of well-behavedness of $P_{\mathcal{G},Br}$.

There are several base cases.

Skip/Assignment/Lookup/Return There is nothing to show for skip, assignment, lookup, or return statements. These do not change the heap at all and the rule does not update Br either, therefore if $\langle\, \alpha \,\rangle$ stmt $\langle\, \beta \,\rangle$ is valid then certainly $\langle\, (\forall z \notin Br.\, \rho) \wedge \alpha \,\rangle$ stmt $\langle\, (\forall z \notin Br.\, \rho) \wedge \beta \,\rangle$ is valid.

Mutation The claim is true for the mutation rule since by the premise of the rule we update the broken set with the impact set consisting of all potential objects where local conditions may not hold.

Function Call Here we simply appeal to the induction hypothesis.

Allocation We refer to our operational semantics, which ensures that no object points to a freshly allocated object. Therefore, the allocation of an object could have only broken the local conditions on itself at most.

Infer LC Outside Br There is nothing to prove for this rule as it does not alter the *Br* set at all.

Assert LC and Remove The claim holds for this rule by construction. If *LC* holds everywhere outside *Br* , and we know that $LC(x)$ holds, then we can conclude that *LC* holds everywhere outside $Br \setminus \{x\}$.

It only remains to show that the claim holds for larger well-behaved programs obtained by composing smaller well-behaved programs using sequencing, branching, or looping constructs. The proof here is trivial as the argument for sequencing is trivial (we can think of a loop as unboundedly many sequenced compositions of the smaller well-behaved program): we can *always* compose two well-behaved programs to obtain a well-behaved program.

C DETAILS FOR SECTION 4

In this appendix we provide further details for the various case studies discussed in the main text and detail some other case studies not featured in the main text.

C.1 Discussion on Sorted List Insertion (Section 4.1)

We provide the specifications and the code augmented with ghost annotations in Figure 5.

Specifications. The precondition states that the broken set is empty at the beginning of the program. The postcondition states that the returned object r satisfies the local conditions and satisfies the correlation formula for a sorted list (i.e., $prev(r) = nil$). However, the broken set is only empty if the input object x was the head of a sorted list, and it is $\{prev(x)\}$ otherwise. The other conjuncts express functional specifications for insertion in terms of the length, heaplet, and set of keys. We also add a ‘modifies’ clause which enables program verifiers for heap manipulating programs to utilize frame reasoning across function calls.

Summary. The proof works at a high-level as follows: we recurse down the list, reaching the appropriate object x before which the new key must be inserted. This is the first branch in Figure 5, and we show the broken set at each point in the comments to the right. We create the new object z with the appropriate key and point $z.next$ to x. We then fix the local conditions on x and z. However, these fixes break the *LC* on $old(prev(x))$. We maintain this property up the recursion, at each point fixing *LC* on x and breaking it on $old(prev(x))$ in the process. This is shown in the last branch in the code. We eventually reach the head of the sorted list, whose *prev* in the pre state is *nil*, and at that point the fixes do not break anything else, i.e., the broken set is empty (as desired).

The verification engineer adds ghost code to perform these fixes as shown in blue in Figure 5. We can also see that there are essentially as many lines of ghost code as there are lines of user code; we compare these values across our benchmark suite (see Table 2) and find that this is typical for many methods. However, the verification conditions for the (augmented) program are *decidable* because they can be stated using quantifier-free formulas over decidable combinations of theories including maps, map updates, and sets.

```
pre:  Br = ∅
post: LC(r) ∧ prev(r) = nil
      ∧ Br = ite(old(prev(x)) = nil, ∅, {old(prev(x))})
      ∧ length(r) = old(length(x)) + 1
      ∧ keys(r) = old(keys(x)) ∪ {k}
      ∧ old(hslist(x)) ⊂ hslist(r)
modifies: hslist(x)
sorted_list_insert(x: C, k: Int, Br: Set(C))
returns r: C, Br: Set(C)
{
  InferLCOutsideBr(x, Br);
  if (x.key ≥ k) then { // k inserted before x
    NewObj(z, Br);          // {z}
    Mut(z, key, k, Br);    // {z} since z.prev = nil
    Mut(z, next, x, Br);   // {z} since z.next = nil
    Mut(z, hslist, {z} ∪ x.hslist, Br); // {z}
    Mut(z, length, 1 + x.length, Br);    // {z}
    Mut(z, keys,  {k} ∪ x.keys, Br);     // {z}
    Mut(x, prev, z, Br);     // {z, x, old(prev(x))}
    AssertLCAndRemove(z, Br);  // {x, old(prev(x))}
    AssertLCAndRemove(x, Br); // {old(prev(x))}
    r := z;
  }
  else {
  if (x.next = nil) then { // one-element list
    NewObj(z, Br);
    Mut(z, key, k, Br);
    Mut(z, next, nil, Br);
    Mut(z, hslist, {z}, Br);
    Mut(z, length, 1, Br);
    Mut(z, keys, {k}, Br);
    Mut(x, next, z, Br);
    Mut(z, prev, x, Br);
    AssertLCAndRemove(z, Br);
    Mut(x, prev, nil, Br);
    Mut(x, hslist, {x} ∪ {z}, Br);
    Mut(x, length, 2, Br);
    Mut(x, keys, {x.key} ∪ {k}, Br);
    AssertLCAndRemove(x, Br);
    r := x;
  }
  else { // recursive case
    y := x.next;
    InferLCOutsideBr(y, Br);
    tmp, Br := sorted_list_insert(y, k, Br);   // {x}
    InferLCOutsideBr(y, Br);
    if (y.prev = x) then {
      Mut(y, prev, nil, Br);    // {y, x}
    }
    Mut(x, next, tmp, Br);      // {y, x}
    AssertLCAndRemove(y, Br);   // {x}
    Mut(tmp, prev, x, Br);      // {tmp, x}
    AssertLCAndRemove(tmp, Br); // {x}
    Mut(x, hslist, {x} ∪ tmp.hslist, Br);  // {x, prev(x)}
    Mut(x, length, 1 + tmp.length, Br);    // {x, prev(x)}
    Mut(x, keys, {x.key} ∪ tmp.keys, Br);  // {x, prev(x)}
    Mut(x, prev, nil, Br);        // {x, old(prev(x))}
    AssertLCAndRemove(x, Br);   // {old(prev(x))}
    r := x;
  }}
}
```

Fig. 5. Code for insertion into a sorted list written in the syntactic fragment for well-behaved programs(Section 4.1). Black lines denote code written by the user, and blue lines denote lines written by the verification engineer. The comments on the right show the state of the broken set *Br* after the statement on the corresponding line.

C.2 BST Right-Rotation

We now turn to another data structure and method that illustrates intrinsic definitions for trees, namely verifying a right rotate on a binary search tree. Such an operation is a common tree operation, and rotations are used widely in maintaining balanced search trees, such as AVL and Red-Black Trees, on which several of our benchmarks operate.

We augment the definition of binary trees discussed in Section 1 to include the $min : BST \longrightarrow Real$ and $max : BST \longrightarrow Real$ maps, which capture the minimum and maximum keys stored in the tree rooted at a node, to help enforce binary search tree properties locally. The local condition and the impact sets are as below:

$$
\begin{aligned}
&LC \equiv \forall x.\, min(x) \le key(x) \le max(x)\\
&\wedge (p(x) \neq nil \Rightarrow l(p(x)) = x \vee r(p(x)) = x)\\
&\wedge (l(x) = nil \Rightarrow min(x) = key(x))\\
&\wedge (l(x) \neq nil \Rightarrow p(l(x)) = x \wedge rank(l(x)) < rank(x)\\
&\qquad\qquad \wedge\ max(l(x)) < key(x)\ \wedge\ min(x) = min(l(x)))\\
&\wedge (r(x) = nil \Rightarrow max(x) = key(x))\\
&\wedge (r(x) \neq nil \Rightarrow p(r(x)) = x \wedge rank(r(x)) < rank(x)\\
&\qquad\qquad \wedge\ min(r(x)) > key(x)\ \wedge\ max(x) = max(r(x)))
\end{aligned}
$$

Mutated Field f	Impacted Objects A_f
l	$\{x, old(l(x))\}$
r	$\{x, old(r(x))\}$
p	$\{x, old(p(x))\}$
key	$\{x\}$
min	$\{x, p(x)\}$
max	$\{x, p(x)\}$
$rank$	$\{x, p(x)\}$

We first describe the gist of how the data structure is repaired and provide the fully annotated program below. Recall that in a BST right rotation, that there are two nodes x and y such that y

is x's left child. After the rotation is performed, y becomes the new root of the subtree, while x becomes y's right child. Several routine updates of the monadic map p (parent) will have to be made. The most interesting update is that of the $rank : BST \longrightarrow Real$ map. Since y is now the root of the affected subtree, its rank must be greater than all its children. One way of doing this is to increase y's rank to something greater than x's rank. This works if y has no parent, but not in general. To solve this issue, we use the density of the Reals to set the rank of y to $(rank(x) + rank(p(y)))/2$. Note that there are a fixed number of ghost map updates, as the various monadic maps for distant ancestors and descendents of x, y do not change (the min/max of subtrees of such nodes do not change).

We present the fully annotated program below, with comments displaying the state of the broken set Br at the corresponding point in the program.

```
pre:  $Br = \emptyset \ \wedge \ l(x) \neq nil \ \wedge \ p(x) = xp$
post: $Br' = \emptyset \ \wedge \ p(ret) = xp$
      $\wedge \ l(ret) = old(l(l(x))) \ \wedge \ ret = old(l(x)) \ \wedge \ r(ret) = x$
      $\wedge \ l(r(ret)) = old(r(l(x))) \ \wedge \ r(r(ret)) = old(r(x))$
bst_right_rotate(x: BST, xp: BST?, Br: Set(BST))
returns ret: BST, Br: Set(BST)
{
  LCOutsideBr(x, Br);
  if (xp ≠ nil) then {
       LCOutsideBr(xp, Br);
  }
  if (x.l ≠ nil) then {
       LCOutsideBr(x.l, Br);
  }
  if (x.l ≠ nil ∧ x.l.r ≠ nil) then {
       LCOutsideBr(x.l.r, Br);
  }
  var y := x.l;                // {}
  Mut(x, l, y.r, Br);          // {x, y}
  if (xp ≠ nil) then {
       if (x = xp.l) then {
           Mut(xp, l, y, Br);  // {xp, x, y}
       }
       else {
           Mut(xp, r, y, Br);  // {xp, x, y}
       }
  }
  Mut(y, r, x, Br);            // {xp, x, y, x.l} (Note: x.l == old(y.r))
  // (1): Repairing x.l
  if (x.l ≠ nil) then {
       Mut(x.l, p, x, Br);     // {xp, x, y, x.l}
  }
  // (2): Repairing x
  Mut(x, p, y, Br);            // {xp, x, y, x.l}
  Mut(x, min, if x = nil then x.k else x.l.min, Br);    // {xp, x, y, x.l}
  // (3): Repairing y
  Mut(y, p, xp, Br);           // {xp, x, y, x.l}
  Mut(y, max, x.max, Br);      // {xp, x, y, x.l}
  Mut(y, rank, if xp = nil then x.rank+1 else (xp.rank+x.rank)/2, Br);     // {xp, x, y, x.l}
  AssertLCAndRemove(x.l, Br);  // {xp, x, y}
  AssertLCAndRemove(x, Br);    // {xp, y}
  AssertLCAndRemove(y, Br);    // {xp}
  AssertLCAndRemove(xp, Br);   // {}
  ret := y // return y
}
```

C.3 Discussion on Sorted List Reversal (Section 4.2)

What follows are the complete local conditions and impact sets for Sorted List Reverse:

$$
\begin{aligned}
LC \equiv\ & \forall x.\, prev(x) \neq nil \Rightarrow next(prev(x)) = x \\
& \wedge\ next(x) \neq nil \Rightarrow prev(next(x)) = x \\
& \qquad \wedge\ length(x) = length(next(x)) + 1 \\
& \qquad \wedge\ keys(x) = keys(next(x)) \cup \{key(x)\} \\
& \qquad \wedge\ hslist(x) = hslist(next(x)) \uplus \{x\} \\
& \qquad \wedge\ sorted(x) \Rightarrow key(x) \leq key(next(x)) \\
& \qquad\qquad \wedge\ sorted(x) = sorted(next(x)) \\
& \qquad \wedge\ rev\_sorted(x) \Rightarrow key(x) \geq key(next(x)) \\
& \qquad\qquad \wedge\ rev\_sorted(x) = rev\_sorted(next(x)) \\
& \wedge\ (next(x) = nil \Rightarrow length(x) = 1 \wedge keys(x) = \{x\} \wedge hslist(x) = \{x\})
\end{aligned}
$$

Fig. 7. Full local condition for lists for Sorted List Reverse

Mutated Field f	Impacted Objects A_f
$next$	$\{x, old(next(x))\}$
key	$\{x, prev(x)\}$
$prev$	$\{x, old(prev(x))\}$
$length$	$\{x, prev(x)\}$
$keys$	$\{x, prev(x)\}$
$hslist$	$\{x, prev(x)\}$
$sorted$	$\{x, prev(x)\}$
rev_sorted	$\{x, prev(x)\}$

Table 3. Full impact sets for lists for Sorted List Reverse

The following program reverses a sorted list as defined by the local condition above. We annotate this program with comments on the current composition of the broken set according to the rules of Table 3.

```
pre: $Br = \emptyset \ \wedge\ \varphi(x) \ \wedge\ sorted(x)$
post: $Br' = \emptyset \ \wedge\ \varphi(ret) \ \wedge\ rev\_sorted(ret) \ \wedge\ keys(ret) = old(keys(x)) \ \wedge\ hslist(ret) = old(hslist(x))$
sorted_list_reverse(x: C, Br: Set(C))
returns ret: C, Br: Set(C)
{
  LCOutsideBr(x, Br);
  var cur := x;
  ret := null;
  while (cur ≠ nil)
     invariant $cur \neq nil \ \Rightarrow\ LC(cur) \wedge sorted(cur) \wedge \varphi(cur)$
     invariant $ret \neq nil \ \Rightarrow\ LC(ret) \wedge rev\_sorted(ret) \wedge \varphi(ret)$
     invariant $cur \neq nil \ \wedge\ ret \neq nil \ \Rightarrow\ key(ret) \ \leq\ key(cur)$
```

```
    invariant old(keys(x)) = ite(cur = nil, ∅, keys(cur)) ∪ ite(ret = nil, ∅, keys(ret))
    invariant old(hslist(x)) = ite(cur = nil, ∅, hslist(cur)) ∪ ite(ret = nil, ∅, hslist(ret))
    invariant Br = ∅
    decreases ite(cur ≠ nil, 0, length(cur))
  {
    var tmp := cur.next;                 // {}
    if (tmp ≠ nil) then {
      LCOutsideBr(tmp, Br);              // {}
      Mut(tmp, p, nil, Br);              // {cur, tmp}
    }
    Mut(cur, next, ret, Br);             // {cur, tmp}
    if (ret ≠ nil) then {
      Mut(ret, p, cur, Br);              // {cur, tmp, ret}
    }
    Mut(cur, keys,
      {cur.k} ∪ (if cur.next=nil then φ else cur.next.keys), Br);    // {cur, tmp, ret}
    Mut(cur, hslist,
      {cur} ∪ (if cur.next=nil then φ else cur.next.hslist), Br);    // {cur, tmp, ret}
    if (cur.next ≠ nil ∧ (cur.key > cur.next.key ∨ ¬cur.next.sorted)) {
      Mut(cur, sorted, false, Br);       // {cur, tmp, ret}
    }
    Mut(cur, rev_sorted, true, Br);      // {cur, tmp, ret}
    AssertLCAndRemove(cur, Br);          // {tmp, ret}
    AssertLCAndRemove(ret, Br);          // {tmp}
    AssertLCAndRemove(tmp, Br);          // {}
    ret := cur;
    cur := tmp;
  }
  // The current value of ret is returned
}
```

C.4 Merging Sorted Lists

We demonstrate the ability of intrinsic definitions to handle multiple data structures at once, using the example of in-place merging of two sorted lists. The method merges the two lists by reusing the two lists' elements, which is a natural pattern for imperative code. Once again, we extend the definition of sorted lists from Case Study 4.1. We add the predicates $list1 : C \longrightarrow Bool$, $list2 : C \longrightarrow Bool$, and $list3 : C \longrightarrow Bool$, to indicate disjoint classes of lists. The relevant local condition and impact sets are:

$$\begin{aligned}
&(list1(x) \lor list2(x) \lor list3(x))\\
\land\ &\neg(list1(x) \land list2(x)) \land \neg(list2(x) \land list3(x))\\
\land\ &\neg(list1(x) \land list3(x))\\
\land\ &(list1(x) \Rightarrow (next(x) \neq nil \Rightarrow list1(next(x))))\\
\land\ &(list2(x) \Rightarrow (next(x) \neq nil \Rightarrow list2(next(x))))\\
\land\ &(list3(x) \Rightarrow (next(x) \neq nil \Rightarrow list3(next(x))))
\end{aligned}$$

Mutated Field f	Impacted Objects A_f
$list1$	$\{x, prev(x)\}$
$list2$	$\{x, prev(x)\}$
$list3$	$\{x, prev(x)\}$

Disjointness of the three lists is ensured by insisting that every object has at most one of the three list predicates hold.

We give a gist of the proof of the merge method. The recursive program compares the keys at the heads of the first and second sorted lists, and adds the appropriate node to the front of the third list. It turns out that we can easily update the ghost maps for this node (making it belong to the third list, and updating its parent pointer and key set) as well as updating the parent pointer of the head of the list where the node is removed from. When one of the lists is empty, we append the third list to the non-empty list using a single pointer mutation and then, using a ghost loop, we update the nodes in the appended list to make $list3$ true (this needs a loop invariant involving the broken set). We provide below the full local conditions and impact sets.

$$\begin{aligned}
LC \equiv \forall x.(&list1(x) \lor list2(x) \lor list3(x)) \\
&\land \neg(list1(x) \land list2(x)) \land \neg(list2(x) \land list3(x)) \\
&\land \neg(list1(x) \land list3(x)) \\
&\land (prev(x) \neq nil \Rightarrow next(prev(x)) = x) \\
&\land (next(x) \neq nil \Rightarrow prev(next(x)) = x \\
&\qquad\qquad \land length(x) = length(next(x)) + 1 \\
&\qquad\qquad \land keys(x) = keys(next(x)) \cup \{key(x)\} \\
&\qquad\qquad \land hslist(x) = hslist(next(x)) \uplus \{x\} \quad \text{(disjoint union)} \\
&\qquad\qquad \land key(x) \leq key(next(x))) \\
&\land (next(x) = nil \Rightarrow length(x) = 1 \land keys(x) = \{key(x)\} \land hslist(x) = \{x\}) \\
&\land (list1(x) \Rightarrow (next(x) \neq nil \Rightarrow list1(next(x)))) \\
&\land (list2(x) \Rightarrow (next(x) \neq nil \Rightarrow list2(next(x)))) \\
&\land (list3(x) \Rightarrow (next(x) \neq nil \Rightarrow list3(next(x))))
\end{aligned} \tag{3}$$

Fig. 9. Full local condition for lists for Sorted List Reverse

Note that we also have a variation of the local condition LC_{NC}, used in ghost loop invariants, which is similar to Equation 3, except the final three conjuncts (those enforcing closure on $list1, list2, list3$) are removed. This is done when converting an entire list from one class to another (i.e., converting from $list1$ to $list3$). The following are the full impact sets for all fields of this data structure.

Mutated Field f	Impacted Objects A_f
$next$	$\{x, old(next(x))\}$
key	$\{x, prev(x)\}$
$prev$	$\{x, old(prev(x))\}$
$length$	$\{x, prev(x)\}$
$keys$	$\{x, prev(x)\}$
$hslist$	$\{x, prev(x)\}$
$list1$	$\{x, prev(x)\}$
$list2$	$\{x, prev(x)\}$
$list3$	$\{x, prev(x)\}$

Fig. 10. Full impact sets for disjoint sorted lists