Pure Bi-Abduction for Specification Inference

Minh-Thai Trinh Quang Loc Le Cristina David Wei-Ngan Chin

Department of Computer Science, National University of Singapore

Abstract

Separation logic is an expressive formalism for specifying the properties of heap-manipulating programs. Recent work on bi-abduction [5] has shown that it can provide a compositional approach to automatically analyze data structure shapes. However, it has omitted the inference of interesting *pure properties* such as size, sum, height, content or min/max of data structures that can capture a higher level of program correctness. In this paper, we propose a novel approach, called *pure bi-abduction*, for inferring non-heap properties, using shape specifications that were obtained from a prior shape analysis. We design a set of fundamental mechanisms for pure bi-abduction to help ensure succinct and precise specifications. Our mechanisms include the inference of obligations and definitions for *uninterpreted relations*, prior to their synthesis.

To capture pure properties, we also design a *predicate extension mechanism* to allow new predicates with additional properties to be automatically derived from shape predicates. This mechanism can be applied for a broad range of recursive data structures. We also support data structures with stronger invariant properties, such as sortedness, by allowing *predicate specialization* to be utilized. We have implemented our new specification inference mechanism and have evaluated its utility on a benchmark of programs.

Keywords Specification Inference, Pure Bi-Abduction, Separation Logic, Program Verification, Functional Correctness.

1. Introduction

One of the challenging areas for software verification concerns programs with heap-based data structures [13]. In the last decade, research methodology based on separation logic [17, 23] has offered good compositional solutions to guarantee memory safety and functional correctness for such programs. Initial works in this direction [1, 16] have strived towards automated verification against user-supplied specifications. These works are typically based on entailment system of the form:

$$\Delta_1 \vdash \Delta_2 \rightsquigarrow \Delta_r$$

which attempts to prove that the current state Δ_1 entails an expected state Δ_2 with Δ_r as its frame (or residual) not required for proving Δ_2. If this entailment holds, we expect it to satisfy $\Delta_1 \Longrightarrow \Delta_2 * \Delta_r$, where $\Longrightarrow$ denotes logical implication. A key feature here is the support for local reasoning via its frame inference capability.

To support shape analysis, the notion of bi-abduction was introduced in [5] that would allow both preconditions and postconditions on shape specification to be automatically inferred. Bi-abduction is based on a more general entailment of the form:

$$\Delta_1 \vdash \Delta_2 \rightsquigarrow (\Delta_p, \Delta_r)$$

whereby a precondition Δ_p may also be inferred. If this bi-abductive entailment holds, we can assert the following logical implication: $\Delta_1 * \Delta_p \Longrightarrow \Delta_2 * \Delta_r$.

Bi-abductive entailment has succeeded in supporting shape analysis for ensuring memory safety (from null- and dangling-dereferences) [5, 10], by helping to infer pre/post specifications that utilize shape predicates, such as `ll` below for acyclic linked-list.

```
data node {int val; node next; }
pred ll⟨root⟩≡root=null ∨ ∃ q·(root↦node⟨_, q⟩ * ll⟨q⟩)
```

The first declaration is for a node with two fields, namely a data field `val` and a pointer field `next` for pointing to the next node. The second declaration is for a heap-based predicate, named `ll`. Note that `root↦node⟨_, q⟩` denotes a heap node pointed by `root`. A key operator present in separation logic, called *spatial conjunction*, is denoted by $*$ as seen in `root↦node⟨_, q⟩*ll⟨q⟩`. This operator states that data node, `root↦node⟨_, q⟩`, and heap predicate for tail of the linked-list, `ll⟨q⟩`, reside in disjoint memory sections.

However, bi-abduction in [5, 10] presently suffers from an inability to analyze for pure properties needed to express a higher-level of program correctness. As an example, it is currently unable to infer pre/post specifications for predicates with pure properties, such as `llMM` below that also captures a minimum value and a maximum value (as pure properties) of its linked-list.

```
pred llMM⟨root, mi, mx⟩ ≡ (root↦node⟨mi, null⟩∧mi=mx)
  ∨ ∃ v, q, mi2, mx2 · (root↦node⟨v, q⟩* llMM⟨q, mi2, mx2⟩
    ∧ mi=min(v, mi2) ∧ mx=max(v, mx2))
```

In particular, the capture of min/max values can be used to guide our reasoning on sortedness property, which is prerequisite in order to capture the correctness of programs such as `bubble sort`.

In this paper, we propose a systematic methodology, called *pure bi-abduction*, for inferring pure properties in the separation logic domain. Our proposal is built on the assumption that heap-related properties in separation logic form have already been inferred in a prior step. To better exploit the expressiveness of separation logic, we integrate inference mechanisms for pure properties directly into it and propose to use an entailment system of the following form:

$$[v_1, .., v_n]\ \Delta_1 \vdash \Delta_2 \rightsquigarrow (\phi_p, \Delta_r, \beta_c)$$

Three new features are added here to support inference on pure properties. First, we may specify a set of variables $\{v_1, .., v_n\}$ for which inference is to be selectively applied. As a special case,

when no variables are specified, the entailment system reduces to forward verification without inference capability. Second, we allow *second-order variables*, in the form of *uninterpreted* relations, to support inference of pure properties for pre/post specifications. Lastly, our version of bi-abduction would collect a set of constraints β_c of the form $\phi_1 \Longrightarrow \phi_2$, to provide interpretations for second-order variables. This approach is critical for capturing inductive definitions that can be refined via fix-point analysis.

The main novelty of our current work is a systematic inference of pure properties for separation logic, a process we refer to as *pure bi-abduction*. While the bi-abductive entailment for pure properties remains a centerpiece of our proposal, we have also devised a set of *derivation techniques* that can systematically transform predicates in order to incorporate new pure properties. These techniques are crucial for automatically enhancing heap predicates (obtained from a prior shape analysis) with relevant pure properties. Our specific technical contributions include the following:

- We design a new bi-abductive entailment procedure for inferring pure properties in heap-based programs. The power of our procedure is significantly enhanced by its collection of pure proof obligations over uninterpreted relations (Secs 3, 5, 6).
- We propose a methodology for automatically deriving new inductive predicate definitions. Our approach supports (i) *predicate extension* for automatically enhancing heap predicates with a variety of pure properties, and (ii) *predicate specialization* to derive predicates with stronger invariant properties (Secs 4, 8). We then show how *high-level user guidance* can be applied to allow pure properties to be systematically inferred (Sec 9).
- We have implemented our approach and evaluated it on a benchmark of programs (Sec 10). We show that pure properties are prerequisite to allow the correctness of more than 20% of analyzed procedures to be captured and verified.

A recent related work [22] focused on refining partial specifications, using a semi-automatic approach whereby predicate definitions are to be manually provided. This work did not take advantage of prior shape analysis, nor did it focus on the fundamental mechanisms for pure bi-abduction. The current paper addresses these issues by designing a new pure bi-abduction entailment procedure, together with the handling of unknown functions and relations. To work with shape analysis, we also propose a set of high-level commands and derivation techniques for systematically enhancing predicates with new pure properties and specializations.

2. Overview and Motivation

We highlight our approach for pure bi-abduction with a simple example that combines two lists of integers into a single list of integers.

```
1  node zip(node x, node y) {
2    if (x == null)
3      return null;
4    else {
5      x.val = x.val + y.val;
6      x.next = zip(x.next,y.next);
7      return x;
8    }
9  }
```

By utilizing shape analysis techniques from [5, 9], we could obtain the following shape specification for `zip`.

$$\texttt{requires ll}\langle\texttt{x}\rangle{*}\texttt{ll}\langle\texttt{y}\rangle$$
$$\texttt{ensures ll}\langle\texttt{res}\rangle;$$

However, this specification is not memory-safe, as it triggers a null-dereferencing error when the x-list is longer than y-list. To rectify this problem, our approach would initially derive a new predicate with length property from the above acyclic linked-list `ll` predicate, as follows:

$$\texttt{pred llN}\langle\texttt{root, n}\rangle \equiv (\texttt{root=null} \wedge \texttt{n=0}) \vee \exists\, \texttt{q} \cdot (\texttt{root}{\mapsto}\texttt{node}\langle \_, \texttt{q}\rangle{*}\ \texttt{llN}\langle\texttt{q, n}{-}1\rangle)$$

With this new predicate, we could enhance pre/post specification for `zip` to include $\texttt{P(a,b)}$ as a pure pre-relation and $\texttt{Q(r,a,b)}$ as a pure post-relation in the specification. We choose $\texttt{a,b,r}$ as parameters to these relations, since they denote pure size properties which we wish to analyze. Pre-relation should be as weak as possible, while post-relation should be as strong as possible.

$$\texttt{infer}\,[\texttt{P,Q}]$$
$$\texttt{requires llN}\langle\texttt{x,a}\rangle{*}\texttt{llN}\langle\texttt{y,b}\rangle \wedge \texttt{P(a,b)}$$
$$\texttt{ensures llN}\langle\texttt{res,r}\rangle \wedge \texttt{Q(r,a,b)};$$

Intuitively, the above syntax is meant to express a pair of pre/post-condition (`requires`/`ensures`) along with inference capability (`infer`) that is to be applied on unknown variables `P`, `Q`.

Using forward reasoning on the `zip` code, our pure bi-abductive entailment procedure would gather the following proof obligations on the two unknown relations:

$$\texttt{a=ar}{+}1 \wedge \texttt{b=br}{+}1 \wedge 0{\leq}\texttt{ar} \wedge 0{\leq}\texttt{br} \wedge \texttt{P(a,b)} \Longrightarrow \texttt{P(ar,br)},$$
$$\texttt{P(a,b)} \Longrightarrow \texttt{b}{\neq}0 \vee \texttt{a}{\leq}0,$$
$$\texttt{r=0} \wedge \texttt{a=0} \wedge 0{\leq}\texttt{b} \wedge \texttt{P(a,b)} \Longrightarrow \texttt{Q(r,a,b)},$$
$$\texttt{rn=r}{-}1 \wedge \texttt{bn=b}{-}1 \wedge \texttt{an=a}{-}1 \wedge 1{\leq}\texttt{b,a,r} \wedge \texttt{P(a,b)} \wedge \texttt{Q(rn,an,bn)} \Longrightarrow \texttt{Q(r,a,b)}.$$

Our approach would strive to infer the strongest possible post-relation `Q`, and the weakest possible pre-relation `P`, from the above proof obligations. Using suitable fix-point analysis techniques, we can synthesize the following approximations which would add a pure pre-condition $\texttt{b}{\geq}\texttt{a}$ that guarantees memory safety.

$$\texttt{P(a,b)} \equiv \texttt{b}{\geq}\texttt{a}$$
$$\texttt{Q(r,a,b)} \equiv \texttt{r=a}$$

In short, we will obtain the new specification which also captures the size property:

$$\texttt{requires llN}\langle\texttt{x,a}\rangle{*}\texttt{llN}\langle\texttt{y,b}\rangle \wedge \texttt{a}{\leq}\texttt{b}$$
$$\texttt{ensures llN}\langle\texttt{res,r}\rangle \wedge \texttt{r=a};$$

With pure inference, we can go beyond memory safety towards functional correctness and also total correctness. Total correctness requires programs to be proven to terminate.

Program termination is typically proven with a well-founded decreasing measure. Our inference mechanism can help discover suitable well-founded ranking functions [19] to support termination proofs. For this task, we would introduce an uninterpreted function `F`, as a possible measure, via the following termination-based specification that is synthesized right after size inference.

$$\texttt{infer}\,[\texttt{F}]$$
$$\texttt{requires llN}\langle\texttt{x,a}\rangle{*}\texttt{llN}\langle\texttt{y,b}\rangle \wedge \texttt{a}{\leq}\texttt{b}{\wedge}\texttt{Term}[\texttt{F(a,b)}]$$
$$\texttt{ensures llN}\langle\texttt{res,r}\rangle \wedge \texttt{r=a};$$

Applying our pure bi-abduction procedure, we can derive the following proof obligations whose satisfaction would guarantee total correctness.

$$\texttt{a}{\geq}0{\wedge}\texttt{b}{\geq}0{\wedge}\texttt{a}{\leq}\texttt{b} \Longrightarrow \texttt{F(a)}{\geq}0$$
$$\texttt{an=a}{-}1{\wedge}\texttt{bn=b}{-}1{\wedge}\texttt{a}{\leq}\texttt{b}{\wedge}\texttt{an}{\geq}0 \Longrightarrow \texttt{F(a,b)}{>}\texttt{F(an,bn)}$$

Using suitable fixpoint analysis techniques, we can synthesize $\texttt{F(a)} \equiv \texttt{a}{-}1$; thus capturing a well-founded decreasing measure for our method. Though termination analysis of programs have been extensively investigated before, we find it refreshing to reconsider it in the context of pure property inference for pre/post

specifications. (For space reasons, we shall not consider this aspect that uses uninterpreted functions in the rest of the paper.)

Thus, pure properties can considerably enrich our specifications, and it is useful to have a systematic way to infer such properties for heap-based programs. Our next section describes key principles used by our fundamental inference mechanism.

3. Principles of Pure Bi-Abduction

We shall first highlight key principles employed by pure bi-abduction with examples. Later in Sec. 5, we shall present formalizations of our proposed techniques.

3.1 Selective Inference

Our first principle is based on the notion that pure bi-abduction is best done selectively. Consider two entailments below with $\mathtt{x}{\mapsto}\mathtt{node}\langle \_, \mathtt{q}\rangle$ as a consequent:

$$[\mathtt{n}]\ \mathtt{lln}\langle \mathtt{x}, \mathtt{n}\rangle \vdash \mathtt{x}{\mapsto}\mathtt{node}\langle \_, \mathtt{q}\rangle \rightsquigarrow (\mathtt{n}{>}0, \mathtt{lln}\langle \mathtt{q}, \mathtt{n}{-}1\rangle, \emptyset)$$

$$[\mathtt{x}]\ \mathtt{lln}\langle \mathtt{x}, \mathtt{n}\rangle \vdash \mathtt{x}{\mapsto}\mathtt{node}\langle \_, \mathtt{q}\rangle \rightsquigarrow (\mathtt{x}{\neq}\mathtt{null}, \mathtt{lln}\langle \mathtt{q}, \mathtt{n}{-}1\rangle, \emptyset)$$

$$[\mathtt{n}, \mathtt{x}]\ \mathtt{lln}\langle \mathtt{x}, \mathtt{n}\rangle \vdash \mathtt{x}{\mapsto}\mathtt{node}\langle \_, \mathtt{q}\rangle \rightsquigarrow (\mathtt{n}{>}0 \vee \mathtt{x}{\neq}\mathtt{null}, \mathtt{lln}\langle \mathtt{q}, \mathtt{n}{-}1\rangle, \emptyset)$$

Predicate $\mathtt{lln}\langle \mathtt{x}, \mathtt{n}\rangle$ by itself does not entail a non-empty node. For entailment to succeed, the current state would have to be strengthened with either $\mathtt{x}{\neq}\mathtt{null}$ or $\mathtt{n}{>}0$. Our procedure can decide on which pre-condition to return, depending on the set of variables for which pre-conditions are to be built from. This selectivity is also helpful when we want to infer specifications incrementally (as shown in Sec. 9). Since in each incremental step, we may only need to consider a subset of variables.

3.2 Succinct and Weaker Preconditions

Consider $[\mathtt{s}_1, \mathtt{b}_1, \mathtt{s}_2, \mathtt{b}_2]\ \mathtt{s}_1{\leq}\mathtt{b}_1 \vdash \mathtt{s}_1{\leq}\mathtt{s}_2$. A weakest precondition would have been the disjunction $\neg(\mathtt{s}_1{\leq}\mathtt{b}_1){\vee}(\mathtt{s}_1{\leq}\mathtt{s}_2)$, while $\mathtt{s}_1{\leq}\mathtt{s}_2$ would be stronger than necessary since it did not take advantage of the antecedent $\mathtt{s}_1{\leq}\mathtt{s}_2$. To avoid creating large disjunctions, we heuristically choose a more succinct precondition, namely $\mathtt{b}_1{\leq}\mathtt{s}_2$.

3.3 Never Inferring `false`

Another principle that we strive in our selective inference is that we never knowingly infer any cumulative precondition that is equivalent to `false`, since such a precondition would not be provable for any satisfiable program state. As an example, consider $[\mathtt{x}]\ \mathtt{true} \vdash \mathtt{x}{>}\mathtt{x}$. Though we could have inferred $\mathtt{x}{>}\mathtt{x}$, we refrain from doing so, since it is only provable under dead code scenarios.

3.4 Antecedent Contradiction

Consider $[v^*]\ \Delta_1 \vdash \Delta_2$. If a contradiction is detected between Δ_1 and Δ_2, the only precondition (over variables v^*) that would allow such an entailment to succeed is one that contradicts with the antecedent Δ_1. We allow such precondition, which is not equivalent to false, to be inferred. For example, consider $[\mathtt{b}]\ \mathtt{x}{=}1{\wedge}\mathtt{b}{>}0 \vdash \mathtt{x}{=}2$. We have a contradiction between $\mathtt{x}{=}1{\wedge}\mathtt{b}{>}0$ and $\mathtt{x}{=}2$. To allow this entailment to succeed, we selectively infer $\mathtt{b}{\leq}0$ as its precondition over just the selected variable $[\mathtt{b}]$. Though precondition $\mathtt{b}{\leq}0$ forces a contradiction in the antecedent, it is by itself not equivalent to `false`, and this does not conflict with our previous principle.

3.5 Uninterpreted Relations

Our inference deals with unknown relations that may appear in either preconditions or postconditions. We refer to the former as *pre-relations* and the latter as *post-relations*. Unknown pre-relations are expected to be as weak as possible, while unknown post-relations should be as strong as possible. Our inference mechanism respect this principle, and would use it to derive weaker pre-relations and stronger post-relations.

To provide definitions for these unknown relations (such as $R(v^*)$), we infer two kinds of relational constraints. The first kind, called *relational obligation*, is of the form $\Delta{\wedge}R(v^*){\rightarrow}c$, where the consequent $\mathtt{c}$ is a *known* constraint and *unknown* $R(v^*)$ is present in the antecedent. The second kind, called *relational definition*, is of the form $\Delta \rightarrow R(v^*)$, where the unknown relation is present in the consequent instead.

Relational obligations

These are useful in two ways. For unknown pre-relations, they act as initial precondition that can be utilized by fix-point analysis. For unknown post-relation, they denote proof obligations which the post-relation must satisfy. We could check this after we have synthesized the unknown post-relation.

As an example, consider the following entailment:

$$[\mathtt{P}]\ \mathtt{s}_1{\leq}\mathtt{b}_1{\wedge}\mathtt{P}(\mathtt{s}_1, \mathtt{b}_1, \mathtt{s}_2, \mathtt{b}_2) \vdash \mathtt{s}_1{\leq}\mathtt{s}_2$$

We infer $\mathtt{P}(\mathtt{s}_1, \mathtt{b}_1, \mathtt{s}_2, \mathtt{b}_2){\rightarrow}\mathtt{b}_1{\leq}\mathtt{s}_2$ which will denote a proof obligation for `P`. More generally, consider $[P]\ \Delta{\wedge}P(v^*) \vdash \alpha$ where α denotes a *known* constraint, we will first selectively infer pre-condition ϕ over selected variables v^* and then collect $P(v^*){\rightarrow}\phi$ as our relational obligation. To obtain succinct pre-conditions, we filter out constraints that contradicts with the current program state.

Relational definitions

These are typically used to form definitions for fixpoint analysis. For unknown post-relations, we should infer the strongest possible definitions for them, where possible. After gathering the relational definitions (both base and inductive cases), we would apply a least fixpoint procedure [20] to discover suitable closed-form definitions for post-relations. For unknown pre-relations, while it may be possible to compute *greatest fix point* to discover the weakest pre-relations that can satisfy all relational constraint, we have designed two simpler techniques for inferring unknown pre-relations. Firstly, we attempt to extract conditions on input variables that were already inferred for the post-relation. If these can satisfy all relational constraints for pre-relations, we simply use them as the approximations of our pre-relations. If not, we proceed with a second technique to construct a recursive definition that expresses the relation between an arbitrary recursive call and the initial method call. After another least fix-point analysis, we can determine a pre-condition on the initial method call which would allow proof obligations for all recursive calls to be satisfied. This can be combined with proof obligation for the initial call to ensure that it is always satisfied for all method invocations. This approach allows us to avoid greatest fix-point analysis techniques, and is sufficient for all practical examples that we have evaluated.

4. Enhancing Predicates with Pure Properties

To exploit prior results from shape analyses, we propose a set of predicate derivation techniques that can enhance heap predicates with new pure properties. Some examples of heap predicates that can be inferred from shape analyses include linked-list segments, doubly-linked lists and trees, as illustrated below.

```
data node2 {int val; node2 left; node2 right; }
pred lseg⟨root, p⟩ ≡ root=p ∨
       root↦node⟨_, q⟩*lseg⟨q, p⟩
pred dll⟨root, p⟩ ≡ root=null ∨
       root↦node2⟨_, p, q⟩*dll⟨q, root⟩
pred tree⟨root⟩ ≡ root=null ∨
       root↦node2⟨_, q, r⟩*tree⟨q⟩*tree⟨r⟩
```

These heap predicates are concerned with only data nodes and their pointer links, and are devoid of pure properties. We propose to use a predicate extension mechanism to add pure properties to heap predicates. Our mechanism is generic in that each specified property can be applied to a broad range recursive data structures, such as linked-lists and trees. Some examples of pure properties that can be specified with our predicate extension mechanism are shown next. They cover the size, height, sum, head, min/max and set of values/addresses properties for heap-based data structures.

```
pred_extn size[@R]⟨n⟩ ≡ n=0 ∨ size⟨R,m⟩∧n=1+m.
pred_extn height[@R]⟨n⟩ ≡ n=0 ∨
        height⟨R,m⟩∧n=1+max(m).
pred_extn sum[@V,@R]⟨s⟩ ≡ s=0 ∨ sum⟨R,r⟩∧s=V+r.
pred_extn head[@V]⟨v⟩ ≡ v=V.
pred_extn minP[@V,@R]⟨mi⟩ ≡ mi=min(V) ∨
        minP⟨R,mi2⟩∧mi=min(V,mi2).
pred_extn maxP[@V,@R]⟨mx⟩ ≡ mx=max(V) ∨
        maxP⟨R,mx2⟩∧mx=max(V,mx2).
pred_extn set[@V,@R]⟨S⟩ ≡ S={} ∨set⟨R,S2⟩∧S={V}⊔S2.
pred_extn setAddr[@R]⟨S⟩ ≡ S={} ∨
        setAddr⟨R,S2⟩∧S={root}⊔S2.
```

These properties are *pure extensions* in that they do not restrict the form of non-null heap structures. However, some of these properties, such as `min`, `max` and `head`, may enforce the non-null requirement. Note that `@V`, `@R` denote some annotations on data fields that are needed to compute the specified pure property, whereas `V`, `R` denote the corresponding data values themselves. Each occurrence of the annotated value, e.g. `V`, is either exactly a single occurrence or it denotes multiple occurrences, depending on its context of use. For example, the `V` value in `head` property denotes a single occurrence since its constraint `v=V` implicitly requires exactly one occurrence. To denote multiple occurrences, the constraint would have to be written as `v=f(V)` where `f` is an associative operator, such as `min`, `max`, as was used in the `max` property definition. The presence of an associative operator would allow us to combine multiple occurrences into a single result. For values that denote recursive pointers, such as `R` in the `set` extension, the occurrences of their values are determined through constraints on their defining properties, such as S_2 from `set`$\langle R, S_2\rangle$. Here, S_2 denotes multiple occurrences since we can identify an associative $\sqcup$ operator from $S=\{V\}\sqcup S_2$ for combining them. To illustrate how property extensions are used, we specify two predicate derivations with them, as shown below:

```
data node {int val@VAL; node next@REC; }
data node2 {int val@VAL; node2 left@REC;
        node2 right@REC; }
pred treeH⟨root,h⟩=derive tree⟨root⟩ with
    height[@REC]⟨h⟩.
pred llhMM⟨root,v,mi⟩=derive ll⟨root⟩ with
    head[@Val]⟨v⟩,minP[@Val,@Rec]⟨mi⟩.
```

Note how we use annotations (such as `VAL`, `REC`[1]) to mark fields of our underlying heap structures that are common across different data types. For non-null properties, such as `head`, we provide a separate technique to derive non-null predicate definitions prior to their use by these non-null properties. In the case of `llhMM` example, we would be using the following non-null predicate definition instead.

```
pred ll⟨root⟩ ≡ root↦node⟨_,null⟩ ∨
    root↦node⟨_,q⟩ * ll⟨q⟩
```

[1] In the case of binary data nodes, we may add `REC1`, `REC2` annotations to better distinguish its two recursive pointers. This distinction is required for expressing in-order sortedness for binary trees.

With the above two commands, our predicate derivation technique would synthesize the following predicates

```
pred treeH⟨root,h⟩ ≡ root=null∧h=0
  ∨ root↦node2⟨_,p,q⟩*treeH⟨p,h1⟩*treeH⟨q,h2⟩
    ∧ h=1+max(h1,h2).
pred llhMM⟨root,v,mi⟩ ≡
  root↦node⟨v,null⟩∧mi=v
  ∨ root↦node⟨v,q⟩*llhMM⟨q,_,mi2⟩∧ mi=min(v,mi2).
```

While property extensions are user-definable, their use within our pure inference sub-system can be completely automated, as we can automatically construct predicate derivation commands and systematically apply them after shape analysis. We replace each heap predicate with its derived counterpart, followed by the introduction of uninterpreted pre- and post-relations before applying Hoare-style verification with pure bi-abductive inference.

We may also allow stronger properties to be added to heap predicates, whereby some invariant property hold recursively throughout the data structure. As such invariants restrict the allowable heap structures, we refer to this mechanism as *predicate specialization*. Two examples are highlighted below.

```
pred_spec AVL[@R]⟨h⟩ ≡ spec height[@R]⟨h⟩ with
    height⟨R,h1⟩*height⟨R,h2⟩∧−1≤h1−h2≤1.
pred_spec SortA[@V,@R]⟨mi⟩ ≡ spec minP[@V,@R]⟨mi⟩
    with minP⟨R,mi2⟩∧V≤min(mi2).
```

Specialization `AVL` is applicable to binary tree structures to ensure that its two subtrees are nearly balanced through an extra constraint $1\leq h_1-h_2\leq 1$ on each recursive node. Specialization `SortA` ensures that its given data structure is sorted in ascending order (from root to leaves) through the constraint $V\leq min(mi_2)$. Two examples of their use in deriving stronger predicates are shown below.

```
pred treeAVL⟨root,h⟩= derive treeH⟨root,h⟩
    with AVL[@Rec]⟨h⟩.
pred llA⟨root,v,mi⟩= derive llhMM⟨root,v,mi⟩
    with SortA[@Val,@Rec]⟨mi⟩.
```

The above two commands could be automatically generated to produce the following specialized predicates with deep invariant properties that are applicable on each of its recursive nodes.

```
pred treeAVL⟨root,h⟩ ≡ root=nil∧h=0
    ∨ root↦node⟨_,p,q⟩*treeAVL⟨p,h1⟩*treeAVL⟨q,h2⟩
      ∧ h=1+max(h1,h2)∧−1≤h1−h2≤1.
pred llA⟨root,v,mi⟩ ≡ root↦node⟨v,null⟩ ∧mi=v
    ∨ root↦node⟨v,q⟩*llA⟨q,_,mi2⟩
      ∧ mi=min(v,mi2)∧v≤mi2.
```

As a summary, we stress that this newly proposed predicate extension and specialization mechanisms are to support high-level user guidance for pure property inference. Once they have been designed and instructed by users, our pure bi-abduction entailment procedure can systematically use them to derive stronger specifications with greater correctness. Our formalization on predicate derivation for property extension and specialization is outlined later in Sec 8, while high-level user guidance commands to aid specification inference will be described in Sec 9.

5. Formalization of Pure Bi-Abduction

Before formally describing the entailment procedure, we introduce the specification language in Figure 1. The language supports data type declarations via *datat*, shape predicate definitions via *spred*, and method specifications via `S`. Each iterative loop is converted to an equivalent tail-recursive method, where mutations on parameters are made visible to the caller via pass-by-reference.

Program	*Prog*	$::= \mathit{tdecl}^*\ \mathit{meth}^*$
	tdecl	$::= \mathit{datat} \mid \mathit{spred} \mid \mathtt{S}$
Data decl.	*datat*	$::= \mathtt{data}\ c\ \{\ (t\ v)^*\ \}$
Shape pred.	*spred*	$::= p\langle v^*\rangle \equiv \Phi$
Method spec.	S	$::= \mathtt{infer}\ [\ v^*, v^*_{rel}\ ]\ \mathtt{requires}\ \Phi_{pr}\ \mathtt{ensures}\ \Phi_{po};$
Formula	Φ	$::= \bigvee(\exists v^* \cdot \kappa \wedge \pi)^*$
Heap formula	κ	$::= \kappa_1 * \kappa_2 \mid p\langle v^*\rangle \mid v \mapsto c\langle v^*\rangle \mid \mathtt{emp}$
Pure formula	π	$::= \pi \wedge \iota \mid \iota \qquad \iota ::= v_{rel}(v^*) \mid \alpha$
	α	$::= \gamma \mid i \mid b \mid \varphi \mid \alpha_1 \vee \alpha_2 \mid \alpha_1 \wedge \alpha_2 \mid \neg\alpha \mid \exists v \cdot \alpha \mid \forall v \cdot \alpha$
Linear arith.	i	$::= a_1 = a_2 \mid a_1 \leq a_2$
	a	$::= k^{\mathtt{int}} \mid v \mid k^{\mathtt{int}} \times a \mid a_1 + a_2 \mid -a \mid \mathit{max}(a_1,a_2) \mid \mathit{min}(a_1,a_2)$
Boolean formula	b	$::= \mathtt{true} \mid \mathtt{false} \mid v \mid b_1 = b_2$
Bag constraint	φ	$::= v \in \mathtt{B} \mid \mathtt{B}_1 = \mathtt{B}_2 \mid \mathtt{B}_1 \sqsubset \mathtt{B}_2 \mid \forall v \in \mathtt{B} \cdot \alpha \mid \exists v \in \mathtt{B} \cdot \alpha$
	B	$::= \mathtt{B}_1 \sqcup \mathtt{B}_2 \mid \mathtt{B}_1 \sqcap \mathtt{B}_2 \mid \mathtt{B}_1 - \mathtt{B}_2 \mid \{\} \mid \{v\}$
Ptr. (dis)equality	γ	$::= v_1 = v_2 \mid v = \mathtt{null} \mid v_1 \neq v_2 \mid v \neq \mathtt{null}$
	β	$::= v_{rel}(v^*) \rightarrow \pi \mid \pi \rightarrow v_{rel}(v^*)$
	Δ	$::= \Delta_1 \vee \Delta_2 \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta \mid \kappa \wedge \pi$
	ϕ	$::= \pi$

Figure 1: The Specification Language used in Pure Bi-Abduction

Regarding each method's specification, as shown in Fig 1, it is made up of a set of inferable variables $[v^*, v^*_{rel}]$, a precondition Φ_{pr} and a postcondition Φ_{po}. The intended meaning is that whenever the method is called in a state satisfying precondition Φ_{pr} and if the method terminates, the resulting state will satisfy the corresponding postcondition Φ_{po}. The user can enable the specification inference process by providing a specification with inferable variables. If $[v^*]$ is specified, suitable preconditions on these variables will be inferred. And if $[v^*_{rel}]$ is specified, suitable approximations for these unknown relations will be inferred.

The Φ constraint is in disjunctive normal form. Each disjunct consists of a heap constraint κ, referred to as *heap part*, and a heap-independent formula π, referred to as *pure part*. The pure part does not contain any heap nodes and is presently restricted to unknown relations $v_{rel}(v^*)$, pointer equality/disequality γ ($v_1 \neq v_2$ and $v \neq \mathtt{null}$ are just short forms for $\neg(v_1 = v_2)$ and $\neg(v = \mathtt{null})$ resp.), linear arithmetic *i*, boolean constraints *b* and bag constraints φ. Internally, each unknown relation is annotated with @pr or @po, depending on whether it comes from the precondition or the post-condition, respectively. This information will be later used for inferring the formula corresponding to the unknown relation. Furthermore, Δ denotes a composite formula that can be normalized into the Φ form, whereas ϕ represents a pure formula. Note that whenever Δ and ϕ consist of only pure formulas, $*$ is equivalent to $\wedge$. The relational definitions and obligations mentioned in Section 3 are denoted as $\pi \rightarrow v_{rel}(v^*)$ and $v_{rel}(v^*) \rightarrow \pi$, respectively.

Recall that our entailment procedure for pure property has the form: $[v^*]\ \Delta_1 \vdash \Delta_2 \rightsquigarrow (\phi_3, \Delta_3, \beta_3)$. This new entailment procedure serves two key roles:

i. From the verification point of view, the goal of the entailment procedure is to reduce the entailment between separation formulas to entailment between pure formulas by successively matching up aliased heap nodes between the antecedent and the consequent through folding, unfolding and matching [16]. When this happens, the heap formula in the antecedent is soundly approximated by returning a pure approximation of the form $\bigvee(\exists v^* \cdot \pi)^*$ from each given heap formula κ.

ii. From the inference point of view, the goal is inferring a precondition ϕ_3, as well as gathering the set of conditions β_3 for the unknown functions and relations. Together with the frame inferred Δ_3, we should be able to construct relevant preconditions and postconditions for each of our methods.

$$\frac{[\textsc{inf}\text{-}[\textsc{and}]]}{[v^*]\ \pi_1 \vdash \pi_2 \rightsquigarrow (\phi_2, \Delta_2, \beta_2) \qquad [v^*]\ \pi_1 \vdash \pi_3 \rightsquigarrow (\phi_3, \Delta_3, \beta_3)}{[v^*]\ \pi_1 \vdash \pi_2 \wedge \pi_3 \rightsquigarrow (\phi_2 \wedge \phi_3, \Delta_2 * \Delta_3, \beta_2 \cup \beta_3)}$$

$$\frac{[\textsc{inf}\text{-}[\textsc{unsat}]]}{\textsc{unsat}(\pi)}\ \frac{}{[v^*]\ \pi \vdash \alpha \rightsquigarrow (\mathtt{true}, \mathtt{false}, \emptyset)} \qquad \frac{[\textsc{inf}\text{-}[\textsc{valid}]]}{\pi \Rightarrow \alpha}\ \frac{}{[v^*]\ \pi \vdash \alpha \rightsquigarrow (\mathtt{true}, \pi, \emptyset)}$$

$$\frac{[\textsc{inf}\text{-}[\textsc{lhs}\text{-}\textsc{contra}]]}{\phi = \forall(FV(\pi) - v^*) \cdot \neg\pi \quad \textsc{unsat}(\pi \wedge \alpha) \quad \phi \neq \mathtt{false}}\ \frac{}{[v^*]\ \pi \vdash \alpha \rightsquigarrow (\phi, \mathtt{false}, \emptyset)} \qquad \frac{[\textsc{inf}\text{-}[\textsc{pre}\text{-}\textsc{derive}]]}{\phi = \forall(FV(\pi, \alpha) - v^*) \cdot (\neg\pi \vee \alpha) \quad \phi \neq \mathtt{false}}\ \frac{}{[v^*]\ \pi \vdash \alpha \rightsquigarrow (\phi, \pi \wedge \phi, \emptyset)}$$

$$\frac{[\textsc{inf}\text{-}[\textsc{rel}\text{-}\textsc{defn}]]}{[v^*, v_{rel}]\ \pi \vdash v_{rel}(u^*) \rightsquigarrow (\mathtt{true}, \mathtt{true}, \{\pi \rightarrow v_{rel}(u^*)\})}$$

$$\frac{[\textsc{inf}\text{-}[\textsc{rel}\text{-}\textsc{oblg}]]}{[u^*]\ \pi \vdash \alpha \rightsquigarrow (\phi, \Delta, \emptyset)}\ \frac{}{[v^*, v_{rel}]\ \pi \wedge v_{rel}(u^*) \vdash \alpha \rightsquigarrow (\mathtt{true}, \Delta, \{v_{rel}(u^*) \rightarrow \phi\})}$$

Figure 2: Pure Bi-Abduction Rules

The focus of the current work is on the second category (ii). From this perspective, the scenario of interests is that when both the antecedent and the consequent are heap free, when the rules in Figure 2 apply. Take note that these rules are to be applied in a *top-down* and *left-to-right* order.

- Rule [INF-[AND]] repeatedly breaks the conjunctive consequent into smaller components.
- Rules [INF-[UNSAT]] and [INF-[VALID]] infer the true precondition whenever the entailment is already satisfied. More specifically, rule [INF-[UNSAT]] applies when the antecedent π of the entailment is unsatisfiable, whereas rule [INF-[VALID]] is used if [INF-[UNSAT]] cannot be applied, meaning that the antecedent is satisfiable.
- The pure precondition inference is captured by two rules [INF-[LHS-CONTRA]] and [INF-[PRE-DERIVE]]. While the first rule handles antecedent contradiction, the second one infers the missing information from the antecedent required for proving the consequent. More specifically, whenever a contradiction is detected between the antecedent π and the consequent

α, then rule [INF−[LHS−CONTRA]] applies and the precondition $\forall(FV(\pi)-v^*)\cdot\neg\pi$ contradicting with the antecedent is being inferred. Note that $FV(\cdots)$ returns the set of free variables from its argument(s), while v^* is a shorthand notation for $v_1,..,v_n$[2]. On the other hand, if no contradiction is detected, then rule [INF−[PRE−DERIVE]] infers a sufficient precondition required for proving the consequent. In order to not contradict the principle stated in Sec 3.3, both aforementioned rules check that the inferred precondition is not `false`.

- The last two rules [INF−[REL−DEFN]] and [INF−[REL−OBLG]] are meant to gather definitions and obligations, respectively, for the unknown relation $v_{rel}(u^*)$.

6. Inference via Hoare-Style Rules

Code verification is typically formulated as a Hoare triple of the form $\vdash\{\Delta_1\}\ c\ \{\Delta_2\}$, with a precondition Δ_1 and a postcondition Δ_2. This verification could either be conducted *forwards* (given Δ_1 and c, what can we calculate for Δ_2) or *backwards* (given Δ_2 and c, what verification condition can we calculate for Δ_1) for the specified properties to be successfully verified, in accordance with the rules of Hoare logic. In separation logic, the predominant mode of verification is forward. Thus, given an initial state Δ_1 and a program code c, such a Hoare-style verification rule is expected to compute a best possible postcondition Δ_2 that satisfies the inference rules of Hoare logic. If the best possible postcondition cannot be calculated, it is always sound and often sufficient to compute a suitable approximation. To support pure bi-abduction, we extend this Hoare-style forward rule to the following form:

$$[v^*]\vdash\{\Delta_1\}\ c\ \{\phi_2,\Delta_2,\beta_2\}$$

with the following additional features (i) a set of specified variables $[v^*]$ (ii) an extra precondition ϕ_2 that must be added (iii) a set of definitions and obligations β_2 on the unknown relations. The selectivity criterion will help ensure that ϕ_2 and β_2 come from only the specified set of variables, namely $\{v^*\}$. In case this set of specified variables is empty, our new rule is simply a special case that only performs verification, without any inference.

Figure 3 captures a set of our Hoare rules with pure bi-abduction. Rule [INF−[SEQ]] shows how sequential composition $e_1;e_2$ is being handled. The two inferred preconditions are conjunctively combined as $\phi_2\wedge\phi_3$. Rule [INF−[IF]] shows how conditional expression is being handled. Our core language allows only variables (e.g. w) in each conditional test. We use a primed notation whereby w denotes the old value, while w' denotes the latest value of each variable w. The conditions w' and $\neg w'$ are asserted for each of the two conditional branches, respectively. As only one of the two branches will be executed, the inferred preconditions ϕ_2,ϕ_3 are combined conjunctively in a conservative manner.

Rule [INF−[ASSIGN]] deals with assignment statement. We first define a *composition with update* operator. Given a state Δ_1, a state change Δ_2, and a set of variables to be updated $X=\{x_1,\ldots,x_n\}$, the composition operator $\mathtt{op}_X$ is defined as: $\Delta_1\ \mathtt{op}_X\ \Delta_2\stackrel{def}{=}\exists\ r_1..r_n\cdot(\rho_1\Delta_1)\ \mathtt{op}\ (\rho_2\Delta_2)$, where $r_1,\ldots,r_n$ are fresh variables and $\rho_1=[r_i/x_i']_{i=1}^n$, $\rho_2=[r_i/x_i]_{i=1}^n$. Note that ρ_1 and ρ_2 are substitutions that link each latest value of x_i' in Δ_1 with the corresponding initial value x_i in Δ_2 via a fresh variable r_i. The binary operator $\mathtt{op}$ is either $\wedge$ or $*$. Instances of this operator will be used in the inference rule for assignment (as $\wedge_u$ in [INF−[ASSIGN]]) and in the inference rule for method invocation (as $*_{V\cup W}$ in [INF−[CALL]]).

For each method call, we must ensure that its precondition is satisfied, and then add the expected postcondition into its residual

[2] If there is no ambiguity, we can use v^* instead of $\{v^*\}$.

$$\frac{[v^*]\vdash\{\Delta\}\ e_1\ \{\phi_2,\Delta_2,\beta_2\}\quad [v^*]\vdash\{\Delta_2\}\ e_2\ \{\phi_3,\Delta_3,\beta_3\}}{[v^*]\vdash\{\Delta\}\ e_1;e_2\ \{\phi_2\wedge\phi_3,\Delta_3,\beta_2\cup\beta_3\}}\ [\mathbf{INF-[SEQ]}]$$

$$\frac{[v^*]\vdash\{\Delta\wedge w'\}\ e_1\ \{\phi_2,\Delta_2,\beta_2\}\quad [v^*]\vdash\{\Delta\wedge\neg w'\}\ e_2\ \{\phi_3,\Delta_3,\beta_3\}}{[v^*]\vdash\{\Delta\}\ \mathtt{if}\ w\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \{\phi_2\wedge\phi_3,\Delta_2\vee\Delta_3,\beta_2\cup\beta_3\}}\ [\mathbf{INF-[IF]}]$$

$$\frac{[v^*]\vdash\{\Delta\}\ e\ \{\phi_2,\Delta_2,\beta_2\}\quad \Delta_3{=}\exists\mathtt{res}\cdot(\Delta_2\wedge_u u'{=}\mathtt{res})}{[v^*]\vdash\{\Delta\}\ u{:=}e\ \{\phi_2,\Delta_3,\beta_2\}}\ [\mathbf{INF-[ASSIGN]}]$$

$$\frac{t_0\ mn\ (\mathtt{ref}\ (t_i\ v_i)_{i=1}^{m-1},(t_j\ v_j)_{j=m}^{n})\ \ \Phi_{pr}\ \ \Phi_{po}\ \{c\}\in Prog\quad \rho{=}[v_k'/v_k]_{k=1}^n\quad \Phi_{pr}'=\rho(\Phi_{pr})\quad W{=}\{v_1,\ldots,v_{m-1}\}\quad V{=}\{v_m,\ldots,v_n\}\quad [v^*]\ \Delta\vdash\Phi_{pr}'\leadsto(\phi_2,\Delta_2,\beta_2)\quad \Delta_3{=}(\Delta_2\wedge nochange(V))\ *_{V\cup W}\ \Phi_{po}}{[v^*]\vdash\{\Delta\}\ mn(v_1,\ldots,v_{m-1},v_m,...v_n)\ \{\phi_2,\Delta_3,\beta_2\}}\ [\mathbf{INF-[CALL]}]$$

$$\frac{[v^*,v_{rel}^*]\vdash\{\Phi_{pr}\wedge\bigwedge(u'{=}u)^*\}\ c\ \{\phi_2,\Delta_2,\beta_2\}\quad [v^*,v_{rel}^*]\ \Delta_2\vdash\Phi_{po}\leadsto(\phi_3,\Delta_3,\beta_3)\quad \rho_1=infer\_pre(\beta_2\cup\beta_3)\quad \rho_2=infer\_post(\beta_2\cup\beta_3)\quad \Phi_{pr}^n=\rho_1(\Phi_{pr}\wedge\phi_2\wedge\phi_3)\quad \Phi_{po}^n=\rho_2(\Phi_{po}*\Delta_3)}{\vdash t_0\ mn\ ((t\ u)^*)\ \mathtt{infer}\ [v^*,v_{rel}^*]\ \Phi_{pr}\ \Phi_{po}\ \{c\}\leadsto\Phi_{pr}^n\ \Phi_{po}^n}\ [\mathbf{INF-[METH]}]$$

Figure 3: Hoare Rules with Pure Bi-Abduction

state, as illustrated in [INF−[CALL]]. Here, $(t_i\ v_i)_{i=1}^{m-1}$ are pass-by-reference parameters, that are marked with `ref`, while the pass-by-value parameters V are equated to their initial values through the *nochange* function, as their updated values are not visible in the methods callers. Note that inference may occur during the entailment of the method's precondition.

Lastly, we discuss the rule for handling each method declaration [INF−[METH]]. At the program level, our inference rules will be applied to each set of mutually-recursive methods in a bottom-up order in accordance with the call hierarchy. This allows us to gather the entire set β of definitions and obligations for each unknown relation. From this set we infer the unknown pre- and post-relations through steps (i) and (ii) described below. Take note that, given the entire set β of relational definitions and obligations, we retrieve the set of definitions and the set of obligations for post-relations through the functions $\mathtt{def}_{\mathtt{po}}$ and $\mathtt{obl}_{\mathtt{po}}$, respectively. Complementarily, we use functions $\mathtt{def}_{\mathtt{pr}}$ and $\mathtt{obl}_{\mathtt{pr}}$ for the pre-relations. With @pr and @po here, we mean to decide if an unknown comes from the precondition or the postcondition respectively.

$$\begin{array}{rcl}\mathtt{def}_{\mathtt{po}}(\beta) &=& \{\pi_i^k{\rightarrow}v_{rel_i}(v_i^*)\mid(\pi_i^k{\rightarrow}v_{rel_i}@\mathtt{po}(v_i^*))\in\beta\}\\ \mathtt{obl}_{\mathtt{po}}(\beta) &=& \{v_{rel_i}(v_i^*){\rightarrow}\pi_j\mid(v_{rel_i}@\mathtt{po}(v_i^*){\rightarrow}\pi_j)\in\beta\}\\ \mathtt{def}_{\mathtt{pr}}(\beta) &=& \{\pi_i^k{\rightarrow}v_{rel_i}(v_i^*)\mid(\pi_i^k{\rightarrow}v_{rel_i}@\mathtt{pr}(v_i^*))\in\beta\}\\ \mathtt{obl}_{\mathtt{pr}}(\beta) &=& \{v_{rel_i}(v_i^*){\rightarrow}\pi_j\mid(v_{rel_i}@\mathtt{pr}(v_i^*){\rightarrow}\pi_j)\in\beta\}\end{array}$$

(i) In order to infer the post-relations, function *infer_post* applies a least fixed point analysis to the set of gathered relational definitions $\mathtt{def}_{\mathtt{po}}(\beta)$. For computing the least fixed point over the two domains used to instantiate the current inference framework in the paper, namely the numerical domain and the set/bag domain, we utilize FIXCALC [20] and FIXBAG [18], respectively. The call to the fixed point analysis is denoted below as $\mathtt{LFP}(\mathtt{def}_{\mathtt{po}}(\beta))$, It takes as inputs the set of relational def-

initions, while returning a set of closed form constraints of the form $\pi_i \rightarrow v_{rel_i}(v_i^*)$, where each constraint corresponds to an unknown relation v_{rel_i}. Given that we aim at inferring strong post-relations that must satisfy the relational obligations from $\texttt{obl}_{\texttt{po}}(\beta)$, we further consider each unknown post-relation $v_{rel_i}(v_i^*)$ to be equal to π_i. Finally, *infer_post* returns a set of substitutions, where each unknown relation is to be substituted by the inferred formula, given that this formula satisfies all the relational obligations from $\texttt{obl}_{\texttt{po}}(\beta)$ corresponding to that particular relation.

$$\begin{array}{r}\mathit{infer\_post}(\beta)=\{\pi_i/v_{rel_i}(v_i^*) \mid (\pi_i \rightarrow v_{rel_i}(v_i^*)) \in \texttt{LFP}(\texttt{def}_{\texttt{po}}(\beta)) \\ \wedge\ \forall(v_{rel_i}(v_i^*) \rightarrow \pi_j) \in \texttt{obl}_{\texttt{po}}(\beta) \cdot \pi_i \Rightarrow \pi_j\}\end{array}$$

(ii) For the pre-relations, our goal is to infer the weakest formulas. Hence, for each pre-relation, we first calculate the conjunction of all its obligations from $\texttt{obl}_{\texttt{pr}}(\beta)$ to obtain a sufficient precondition `pre_fst` for the base cases. To capture the precondition for an arbitrary recursive call, we need to derive the *recursive invariant* in order to relate the parameters of an arbitrary one to those of the first one. Such relation can be achieved via a top-down fixed point analysis [21]. Obviously, for every recursive call, the candidate precondition must satisfy the initial definition `pre_fst`. Thus, the approximation π_i for each relation $v_{rel_i}(v_i^*)$ will satisfy both the precondition for the first call ($\texttt{pre\_fst}_\texttt{i}$) and for an arbitrary recursive call ($\texttt{pre\_rec}_\texttt{i}$). The last step is to check the quality of candidate preconditions in order to keep the ones that satisfy not only the obligation but also definitions of each relation.

$$\frac{\begin{array}{c}\texttt{pre\_fst}= \{(\wedge_j \pi_j)/v_{rel_i}(v_i^*) \mid (v_{rel_i}(v_i^*) \rightarrow \pi_j) \in \texttt{obl}_{\texttt{pr}}(\beta)\} \\ \texttt{rec\_inv}= \texttt{TDFP}(\texttt{def}_{\texttt{pr}}(\beta)) \\ \texttt{pre\_rec}_\texttt{i}= \forall(\mathit{FV}(\texttt{rec\_inv}_\texttt{i}) - v_i^*) \cdot (\neg \texttt{rec\_inv}_\texttt{i} \vee \texttt{pre\_fst}_\texttt{i}) \\ \pi_i = \texttt{pre\_fst}_\texttt{i} \wedge \texttt{pre\_rec}_\texttt{i} \\ \texttt{res} \ = \ \texttt{sanity\_checking}(\{\pi_\texttt{i}/\texttt{v}_{\texttt{rel}_\texttt{i}}(\texttt{v}_\texttt{i}^*)\}, \texttt{obl}_{\texttt{pr}}, \texttt{def}_{\texttt{pr}})\end{array}}{\mathit{infer\_pre}(\beta) \ = \ \texttt{res}}$$

With the help of functions *infer_pre* and *infer_post*, we can next define the rule for deriving the pre- and postconditions, Φ_{pr}^n and Φ_{po}^n, of a method *mn*. Note that v_{rel}^* denotes the set of unknown relations that are to be inferred, whereas ρ_1 and ρ_2 represent the substitutions obtained for pre- and post-relations, respectively.

7. Soundness of Inference

In this section we outline the soundness properties for both the entailment procedure (with selective inference) and Hoare-style verification (with specification inference).

LEMMA 7.1 (Soundness of Entailment with Selective Inference).
Given: $[v^*]\ \Delta_1 \vdash \Delta_2 \rightsquigarrow (\phi, \Delta_3, \beta)$
We can show: $[\,]\ \Delta_1 \wedge \phi \wedge \bigwedge_{\alpha \in \beta} \alpha \vdash \Delta_2 \rightsquigarrow (\texttt{true}, \Delta_3, \emptyset)$

LEMMA 7.2 (Soundness of Hoare Logic with Spec Inference).
Given: $[v^*] \vdash \{\Delta_1\}\ e\ \{\phi, \Delta_2, \beta\}$
We can show: $[\,] \vdash \{\Delta_1 \wedge \phi \wedge \bigwedge_{\alpha \in \beta} \alpha\}\ e\ \{\texttt{true}, \Delta_2, \emptyset\}$

The conclusions of Lemmas 7.1 and 7.2 make references to their respective procedures *without* any inferable variables. These are equivalent to previously proposed entailment procedure and verifier (without inference) whose soundness, with respect to store semantics and program semantics have already been established in [6]. For Lemma 7.1, given that the underlying entailment procedure without inference capabilities is sound, the collection of suitable preconditions in [INF−[AND]], [INF−[PRE−DERIVE]], [INF−[LHS−CONTRA]] will only produce valid entailments. Moreover, if our framework is instantiated with sound fixed point procedures, the relations inferred over the definitions collected by [INF−[REL−DEFN]] will conform to the relational obligations from [INF−[REL−OBLG]]. Lemma 7.2 assumes preconditions of recursive methods are inferred using pre-relations. Both lemmas can be shown to hold by inductive proofs over their respective rules.

Proof: We first present two auxiliary propositions: Partial Substitution Law for Assertions and the Monotonicity rule [23].

The Monotonicity rule can be defined as follows:

$$\frac{p_0 \Rightarrow p_1 \quad q_0 \Rightarrow q_1}{p_0 * q_0 \Rightarrow p_1 * q_1}$$

while the latter proposition is stated below. Substitution is defined in the standard way. We begin by considering total substitutions that act upon all the free variables of a phrase: For any phrase p such that $\mathit{FV}(p) \subseteq \{v_1, \ldots, v_n\}$, we write

$$[e_1/v_1, \ldots, e_n/v_n](p)$$

to denote the phrase obtained from p by simultaneously substituting each expression e_i for the variable v_i. (When there are bound variables in p, they will be renamed to avoid capture.) When $\mathit{FV}(p)$ is not a subset of $\{v_1, \ldots, v_n\}$,

$$[e_1/v_1, \ldots, e_n/v_n](p)$$

abbreviates

$$[e_1/v_1, \ldots, e_n/v_n, v_1'/v_1', \ldots, v_k'/v_k'](p)$$

where $\{v_1', \ldots, v_k'\} = \mathit{FV}(p) - \{v_1, \ldots, v_n\}$.

PROPOSITION 1 (Partial Substitution Law for Assertions). *Suppose p is an assertion, and let δ abbreviate the substitution*

$$e_1/v_1, \ldots, e_n/v_n.$$

Then let s be a store such that $(\mathit{FV}(p) - \{v_1, \ldots, v_n\}) \cup \mathit{FV}(e_1) \cup \ldots \cup \mathit{FV}(e_n) \subseteq \texttt{dom}\ s$, and let

$$\widehat{s} = [\ s \mid v_1 : [\![e_1]\!]_{\texttt{exp}} s \mid \ \ldots \ \mid v_n : [\![e_n]\!]_{\texttt{exp}} s\ ].$$

Then

$$s, h \models \delta(p) \ \texttt{iff}\ \widehat{s}, h \models p.$$

Here $[\ f \mid x_1 : y_1 \mid \ \ldots \ \mid x_n : y_n\ ]$ (where $x_1, \ldots, x_n$ are distinct) is meant for the function whose domain is the union of the domain of f with $\{x_1, \ldots, x_n\}$, that maps each x_i into y_i and all other members x of the domain of f into $f\ x$.

Lemma 7.1.
Using these auxiliary propositions, Lemma 7.1 can be proven by induction on entailment rules from Figure 2:

- Case [INF−[AND]]:
  Suppose $[v^*]\ \pi_1 \vdash \pi_2 \rightsquigarrow (\phi_2, \Delta_2, \beta_2)$ and
  $[v^*]\ \pi_1 \vdash \pi_3 \rightsquigarrow (\phi_3, \Delta_3, \beta_3)$.
  By the inductive hypothesis, we have:
  $[\,]\ \pi_1 \wedge \phi_2 \wedge \bigwedge_{\alpha \in \beta_2} \alpha \vdash \pi_2 \rightsquigarrow (\texttt{true}, \Delta_2, \emptyset)$ and
  $[\,]\ \pi_1 \wedge \phi_3 \wedge \bigwedge_{\alpha \in \beta_3} \alpha \vdash \pi_3 \rightsquigarrow (\texttt{true}, \Delta_3, \emptyset)$.
  According to the Monotonicity rule, we obtain:
  $[\,]\ \pi_1 \wedge \pi_1 \wedge \phi_2 \wedge \phi_3 \wedge \bigwedge_{\alpha \in \beta} \alpha \vdash \pi_2 \wedge \pi_3 \rightsquigarrow (\texttt{true}, \Delta_2 * \Delta_3, \emptyset)$
  $\Leftrightarrow \quad [\,]\ \pi_1 \wedge \phi_2 \wedge \phi_3 \wedge \bigwedge_{\alpha \in \beta} \alpha \vdash \pi_2 \wedge \pi_3 \rightsquigarrow (\texttt{true}, \Delta_2 * \Delta_3, \emptyset)$
  with $\beta = \beta_2 \cup \beta_3$.
  Thus, Lemma 7.1 holds for:
  $[v^*]\ \pi_1 \vdash \pi_2 \wedge \pi_3 \rightsquigarrow (\phi_2 \wedge \phi_3, \Delta_2 * \Delta_3, \beta_2 \cup \beta_3)$.
- Case [INF−[UNSAT]]: This case holds trivially.
- Case [INF−[VALID]]: This case holds trivially.
- Case [INF−[LHS−CONTRA]]:
  We need to show that:
  $[v^*]\ \pi \vdash \alpha \rightsquigarrow (\phi, \texttt{false}, \emptyset) \wedge \phi = \forall(\mathit{FV}(\pi) - v^*) \cdot \neg \pi$

$\Rightarrow$ $[\,]\ \pi{\wedge}\phi \vdash \alpha \rightsquigarrow (\texttt{true}\,,\texttt{false}\,,\emptyset)$

Suppose that the negation operator and quantifier elimination algorithms supported by provers are all sound, we need to show that with $\phi{=}\forall(FV(\pi){-}v^*){\cdot}\neg\pi$, we still have the contradiction between ϕ and π.

In fact,

- Case $\{v^*\}{=}\emptyset \Rightarrow \phi{\equiv}\texttt{false}$
- Case $\{v^*\}{=}FV(\pi) \Rightarrow \phi{\equiv}\neg\pi \Rightarrow \phi{\wedge}\pi{\equiv}\neg\pi{\wedge}\pi{\equiv}\texttt{false}$
- Otherwise:
  - if we can sufficiently express the contradiction of π using v^*, Lemma 7.1 is sound.
  - otherwise, $\phi{\equiv}\texttt{false}$. Since we require $\phi{\not\equiv}\texttt{false}$, this case we do not infer any preconditions.

- Case $[\underline{\textsc{inf}{-}[\textsc{pre}{-}\textsc{derive}]}]$:
  We need to show that:
  $[v^*]\ \pi \vdash \alpha \rightsquigarrow (\phi, \pi{\wedge}\phi, \emptyset) \wedge \phi{=}\forall(FV(\pi,\alpha){-}v^*)\cdot(\neg\pi{\vee}\alpha)$
  $\Rightarrow$ $[\,]\ \pi{\wedge}\phi \vdash \alpha \rightsquigarrow (\texttt{true}\,, \pi{\wedge}\phi, \emptyset)$
  Indeed,
  - Case $\{v^*\}{=}\emptyset \Rightarrow \phi{\equiv}\texttt{false}$
  - Case $\{v^*\}{=}FV(\pi,\alpha) \Rightarrow \phi{\equiv}\neg\pi \vee \alpha \Rightarrow \phi{\wedge}\pi{\equiv}\alpha{\wedge}\pi$
  - Otherwise:
    - if we can sufficiently express $\neg\pi{\vee}\alpha$ using v^*, Lemma 7.1 is sound.
    - otherwise, in the worst case, we can still infer a precondition as $\forall(FV(\pi,\alpha){-}v^*){\cdot}\neg\pi$. Thus, Lemma 7.1 holds.
- Case $[\underline{\textsc{inf}{-}[\textsc{rel}{-}\textsc{defn}]}]$: This case holds trivially.
- Case $[\underline{\textsc{inf}{-}[\textsc{rel}{-}\textsc{oblg}]}]$: This follows from the soundness of rule $[\underline{\textsc{inf}{-}[\textsc{pre}{-}\textsc{derive}]}]$.

Lemma 7.2.
Lemma 7.2 can be proven by induction on Hoare rules with pure bi-abduction from Figure 3:

- Case $[\underline{\textsc{inf}{-}[\textsc{seq}]}]$:
  Suppose $[v^*] \vdash \{\Delta\}\ e_1\ \{\phi_2, \Delta_2, \beta_2\}$ and
  $[v^*] \vdash \{\Delta_2\}\ e_2\ \{\phi_3, \Delta_3, \beta_3\}$.
  By the inductive hypothesis, we have:
  $$[\,] \vdash \{\Delta{\wedge}\phi_2{\wedge}\textstyle\bigwedge_{\alpha\in\beta_2}\alpha\}\ e_1\ \{\texttt{true}\,, \Delta_2, \emptyset\} \text{ and}$$
  $$[\,] \vdash \{\Delta_2{\wedge}\phi_3{\wedge}\textstyle\bigwedge_{\alpha\in\beta_3}\alpha\}\ e_2\ \{\texttt{true}\,, \Delta_3, \emptyset\} \quad (1)$$
  The Frame rule ensures that the following holds:
  $$[\,] \vdash \{\Delta{\wedge}\phi_2{\wedge}\phi_3{\wedge}\textstyle\bigwedge_{\alpha\in\beta}\alpha\}\ e_1\ \{\texttt{true}\,, \Delta_2{*}\phi_3{\wedge}\textstyle\bigwedge_{\alpha\in\beta_3}\alpha, \emptyset\} \quad (2)$$
  with $\beta = \beta_2{\cup}\beta_3$.
  From (1), (2) and the Sequential Composition rule, we obtain:
  $[\,] \vdash \{\Delta{\wedge}\phi_2{\wedge}\phi_3{\wedge}\textstyle\bigwedge_{\alpha\in(\beta_2\cup\beta_3)}\alpha\}\ e_1; e_2\ \{\texttt{true}\,, \Delta_3, \emptyset\}$.
  Thus, Lemma 7.2 holds for:
  $[v^*] \vdash \{\Delta\}\ e_1; e_2\ \{\phi_2{\wedge}\phi_3, \Delta_3, \beta_2{\cup}\beta_3\}$.
- Case $[\underline{\textsc{inf}{-}[\textsc{if}]}]$:
  Suppose $[v^*] \vdash \{\Delta{\wedge}w'\}\ e_1\ \{\phi_2, \Delta_2, \beta_2\}$ and
  $[v^*] \vdash \{\Delta{\wedge}\neg w'\}\ e_2\ \{\phi_3, \Delta_3, \beta_3\}$.
  By the inductive hypothesis, we have:
  $$[\,] \vdash \{\Delta{\wedge}\phi_2{\wedge}w'{\wedge}\textstyle\bigwedge_{\alpha\in\beta_2}\alpha\}\ e_1\ \{\texttt{true}\,, \Delta_2, \emptyset\} \quad (3)$$
  $$[\,] \vdash \{\Delta{\wedge}\phi_3{\wedge}\neg w'{\wedge}\textstyle\bigwedge_{\alpha\in\beta_3}\alpha\}\ e_2\ \{\texttt{true}\,, \Delta_3, \emptyset\} \quad (4)$$
  Applying Strengthening Precedent and Weakening Consequent rules for both (3) and (4):
  $[\,] \vdash \{\Delta{\wedge}\phi_2{\wedge}\phi_3{\wedge}\textstyle\bigwedge_{\alpha\in\beta}\alpha{\wedge}w'\}\ e_1\ \{\texttt{true}\,, \Delta_2{\vee}\Delta_3, \emptyset\}$
  $[\,] \vdash \{\Delta{\wedge}\phi_3{\wedge}\phi_2{\wedge}\textstyle\bigwedge_{\alpha\in\beta}\alpha{\wedge}\neg w'\}\ e_2\ \{\texttt{true}\,, \Delta_2{\vee}\Delta_3, \emptyset\}$
  with $\beta = \beta_2{\cup}\beta_3$.
  From the Conditional rule, we obtain:
  $[\,] \vdash \{\Delta{\wedge}\phi_2{\wedge}\phi_3{\wedge}\textstyle\bigwedge_{\alpha\in\beta}\alpha\}\ \texttt{if}\ w\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \{\texttt{true}\,, \Delta_2{\vee}\Delta_3, \emptyset\}$.
  Thus, Lemma 7.2 holds for:
  $[v^*] \vdash \{\Delta\}\ \texttt{if}\ w\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \{\phi_2{\wedge}\phi_3, \Delta_2{\vee}\Delta_3, \beta_2{\cup}\beta_3\}$.
- Case $[\underline{\textsc{inf}{-}[\textsc{assign}]}]$:
  Suppose $[v^*] \vdash \{\Delta\}\ e\ \{\phi_2, \Delta_2, \beta_2\}$.
  By the inductive hypothesis, we have:
  $[\,] \vdash \{\Delta{*}\phi_2{\wedge}\textstyle\bigwedge_{\alpha\in\beta_2}\alpha\}\ e\ \{\texttt{true}\,, \Delta_2, \emptyset\}$.
  With $\Delta_3{=}\exists\texttt{res}{\cdot}(\Delta_2 \wedge_u u'{=}\texttt{res})$, we obtain (similarly in [6]):
  $[\,] \vdash \{\Delta{*}\phi_2{\wedge}\textstyle\bigwedge_{\alpha\in\beta_2}\alpha\}\ u{:=}e\ \{\texttt{true}\,, \Delta_3, \emptyset\}$.
  Thus, Lemma 7.2 holds for:
  $[v^*] \vdash \{\Delta\}\ u{:=}e\ \{\phi_2, \Delta_3, \beta_2\}$.
- Case $[\underline{\textsc{inf}{-}[\textsc{call}]}]$:
  Suppose we have:
  $[v^*]\ \Delta \vdash \rho(\Phi_{pr}) \rightsquigarrow (\phi_2, \Delta_2, \beta_2)$
  with $\rho{=}[v'_k/v_k]_{k=1}^n$ and
  $t_0\ mn\ (\texttt{ref}\ (t_i\ v_i)_{i=1}^{m-1}, (t_j\ v_j)_{j=m}^{n})\ \ \Phi_{pr}\ \ \Phi_{po}\ \{c\} \in \mathit{Prog}$.
  By the inductive hypothesis, we obtain:
  $[\,]\ \Delta{\wedge}\phi_2{\wedge}\textstyle\bigwedge_{\alpha\in\beta_2}\alpha \vdash \rho(\Phi_{pr}) \rightsquigarrow (\texttt{true}\,, \Delta_2, \emptyset)$.
  Similarly in [6], we also have:
  $[\,] \vdash \{\Delta{*}\phi_2{\wedge}\textstyle\bigwedge_{\alpha\in\beta_2}\alpha\}\ mn(v_1,\ldots,v_{m-1},v_m,...v_n)\ \{\texttt{true}\,, \Delta_3, \emptyset\}$
  with $W{=}\{v_1,\ldots,v_{m-1}\}$, $V{=}\{v_m,\ldots,v_n\}$ and
  $\Delta_3{=}(\Delta_2 \wedge \mathit{nochange}(V))\ \ *_{V\cup W}\ \Phi_{po}$.
  Thus, Lemma 7.2 holds for:
  $[v^*] \vdash \{\Delta\}\ mn(v_1,\ldots,v_{m-1},v_m,...v_n)\ \{\phi_2, \Delta_3, \beta_2\}$ $\square$

Since both Lemma 7.1 and 7.2 has been proven to be sound, the soundness of rule $[\underline{\textsc{inf}{-}[\textsc{meth}]}]$ only depends on the soundness of fixpoint procedures, which are done via *infer_pre* and *infer_post* functions. The least fixpoint analysis (used in *infer_post*) is from [18, 20] while the top-down fixpoint procedure (used in *infer_pre*) is inspired by [21].

8. Formalization of Predicate Derivation

We outline how predicate extension and specialization are being implemented in our system. We introduce the following derivation judgement: $\Phi_1 \# \mathit{Dict} \Longrightarrow \Phi_2$, where Φ_1 denotes the original heap-based state, Dict is a dictionary built from the property definitions and predicate derivation commands, Φ_2 denotes the new heap state (where the old predicates have been replaced by the derived ones), together with derived pure constraint.

Our system first builds a dictionary where each derivation command has an entry that associates each derived predicate with the properties that it uses. Generically, a dictionary entry looks as $(npred\langle v^*, w^*\rangle, [@V^*], [@R^*], base, rec)$, where $npred$ is the name of the derived heap predicate, v^* are the arguments from the original heap predicate, and w^* are the arguments for the new properties being defined, $[@V^*]$ represents the list of all the value field annotations and $[@R^*]$ denotes the list of the recursive pointer annotations from the property definitions used, and $base$ and rec are meta-predicates describing the base and inductive cases of the extension derivation. For parameters that accept multiple occurrences, we shall use the annotation # to mark them, for e.g. `V#`, `R#`.

For illustration, let us consider the derivation command for the `llhMM` predicate from Sec 4. Though the derivation invokes two properties, `minP`$[$`@Val`, `@Rec`$]\langle$`mi`$\rangle$ and `head`$[$`@Val`$]\langle$`v`$\rangle$, the dictio-

nary will contain only one entry, as shown below.

$$(\texttt{llhMM}\langle\texttt{root},\texttt{v},\texttt{mi}\rangle, [@\texttt{Val}], [@\texttt{Rec}\#], base, rec)$$

The `head` property uses only `@Val`, but the `min` property uses both arguments from `@Val`, `@Rec#` with the latter annotated with a # marker to allow multiple occurrences of its recursive pointers that is associated with an extracted `min` operator. Corresponding to this is a list of minimum properties $\texttt{m}\#_1$ that were taken from their recursive pointers. The corresponding meta-predicates, $base$ and rec, capture both `min` and `head` pure property definitions as given below:

$base = \setminus [\texttt{v}_1, \_] \rightarrow \texttt{v}{=}\texttt{v}_1 \wedge \texttt{mi}{=}\texttt{v}_i$
$rec = \setminus [\texttt{v}_1, \texttt{m}\#_1] \rightarrow \texttt{v}{=}\texttt{v}_1 \wedge \texttt{mi}{=}\texttt{min}(\texttt{v}_1, \texttt{fold}(\texttt{min}, \texttt{m}\#_1, \infty))$

Since $\texttt{m}\#_1$ is a list of values, we use a higher-order meta-operation $\texttt{fold}(\texttt{min}, \texttt{m}\#_1, \infty)$[3] to obtain the list's minimum. Corresponding to the `head` property, the single occurrence value field is used to provide a definition for the head `v`.

After the dictionary is constructed, the derivation judgement is used to derive the definitions of the new predicates. Consider again the derivation command for $\texttt{llhMM}\langle\texttt{root},\texttt{v},\texttt{mi}\rangle$ that is applied to the non-null version of predicate $\texttt{ll}\langle\texttt{root}\rangle$ given in Sec 4. Our system will apply simultaneously the two properties `min` and `head` to the base and inductive cases of the non-null definition, which results in two derivation judgements:

$\texttt{x}{\mapsto}\texttt{node}\langle\texttt{v}_1,\texttt{null}\rangle\#\texttt{Dict} \Longrightarrow \texttt{x}{\mapsto}\texttt{node}\langle\texttt{v}_1,\texttt{null}\rangle \wedge \texttt{mi}{=}\texttt{v}_1 \wedge \texttt{v}_1{=}\texttt{v}$
$\texttt{x}{\mapsto}\texttt{node}\langle\texttt{v}_1,\texttt{q}\rangle * \texttt{ll}\langle\texttt{q}\rangle\#\texttt{Dict} \Longrightarrow$
$\quad \texttt{x}{\mapsto}\texttt{node}\langle\texttt{v}_1,\texttt{q}\rangle * \texttt{llhMM}\langle\texttt{q},\texttt{v}_2,\texttt{mi}_2\rangle \wedge \texttt{mi}{=}\texttt{min}(\texttt{v}_1,\texttt{mi}_2) \wedge \texttt{v}_1{=}\texttt{v}$

After this, we obtain the following new predicate definition for `llhMM`, which is equivalent to the one provided in Sec 4:

$\texttt{pred}\ \texttt{llhMM}\langle\texttt{root},\texttt{v},\texttt{mi}\rangle \equiv \texttt{root}{\mapsto}\texttt{node}\langle\texttt{v}_1,\texttt{null}\rangle \wedge \texttt{mi}{=}\texttt{v}_1$
$\quad \wedge \texttt{v}_1{=}\texttt{v} \vee \texttt{root}{\mapsto}\texttt{node}\langle\texttt{v}_1,\texttt{q}\rangle * \texttt{llhMM}\langle\texttt{q},\texttt{v}_2,\texttt{mi}_2\rangle$
$\quad \wedge \texttt{mi}{=}\texttt{min}(\texttt{v}_1,\texttt{mi}_2) \wedge \texttt{v}_1{=}\texttt{v}.$

Take note that even though we illustrated the predicate derivation mechanism for predicate extension, the same procedure applies for predicate specialization. For instance, the dictionary entry for $\texttt{llA}\langle\texttt{root},\texttt{v},\texttt{mi}\rangle$ and $\texttt{SortA}[@\texttt{Val},@\texttt{Rec}]\langle\texttt{mi}\rangle$ from Sec 4 is $(\texttt{llA}\langle\texttt{root},\texttt{v},\texttt{mi}\rangle, [@\texttt{Val}], [@\texttt{Rec}\#], \setminus\_ \rightarrow \texttt{true}, \texttt{rec}_\texttt{s})$ where $\texttt{rec}_\texttt{s}$ is constructed as $\setminus [\texttt{v}_1,\texttt{m}\#_1] \rightarrow \texttt{v}_1 \leq \texttt{fold}(\texttt{min},\texttt{m}\#_1,\infty)$.

9. High-Level User Guidance

Let us look at how user guidance may be given to support predicate derivations and specification inference. Our commands are directed towards either predicate extension or specialization, prior to specification inference on corresponding pure properties. For example, given a predicate $\texttt{p}\langle\texttt{v}^*\rangle$, we may extend it with a new pure property $\texttt{prop}[@\texttt{V}^*]\langle\texttt{m}\rangle$, by applying the command $+\texttt{prop}[@\texttt{V}^*]\langle\,\rangle$ to the given predicate using $\texttt{p}\langle\texttt{v}^*\rangle{+}\texttt{prop}[@\texttt{V}^*]\langle\,\rangle$. Such a command would automatically generate a predicate derivation of the form:

$$\texttt{np}\langle\texttt{v}^*,\texttt{m}\rangle \ =\ \texttt{derive}\ \texttt{p}\langle\texttt{v}^*\rangle\ \texttt{with}\ \texttt{prop}[@\texttt{V}^*]\langle\texttt{m}\rangle$$

prior to the systematic replacement of $\texttt{p}\langle\texttt{v}^*\rangle$ by $\texttt{np}\langle\texttt{v}^*,\texttt{m}\rangle$ in support of pure property inference. Correspondingly, if we had defined a template for specialization $\texttt{spec}[@\texttt{V}^*]\langle\,\rangle$, we can explore the possibility of using specialized predicates by applying the command $+\texttt{spec}[@\texttt{V}^*]\langle\,\rangle$ to each predicate of interests. This command would initially generate the specialized predicate, before exploring the possibility of automatically replacing the current predicate with its specialized counterpart, where possible. We achieve this by first performing the required predicate extensions, followed by pure property inference. Once this is done, we would consider if the specialized predicates can be used or not, using a simple strategy[4] that relied on user guidance to indicate which of the predicates could be replaced by their specialized counterparts.

To facilitate flexible interaction with the user, we provide three levels of user guidance. At the top level, the user may issue guidance commands that are applicable to the entire program. This is done using stand-alone commands of the form `infer` [`cmd`]; which would be effective for all methods encountered after command has been issued. At the next level, we may issue a user guidance for the entire pre/post specification of a given method. Such a command takes the form `infer` [`cmd`, $\texttt{v}^*$] `requires pre ensures post`, where the `cmd` would be used to guide predicate derivation (if not already done), and the inference of relevant pure properties. Lastly, we may apply our command to a specific predicate, say $\texttt{p}\langle\texttt{v}^*\rangle$, in a pre/post specification. This is simply done via command $\texttt{p}\langle\texttt{v}^*\rangle{+}\texttt{cmd}$.

Our high-level user commands have the following form:

$$\begin{array}{rcl} \mathit{cmd} & ::= & \texttt{extn}[@\texttt{V}^*]\langle\,\rangle \mid \texttt{spec}[@\texttt{V}^*]\langle\,\rangle \mid \mathit{cmd}{+}\mathit{cmd} \\ & & \mid \{\mathit{cmd},..,\mathit{cmd}\} \mid \texttt{p}\langle\texttt{v}^*\rangle{+}\mathit{cmd} \end{array}$$

When a command is specified without any reference to predicates, we shall apply it to every predicate in its scope. Otherwise, the command is only applicable to the specified predicate of interests. We also allow two guidance commands `cmd1`, `cmd2` to be *simultaneously* applied using {`cmd1`, `cmd2`}, or *incrementally* applied using $\texttt{cmd1}{+}\texttt{cmd2}$. In summary, our guidance command provides a high-level and flexible mechanism for supporting pure property inference, through predicate extension and specialization.

10. Experimental Results

We have implemented our pure bi-abduction technique into an automated program verification system for separation logic, giving us a new version with inference capability, called SPECINFER. We have conducted three case studies in order to examine (i) the quality of inferred specifications, (ii) the feasibility of our technique in dealing with a variety of data structures and pure properties to be inferred, and (iii) the applicability of our tool in real benchmarks.

Small Examples

To highlight the quality of inferred specifications, we summarize *sufficient specifications* (that our tool can infer) for some well-known recursive examples in Table 1. Each procedure can take as its inputs the singly linked lists (pointed by `x` or `y`), the `i_th` index and an integer value `a`. Though the codes for these examples are not complicated, they illustrate the treatment of recursion (and, thus, the inter-procedural aspect). Therefore, the preconditions and postconditions derived can be rather intricate and would require considerable human efforts if they were constructed manually.[5]

We start with specifications that are obtained from the prior shape analysis step, and enrich the heap part with additional size property or multi-set/bag property. In Table 1, inferred constraints are highlighted using grayed background.

One interesting thing to note is that each example may require a different level of correctness to allow its correctness to be fully captured and verified. As stated in Sec. 9, such level can be achieved with the help of appropriate guidance. For instance, in Table 1, the size property is not enough to fully capture the functional correctness of `del_val` method, whose source code is described below.

```
1  node del_val(node x, int a) {
2    if (x == null)
3      return x;
```

[3] Similar to a `List.fold_right` reduction operation found in OCaml

[4] We could design smarter strategies for deciding when specialized predicates can be used, but this topic is beyond the scope of the current paper.

[5] The source code of all examples can be found at:
http://loris-7.ddns.comp.nus.edu.sg/~project/SpecInfer/

	Size Property		Size + Bag Property	
Method	Pre	Post	Pre	Post
append(x, y)	$\texttt{lln}\langle \texttt{x}, \texttt{n}\rangle * \texttt{lln}\langle \texttt{y}, \texttt{m}\rangle \wedge \texttt{x}\neq\texttt{null}$	$\texttt{lln}\langle \texttt{x}, \texttt{z}\rangle \wedge \texttt{z}=\texttt{n}+\texttt{m}$	$\texttt{llB}\langle \texttt{x}, \texttt{n}, \texttt{B}_1\rangle * \texttt{llB}\langle \texttt{y}, \texttt{m}, \texttt{B}_2\rangle \wedge \texttt{x}\neq\texttt{null}$	$\texttt{llB}\langle \texttt{x}, \texttt{z}, \texttt{B}_3\rangle \wedge \texttt{B}_3=\texttt{B}_1\sqcup\texttt{B}_2$
copy(x)	$\texttt{lln}\langle \texttt{x}, \texttt{n}\rangle$	$\texttt{lln}\langle \texttt{x}, \texttt{m}\rangle * \texttt{lln}\langle \texttt{res}, \texttt{z}\rangle \wedge \texttt{n}=\texttt{m} \wedge \texttt{n}=\texttt{z}$	$\texttt{llB}\langle \texttt{x}, \texttt{n}, \texttt{B}_1\rangle$	$\texttt{llB}\langle \texttt{x}, \texttt{m}, \texttt{B}_2\rangle * \texttt{llB}\langle \texttt{res}, \texttt{z}, \texttt{B}_3\rangle \wedge \texttt{B}_1=\texttt{B}_2 \wedge \texttt{B}_2=\texttt{B}_3$
del_index(x, i)	$\texttt{lln}\langle \texttt{x}, \texttt{n}\rangle \wedge 1\leq\texttt{i}\wedge\texttt{i}<\texttt{n}$	$\texttt{lln}\langle \texttt{x}, \texttt{m}\rangle \wedge \texttt{n}=1+\texttt{m}$	$\texttt{llB}\langle \texttt{x}, \texttt{n}, \texttt{B}_1\rangle \wedge 1\leq\texttt{i}\wedge\texttt{i}<\texttt{n}$	$\texttt{llB}\langle \texttt{x}, \texttt{m}, \texttt{B}_2\rangle \wedge \texttt{n}=1+\texttt{m} \wedge \texttt{B}_2\sqsubset\texttt{B}_1$
del_val(x, a)	$\texttt{lln}\langle \texttt{x}, \texttt{n}\rangle$	$\texttt{lln}\langle \texttt{res}, \texttt{m}\rangle \wedge \texttt{n}\geq\texttt{m}\geq\texttt{n}-1$	$\texttt{llB}\langle \texttt{x}, \texttt{n}, \texttt{B}_1\rangle$	$\texttt{llB}\langle \texttt{res}, \texttt{m}, \texttt{B}_2\rangle \wedge (\texttt{B}_1=\texttt{B}_2\sqcup\{\texttt{a}\} \vee(\texttt{a}\notin\texttt{B}_1 \wedge \texttt{B}_2=\texttt{B}_1))$
insert(x, a)	$\texttt{lln}\langle \texttt{x}, \texttt{n}\rangle \wedge\texttt{x}\neq\texttt{null}$	$\texttt{lln}\langle \texttt{x}, \texttt{m}\rangle \wedge \texttt{m}=\texttt{n}+1$	$\texttt{llB}\langle \texttt{x}, \texttt{n}, \texttt{B}_1\rangle \wedge\texttt{x}\neq\texttt{null}$	$\texttt{llB}\langle \texttt{x}, \texttt{m}, \texttt{B}_2\rangle \wedge \texttt{B}_2=\texttt{B}_1\sqcup\{\texttt{a}\}$
traverse(x)	$\texttt{lln}\langle \texttt{x}, \texttt{n}\rangle$	$\texttt{lln}\langle \texttt{x}, \texttt{m}\rangle \wedge \texttt{m}=\texttt{n}$	$\texttt{llB}\langle \texttt{x}, \texttt{n}, \texttt{B}_1\rangle$	$\texttt{llB}\langle \texttt{x}, \texttt{m}, \texttt{B}_2\rangle \wedge \texttt{B}_2=\texttt{B}_1$

Table 1. Pre-/Post-Conditions Inferred for Selected List-Based Examples

			Shape Prop.		Shape + Quantitative Properties				Shape + Quan. + Functional Properties			
Program	LOC	P#	Proven#	%	Add. Properties	Proven#	%	Time	Add. Properties	Proven#	%	Time
LList	287	29	23	79	Size	29	100	1.53	Bag of values	29	100	3.09
SoLList	237	28	22	79	Size	28	100	0.93	Sortedness	28	100	1.62
DLList	313	29	23	79	Size	29	100	1.69	Bag of values	29	100	4.19
Heaps	179	5	2	40	Size	4	80	2.14	Max. element	5	100	6.63
CBTree	115	7	7	100	Size & Height	7	100	2.76	Bag of values	7	100	98.81
AVLTree	313	11	9	82	Size & Height	11	100	8.85	Balance factor	11	100	10.66
BSTree	177	9	9	100	Size & Height	9	100	1.76	Sortedness	9	100	2.75
RBTree	407	19	18	95	Size & Height	19	100	5.97	Color	19	100	6.01

Table 2. Specification Inference with Pure Properties for a Variety of Data Structures

			Shape Property		Shape + Size Property		
Program	LOC	P#	Proven#	%	Proven#	%	Time
schedule	512	19	14	75	19	100	6.86
schedule2	474	14	6	43	14	100	10.58

Table 3. Specification Inference with Size Property for Real Programs

```
4     else
5       if (x.val == a) {
6         node tmp = x.next;
7         free(x);
8         return tmp; }
9       else {
10        x.next = del_val(x.next, a);
11        return x; }
12  }
```

Method del_val deletes the first node whose value is equal to value a from the linked-list pointed by x. Since the behavior of this method depends on information of values stored in the list, the user needs to guide SPECINFER via command:

$$\texttt{infer}\,[\texttt{ll}\langle\texttt{root}\rangle+\{\texttt{size}[@\texttt{Rec}]\langle\rangle, \texttt{set}[@\texttt{Val}, @\texttt{Rec}]\langle\rangle\}]$$

in order to obtain llB predicate, which also captures a multi-set B of values stored in the list:

$$\texttt{pred}\ \texttt{llB}\langle\texttt{root}, \texttt{n}, \texttt{B}\rangle \equiv (\texttt{root}=\texttt{null} \wedge \texttt{n}=0 \wedge \texttt{B}=\{\}) \vee \exists\, \texttt{s}, \texttt{q}, \texttt{B}_0 \cdot (\texttt{root}\mapsto\texttt{node}\langle\texttt{s}, \texttt{q}\rangle * \texttt{llB}\langle\texttt{q}, \texttt{n}-1, \texttt{B}_0\rangle \wedge \texttt{B}=\texttt{B}_0\sqcup\{\texttt{s}\}).$$

Accordingly, our tool is capable of inferring the following specification for the del_val method:

$$\texttt{requires}\ \texttt{llB}\langle\texttt{x}, \texttt{n}, \texttt{B}_1\rangle$$
$$\texttt{ensures}\ \texttt{llB}\langle\texttt{res}, \texttt{m}, \texttt{B}_2\rangle \wedge ((\texttt{a}\notin\texttt{B}_1 \wedge \texttt{B}_2=\texttt{B}_1) \vee \texttt{B}_1=\texttt{B}_2\sqcup\{\texttt{a}\});$$

where res denotes the method's result.

Medium Examples

We tested our tool on a set of challenging programs that manipulate a variety of data structures. The results are shown in Table 2, where the first column contains tested programs: LList (singly-linked list), SoLList (sorted singly-linked list), DLList (doubly-linked list), CBTree (complete binary tree), Heaps (priority queue), AVLTree (AVL tree), BSTree (binary search tree) and RBTree (red-black tree). The second and third columns denote the number of lines of code (LOC) and the number of procedures (P#) respectively.

For each test, we first start with shape specifications that are obtained from the prior shape analysis step. The number of procedures with consistent specifications is reported in the Proven# column while the percentage of these over all analyzed procedures is in the

% column. In the next two phases, we incrementally add new pure properties (to be inferred) to the existing specifications. These additional properties are listed in the `Add.Properties` column. While the second phase only focuses on quantitative properties such as size (number of nodes) and height (for trees), the third one aims at other functional properties. We also measure the time (in seconds) taken for verification with selective inference, in the `Time` column.

For procedures that SPECINFER cannot infer any consistent specifications, we try to construct the specifications manually. However due to the restriction of properties the resulting specification can capture, we fail to do so for these procedures. More specifically, we cannot construct any consistent specification for about 18% of procedures in phase 1. Even in phase 2, there is still one example (`delete_max` method) in `Heaps` test, for which we cannot obtain any consistent specification. This method is used to delete the root of a heap tree, thus it requires the information about the maximum element.

Open Source Programs

Table 3 demonstrates the applicability of SPECINFER on real open source programs, where finding preconditions for each procedural calls requires the precision of inference. The two programs are adopted from Siemens test suites [8] that have been used to perform process scheduling. Note that for these case studies we enrich the shape specification with size property only.

11. Related Works

Specification inference can be used to support formal software documentation. It is also important for making program verification more practical by minimizing on the need for manual provision of specifications. One research direction in the area of specification inference is concerned with inferring shapes of data structures. SLAyer [2] is an automatic program analysis tool designed to prove the absence of memory safety errors such as dangling pointer dereferences, double frees, and memory leaks. Other two tools focusing on memory safety are SALSA (Scalable Analysis via Lazy Scope expAnsion) [13], which verifies the safety of pointer dereferences through a staged expanding-scope algorithm for inter-procedural abstract interpretation, and Undangle [3], which proposes a runtime approach for the early detection of use-after-free and double-free vulnerabilities. The footprint analysis described in [4] infers descriptions of data structures without requiring a given precondition. Furthermore, in [5], Calcagno et al propose a compositional shape analysis. Both aforementioned analyses use an abstract domain based on a limited fragment of separation logic, centered around some common heap-centered predicates. Abductor [5] is a tool implementing a compositional shape analysis based on bi-abduction, which was used to check memory safety of large open source codebases [7]. A recent work [25] attempts to infer partial annotations required in a separation-logic based verifier, called Verifast. It can infer annotations related to unfold/fold steps, and also shape analysis when pre-condition is given. Our current proposal is complementary to the aforesaid works, as it is focused on inferring the more varied pure properties. We support it with a set of fundamental pure bi-abduction techniques, together with a general predicate extension and specialization mechanism. Our aim is to provide a systematic machinery for deriving formal specifications with more precise functional correctness properties.

A closely related research direction concerns the inference of both shape and numerical properties. In [14], the authors combine shape analysis based on separation logic with an external numeric-based program analysis in order to verify properties such as memory-safety and absence of memory leaks. Their method was tested on a number of programs where memory safety depends on relationships between the lengths of the lists involved. In the same category, Thor is a tool for reasoning about a combination of list reasoning and arithmetic by using two separate analyses [15]. Their goal is to investigate how precise an analysis could be, while maintaining full automation. The arithmetic support added by Thor includes stack-based integers, integers in the heap, lengths of lists. However, the current work is limited to handling list segments together with its length as property, and does not cover other pure properties, such as min/max or set. In addition, they require two separate analysis, as opposed to an integrated analysis (or entailment procedure) that can handle both heap and pure properties simultaneously. On the type system side, the authors of [11, 12, 24] require templates in order to infer dependent types precise enough to prove a variety of safety properties such as safety of array accesses. However, mutable data structures are not supported there. Compared to these works, our proposal may be considered fundamental, as we seek to incorporate pure property inference directly into the entailment proving process for the underlying logics, as opposed to building more complex analyses techniques.

12. Conclusion

We have proposed a new bi-abductive entailment proving procedure that is capable of inferring reasonably precise and concise pure properties for our specifications. This new entailment procedure forms the cornerstone of our approach towards incremental development of specifications to support higher-levels of program correctness. We have designed a set of fundamental mechanisms that can collect relational definitions and obligations for uninterpreted relations. These can be used to infer weaker pre-condition and stronger post-condition for our specifications. If the constraints are recursive, we can also rely on suitable fix-point analysis techniques to determine closed-form approximations. To support incremental development of specification, we have also designed a generic mechanism for heap predicate extension and specialization. These can act as high-level user guidance on the set of pure properties that are to be systematically inferred. We have also implemented our ideas in a prototype specification inference system, and have shown that it can be used to infer a broad range of program properties, from ensuring memory safety, to proving program termination and also various aspects of functional correctness.

References

[1] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, Springer LNCS 4111, pages 115–137, 2006.

[2] J. Berdine, B. Cook, and S. Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, pages 178–183, 2011.

[3] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA*, pages 133–143, 2012.

[4] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In *SAS*, pages 402–418, 2007.

[5] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2.

[6] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 2011.

[7] D. Distefano. Attacking large industrial code with bi-abductive inference. In *FMICS*, pages 1–8, 2009.

[8] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10:405–435, October 2005. ISSN 1382-3256.

[9] K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *CAV*, pages 372–378, 2011.

[10] B. S. Gulavani, S. Chakraborty, G. Ramalingam, and A. V. Nori. Bottom-up shape analysis. In *SAS*, pages 188–204, 2009.

[11] M. Kawaguchi, P. M. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, pages 304–315, 2009.

[12] M. Kawaguchi, P. M. Rondon, and R. Jhala. Dsolve: Safety verification via liquid types. In *CAV*, pages 123–126, 2010.

[13] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. G. Nanda. Verifying dereference safety via expanding-scope analysis. In *ISSTA*, pages 213–224, 2008.

[14] S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, pages 419–436, 2007.

[15] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Thor: A tool for reasoning about shape and arithmetic. In *CAV*, pages 428–432, 2008.

[16] H. Nguyen, C. David, S. Qin, and W. Chin. Automated Verification of Shape And Size Properties via Separation Logic. In *VMCAI*, pages 251–266, Jan. 2007.

[17] P. W. O'Hearn, J. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In *Proceedings of the 15th Annual Conference of the European Association for Computer Science Logic*, Paris, France, Sept. 2001.

[18] T.-H. Pham, M.-T. Trinh, A.-H. Truong, and W.-N. Chin. Fixbag: A fixpoint calculator for quantified bag constraints. In *CAV*, pages 656–662, 2011.

[19] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.

[20] C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *ASIAN*, pages 331–345, 2006.

[21] C. Popeea, D. N. Xu, and W.-N. Chin. A practical and precise inference and specializer for array bound checks elimination. In *PEPM*, pages 177–187, 2008.

[22] S. Qin, C. Luo, W.-N. Chin, and G. He. Automatically refining partial specifications for program verification. In *FM*, pages 369–385, 2011.

[23] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, pages 55–74, 2002.

[24] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.

[25] F. Vogels, B. Jacobs, F. Piessens, and J. Smans. Annotation inference for separation logic based verifiers. In *FMOODS/FORTE*, pages 319–333, 2011.