Bi-Abduction with Pure Properties 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 a state-of-the-art logic for dealing with the heap. Using its frame rule, initial works have strived towards automated modular verification for heap-manipulating programs against user-supplied specifications. Since manually writing specifications is a tedious and error-prone engineering process, the so-called bi-abduction (a combination of the frame rule and abductive inference) is proposed to automatically infer pre/post specifications on data structure shapes. However, it has omitted the inference of *pure properties* of data structures such as their size, sum, height, content and min/max, which are needed to express a higher level of program correctness. In this paper, we propose a novel approach, called *pure bi-abduction*, for inferring pure information for pre/post specifications, using the result from a prior shape analysis step. Additionally, we design a *predicate extension* mechanism to systematically extend shape predicates with pure properties. We have implemented our inference mechanism and evaluated its utility on a benchmark of programs. We show that pure properties are prerequisite to allow the correctness of about 20% of analyzed procedures to be captured and verified.

1 Introduction

One of the challenging areas for software verification concerns programs using heap-based data structures. In the last decade, research methodologies based on separation logic have offered good solutions [1,14,3] for proving the correctness of such programs.

Separation logic [21,15], an extension of Hoare logic, is a state-of-the-art logic for dealing with the heap. Its assertion language can succinctly describe how data structures are laid out in memory, by providing the separating conjunction operator that splits the heap into disjoint regions: reasoning about each such region is independent of the others. This local reasoning is captured by the frame rule of separation logic, a proof rule that enables compositional verification of heap-manipulating programs.

Initial works [1,14] based on separation logic have strived towards automated compositional verification against user-supplied specifications. However, manually writing specifications is a tedious and error-prone engineering process. Thus, more recent separation logic-based shape analyses endeavor to automatically construct such specifications in order to prove that programs do not commit pointer-safety errors (dereferencing a null or dangling pointer, or leaking memory). One such leading shape analysis [3] proposes bi-abduction to be able to scale up to millions lines of codes. Bi-abduction, a combination of the frame rule and abductive inference, is able to infer "frames" describing extra, unneeded portions of state (via the frame rule) as well as the needed,

missing portions (via abductive inference). Consequently, it would automatically infer both preconditions and postconditions on the shape of the data structures used by program codes, enabling a compositional and scalable shape analysis.

However, bi-abduction in [3] (also shape analyses) presently suffers from an inability to analyze for pure (i.e., heap-independent) properties of data structures, which are needed to express a higher-level of program correctness. For illustration, consider a simple C-style code example in Ex. 1 that zips two lists of integers into a single one. To reduce the performance overhead of redundant null-checking, in `zip` method there is no null-checking for `y`. As a result, the field access `y.next` on line 7 may not be memory-safe. In fact, it triggers a null-dereferencing error whenever the list pointed by x is longer than the list pointed by y. Naturally, to ensure memory-safety, the method's precondition needs to capture the size of each list.

Ex. 1. A method where pure properties of its data structure are critical for proving its memory safety

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

A direct solution to such limitation is to rely on numerical analyses. However, since numerical static analyses are often unaware of the shape of a program's heap, it is really difficult for them to capture pure properties of heap-based data structures.

In this paper, we propose a systematic methodology for inferring pure information for pre/post specifications in the separation logic domain, by utilizing the result from a prior shape analysis step. This pure information is not only critical for proving memory safety but also helpful to express a higher-level of program correctness. We call our inference methodology *pure bi-abduction*, and employ it for inferring pure properties of data structures such as their size, height, sum, content and minimum/maximum value. Like bi-abduction, pure bi-abduction is meant to combine the frame rule and abductive inference, but focused on strengthening pre/post shape specification with inferred pure information (Section 4).

Though the main novelty of our current work is a systematic inference of pure information for specifications of heap-manipulating programs, we have also devised a *predicate extension* mechanism that can systematically transform shape predicates in order to incorporate new pure properties. This technique is crucial for enhancing inductive shape predicates with relevant pure properties.

Our technical contributions include the following:

- We design a new bi-abductive entailment procedure for inferring pure information for specifications of heap-manipulating programs. The power of our procedure is significantly enhanced by its collection of pure proof obligations over *uninterpreted relations (functions)* (Sections 4, 5, 6).
- We propose an extension mechanism for systematically enhancing inductive shape predicates with a variety of pure properties (Section 7).
- We have implemented our approach and evaluated it on a benchmark of programs (Sec 8). We show that pure properties are prerequisite to allow the correctness of about 20% of analyzed procedures to be captured and verified.

2 Overview and Motivation

Memory Safety. For the `zip` method, by using shape analysis techniques [3,8], we could only obtain the following shape specification:

$$\begin{array}{ll} & \texttt{requires ll}\langle \texttt{x}\rangle * \texttt{ll}\langle \texttt{y}\rangle \\ & \texttt{ensures ll}\langle \texttt{res}\rangle; \\ \texttt{where} \quad \texttt{pred ll}\langle \texttt{root}\rangle \equiv (\texttt{root}{=}\texttt{null}) \vee \exists\, \texttt{q}\cdot(\texttt{root}{\mapsto}\texttt{node}\langle \_, \texttt{q}\rangle * \texttt{ll}\langle \texttt{q}\rangle). & \end{array}$$

Although the specification cannot ensure memory safety for `y.next` field access (Line 7,9), it still illustrates two important characteristics of separation logic. First, by using separation logic, the assertion language can provide inductive spatial predicates that describe the shape of unbounded linked data structures such as lists, trees, etc. Here, `ll` predicate describes the shape of an acyclic singly-linked list pointed by `root`. In its definition, the first disjunct corresponds to the case of an empty list, whereas the second one separates a list into two parts: the head $\texttt{root}{\mapsto}\texttt{node}\langle \_, \texttt{q}\rangle$, where $\mapsto$ denotes a points-to operator, and the tail $\texttt{ll}\langle \texttt{q}\rangle$. Second, the use of (separating conjunction) $*$ operator guarantees that these two parts reside in disjoint memory regions. In short, for the `zip` method, its precondition requires `x` and `y` point to linked lists (using `ll`) that reside in disjoint memory regions (using $*$), while its postcondition ensures the result also points to a linked list.

Generally, we cannot obtain any valid pre/post specification (valid Hoare triple) for the `zip` method using the shape domain only. To ensure memory safety, the specification must capture the size of each list. Using predicate extension mechanism, we first inject the size property (captured by `n`) into the `ll` predicate to derive the `llN` predicate as follows:

$$\begin{array}{l} \texttt{pred llN}\langle \texttt{root}, \texttt{n}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{n}{=}0) \\ \vee \exists\, \texttt{q}, \texttt{m}\cdot(\texttt{root}{\mapsto}\texttt{node}\langle \_, \texttt{q}\rangle * \texttt{llN}\langle \texttt{q}, \texttt{m}\rangle \wedge \texttt{n}{=}\texttt{m}{+}1). \end{array}$$

With the `llN` predicate, we could then strengthen the pre/post specification for `zip` to include uninterpreted relations: $\texttt{P}(\texttt{a}, \texttt{b})$ in the precondition and $\texttt{Q}(\texttt{r}, \texttt{a}, \texttt{b})$ in the postcondition. They are meant to capture the relationship between newly-introduced variables $\texttt{a}, \texttt{b}, \texttt{r}$ denoting size properties of linked lists. Uninterpreted relations in the precondition should be as weak as possible, while ones in the postcondition should be as strong as possible.

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

Intuitively, it is meant to incorporate the inference capability (via `infer`) into a pair of pre/post-condition (via `requires`/`ensures`). Here the inference will be applied to second-order variables $\texttt{P}, \texttt{Q}$.

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

$$\begin{array}{l} \texttt{P}(\texttt{a}, \texttt{b}) \Longrightarrow \texttt{b}{\neq}0 \vee \texttt{a}{\leq}0, \\ \texttt{P}(\texttt{a}, \texttt{b}) \wedge \texttt{a}{=}\texttt{ar}{+}1 \wedge \texttt{b}{=}\texttt{br}{+}1 \wedge 0{\leq}\texttt{ar} \wedge 0{\leq}\texttt{br} \Longrightarrow \texttt{P}(\texttt{ar}, \texttt{br}), \\ \texttt{P}(\texttt{a}, \texttt{b}) \wedge \texttt{r}{=}\texttt{b} \wedge \texttt{a}{=}0 \wedge 0{\leq}\texttt{b} \Longrightarrow \texttt{Q}(\texttt{r}, \texttt{a}, \texttt{b}), \\ \texttt{P}(\texttt{a}, \texttt{b}){\wedge}\texttt{rn}{=}\texttt{r}{-}2{\wedge}\texttt{bn}{=}\texttt{b}{-}1{\wedge}\texttt{an}{=}\texttt{a}{-}1{\wedge}0{\leq}\texttt{bn}, \texttt{an}, \texttt{rn}{\wedge}\texttt{Q}(\texttt{rn}, \texttt{an}, \texttt{bn}) \Longrightarrow \texttt{Q}(\texttt{r}, \texttt{a}, \texttt{b}). \end{array}$$

Using suitable fix-point analysis techniques, we can synthesize the approximations which would add a pure pre-condition $\texttt{a}{\leq}\texttt{b}$ that guarantees memory safety. More specifically, we have $\texttt{P}(\texttt{a},\texttt{b}) \equiv \texttt{a}{\leq}\texttt{b}$, $\texttt{Q}(\texttt{r},\texttt{a},\texttt{b}) \equiv \texttt{r}{=}\texttt{a}{+}\texttt{b}$ and a new specification that ensures memory safety:

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

Total Correctness. With inference of pure properties for specifications, we can go beyond memory safety towards functional correctness and even 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 [17] to support termination proofs. For this task, we would introduce an uninterpreted function $\texttt{F}(\texttt{a},\texttt{b})$, as a possible measure, via the following termination-based specification that is synthesized right after size inference. Thus, size inference is crucial for both program safety and program liveness proof.

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

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

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

Using suitable fixpoint analyses, we can synthesize $\texttt{F}(\texttt{a},\texttt{b}) \equiv \texttt{a}{-}1$, thus capturing a well-founded decreasing measure for our method. Though termination analysis of programs has been extensively investigated before, we find it refreshing to re-consider it in the context of pure property inference for pre/post specifications. (For space reason, we shall not consider this aspect that uses uninterpreted functions in the rest of the paper.)

3 Specification Language

In this section, we introduce the specification language used in pure bi-abduction (Figure 1). The language supports data type declarations *datat* (e.g. `node`), inductive shape predicate definitions *spred* (e.g. `ll`) and method specifications *spec*. 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.

Regarding each method's specification 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 while if $[v^*_{rel}]$ is specified, suitable approximations for these uninterpreted relations will be inferred.

The Φ constraint is in disjunctive normal form. Each disjunct consists of a $*$-separated heap constraint κ, referred to as *heap part*, and a heap-independent constraint π, referred to as *pure part*. The pure part does not contain any heap nodes and is presently restricted to uninterpreted relations $v_{rel}(v^*)$, pointer equality/disequality γ ($v_1{\neq}v_2$ and $v{\neq}\texttt{null}$ are just short forms for $\neg(v_1{=}v_2)$ and $\neg(v{=}\texttt{null})$ respectively), linear arithmetic *i*, boolean constraints *b* and bag constraints φ. Internally, each uninterpreted relation is annotated with `@pr` or `@po`, depending on whether it comes from the precondition or the postcondition, respectively. This information will be later used for inferring the formula corresponding to the uninterpreted relation (Sec 6). Furthermore, Δ denotes a composite formula that can be normalized into the Φ form, whereas ϕ represents a pure formula. The relational definitions and obligations (Sec 4.4) are denoted as $\pi{\rightarrow}v_{rel}(v^*)$ and $v_{rel}(v^*){\rightarrow}\alpha$, respectively.

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

Fig. 1: The Specification Language used in Pure Bi-Abduction

4 Principles of Pure Bi-Abduction

Initial works [1,14] are typically based on an 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.

To support shape analysis, bi-abduction [3] 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, the condition for the entailment checking to succeed, may be inferred.

In this paper, we propose pure bi-abduction technique for inferring pure properties for pre/post specifications. 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:

- 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.

• We allow *second-order variables*, in the form of *uninterpreted* relations, to support inference of pure properties for pre/post specifications.
• We then 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 analyses.

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.

4.1 Selective Inference

Our first principle is based on the notion that pure bi-abduction is best done selectively. Consider three 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 the entailment checking 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. The selectivity is important since we only consider a subset of variables (e.g. $\mathtt{a},\mathtt{b},\mathtt{r}$), which are introduced to capture pure properties of data structures. Note that this selectivity does not affect the automation of pure bi-abduction technique, since the variables of interest can be generated automatically right after applying predicate extension mechanism in Sec 7.

4.2 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.

4.3 Antecedent Contradiction

The problem of traditional abduction is to find an explanatory hypothesis such that it is satisfiable with the antecedent. Our purpose here is different in the sense that we want to find a sufficient precondition that would allow an entailment to succeed. Considering $[v^*]\ \Delta_1 \vdash \Delta_2$, 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. Although we disallow `false` to be inferred, we allow such above precondition if it is not equivalent to `false`. For example, with $[\mathtt{n}]\ \mathtt{x}{=}\mathtt{null}{\wedge}\mathtt{n}{=}0 \vdash \mathtt{x}{\neq}\mathtt{null}$, we have a contradiction between $\mathtt{x}{=}\mathtt{null}{\wedge}\mathtt{n}{=}0$ and $\mathtt{x}{\neq}\mathtt{null}$. To allow this entailment to succeed, we infer $\mathtt{n}{\neq}0$ as its precondition over just the selected variable $[\mathtt{n}]$.

4.4 Uninterpreted Relations

Our inference deals with uninterpreted relations that may appear in either preconditions or postconditions. We refer to the former as *pre-relations* and the latter as *post-relations*. Pre-relations are expected to be as weak as possible, while 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 uninterpreted relations (such as $R(v^*)$), we infer two kinds of relational constraints. The first kind, called *relational obligation*, is of the form $\pi \wedge R(v^*) \rightarrow c$, 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 $\pi \rightarrow R(v^*)$, where the uninterpreted relation is present in the consequent instead.

Relational obligations. They are useful in two ways. For pre-relations, they act as preconditions for the base cases of method definitions. For post-relations, they denote proof obligations that post-relations must satisfy. We could check these obligations after we have synthesized post-relations.

As an example, consider the entailment: $[\mathtt{P}]\ \mathtt{a} \geq 1 \wedge \mathtt{b} = 0 \wedge \mathtt{P}(\mathtt{a},\mathtt{b}) \vdash \mathtt{b} \neq 0$. We infer $\mathtt{P}(\mathtt{a},\mathtt{b}) \rightarrow \mathtt{a} \leq 0 \vee \mathtt{b} \neq 0$, which will denote a proof obligation for $\mathtt{P}$. More generally, with $[\mathtt{P}]\ \alpha_1 \wedge \mathtt{P}(v^*) \vdash \alpha_2$ where α_1 and α_2 denote *known* constraints, we first selectively infer precondition ϕ over selected variables v^* and then collect $\mathtt{P}(v^*) \rightarrow \phi$ as our relational obligation. To obtain succinct pre-conditions, we filter out constraints that contradicts with the current program state.

Relational definitions. They are typically used to form definitions for fixpoint analyses. For post-relations, we should infer the strongest definitions for them, where possible. After gathering the relational definitions (both base and inductive cases), we would apply a least fixpoint procedure [18] to discover suitable closed-form definitions for post-relations. For pre-relations, while it may be possible to compute greatest fixpoint to discover the weakest pre-relations that can satisfy all relational constraints, we have designed two simpler techniques for inferring pre-relations. After finding the interpretations for post-relations, we attempt to extract conditions on input variables from them. If the extracted conditions can satisfy all relational constraints for pre-relations, we simply use them as the approximations for our pre-relations. If not, we proceed with a second technique to first construct a recursive invariant which relates the parameters of an arbitrary call (e.g. $\mathtt{REC_a}, \mathtt{REC_b}$) to those of the first one (e.g. $\mathtt{a},\mathtt{b}$) using top-down fixpoint [19]. For example, a recursive invariant $\mathtt{rec\_inv}$ for $\mathtt{zip}$ method is $\mathtt{REC_a} \geq 0 \wedge \mathtt{a} \geq 1 + \mathtt{REC_a} \wedge \mathtt{REC_a} + \mathtt{b} = \mathtt{REC_b} + \mathtt{a}$. Next, since parameters of an arbitrary call must also satisfy relevant relational obligations, the precondition $\mathtt{pre\_rec}$ for a recursive call would then be $\forall \mathtt{REC_a}, \mathtt{REC_b} \cdot\ \mathtt{rec\_inv} \rightarrow \mathtt{pre\_fst}(\mathtt{REC_a}, \mathtt{REC_b})$, where $\mathtt{pre\_fst}(\mathtt{a},\mathtt{b}) = \mathtt{b} \neq 0 \vee \mathtt{a} \leq 0$ is the obligation. Finally, the precondition for all method invocations is $\mathtt{pre\_fst} \wedge \mathtt{pre\_rec} \wedge \mathtt{a} \geq 0 \wedge \mathtt{b} \geq 0 = 0 \leq \mathtt{a} \leq \mathtt{b}$. This approach allows us to avoid greatest fix-point analysis techniques, and is sufficient for all practical examples that we have evaluated.

5 Formalization of Pure Bi-Abduction

Recall that our entailment procedure for pure property has the following form:

$$[v^*]\ \Delta_1 \vdash \Delta_2 \rightsquigarrow (\phi_3, \Delta_3, \beta_3).$$

This new entailment procedure serves two key roles:

- From the verification point of view, the goal of the entailment procedure is to re-

duce 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 [14]. 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 (using *XPure* function as in [14]).

• From the inference point of view, the goal is inferring a precondition ϕ_3, as well as gathering the set of constraints β_3 over the uninterpreted relations. Together with the frame inferred Δ_3, we should be able to construct relevant preconditions and postconditions for each method.

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

$$\frac{}{[v^*]\ \pi_1 \vdash \pi_2 \rightsquigarrow (\phi_2, \Delta_2, \beta_2) \quad [v^*]\ \pi_1 \vdash \pi_3 \rightsquigarrow (\phi_3, \Delta_3, \beta_3)}\ [\textbf{INF}\text{-}[\textbf{AND}]]$$
$$[v^*]\ \pi_1 \vdash \pi_2 \wedge \pi_3 \rightsquigarrow (\phi_2 \wedge \phi_3, \Delta_2 \wedge \Delta_3, \beta_2 \cup \beta_3)$$

$$\frac{\textbf{UNSAT}(\alpha_1)}{[v^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\texttt{true}, \texttt{false}, \emptyset)}\ [\textbf{INF}\text{-}[\textbf{UNSAT}]] \qquad \frac{\alpha_1 \Rightarrow\ \alpha_2}{[v^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\texttt{true}, \alpha_1, \emptyset)}\ [\textbf{INF}\text{-}[\textbf{VALID}]]$$

$$\frac{\phi = \forall(FV(\alpha_1) - v^*) \cdot \neg\alpha_1 \quad \textbf{UNSAT}(\alpha_1 \wedge \alpha_2) \quad \phi \neq \texttt{false}}{[v^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\phi, \texttt{false}, \emptyset)}\ [\textbf{INF}\text{-}[\textbf{LHS}\text{-}\textbf{CONTRA}]]$$

$$\frac{\phi = \forall(FV(\alpha_1, \alpha_2) - v^*) \cdot (\neg\alpha_1 \vee \alpha_2) \quad \phi \neq \texttt{false}}{[v^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\phi, \alpha_1 \wedge \phi, \emptyset)}\ [\textbf{INF}\text{-}[\textbf{PRE}\text{-}\textbf{DERIVE}]]$$

$$\frac{}{[v^*, v_{rel}]\ \pi \vdash v_{rel}(u^*) \rightsquigarrow (\texttt{true}, \texttt{true}, \{\pi \rightarrow v_{rel}(u^*)\})}\ [\textbf{INF}\text{-}[\textbf{REL}\text{-}\textbf{DEFN}]]$$

$$\frac{[u^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\phi_1, \Delta_1, \emptyset) \quad [v^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\phi_2, \Delta_2, \emptyset)}{[v^*, v_{rel}]\ \alpha_1 \wedge v_{rel}(u^*) \vdash \alpha_2 \rightsquigarrow (\phi_2, \Delta_1 \wedge \Delta_2, \{v_{rel}(u^*) \rightarrow \phi_1\})}\ [\textbf{INF}\text{-}[\textbf{REL}\text{-}\textbf{OBLG}]]$$

Fig. 2: Pure Bi-Abduction Rules

• Rule [INF−[AND]] repeatedly breaks the conjunctive consequent into smaller components.

• Rules [INF−[UNSAT]] and [INF−[VALID]] infer `true` precondition whenever the entailment already succeeds. Specifically, rule [INF−[UNSAT]] applies when the antecedent α_1 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. Specifically, whenever a contradiction is detected between the antecedent α_1 and the consequent α_2, then rule [INF−[LHS−CONTRA]] applies and the precondition $\forall(FV(\alpha_1) - v^*) \cdot \neg\alpha_1$ 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 short-

hand notation for $v_1, .., v_n$[1]. 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 4.2, both aforementioned rules check that the inferred precondition is not equivalent to `false`.

• The last two rules [INF−[REL−DEFN]] and [INF−[REL−OBLG]] are meant to gather definitions and obligations, respectively, for the uninterpreted relation $v_{rel}(u^*)$. For simplicity, in rule [INF−[REL−OBLG]], we just formalize the case when there is only one uninterpreted relation in the antecedent.

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 form: $[v^*] \vdash \{\Delta_1\}\ c\ \{\phi_2, \Delta_2, \beta_2\}$ with three 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 uninterpreted relations. The selectivity criterion will help ensure that ϕ_2 and β_2 come from only the specified set of variables, namely $\{v^*\}$. If 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, and 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

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

inference rule for assignment (as $\wedge_u$ in $[\underline{\textbf{INF}-[\textbf{ASSIGN}]}]$) and in the inference rule for method invocation (as $*_{V\cup W}$ in $[\underline{\textbf{INF}-[\textbf{CALL}]}]$).

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

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

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

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

$$\frac{\begin{array}{c}[v^*,v^*_{rel}]\vdash\{\Phi_{pr}\wedge\bigwedge(u'{=}u)^*\}\ c\ \{\phi_2,\Delta_2,\beta_2\}\qquad [v^*,v^*_{rel}]\ \Delta_2\vdash\Phi_{po}\rightsquigarrow(\phi_3,\Delta_3,\beta_3)\\ \rho_1=infer\_pre(\beta_2\cup\beta_3)\qquad \rho_2=infer\_post(\beta_2\cup\beta_3)\\ \Phi^n_{pr}=\rho_1(\Phi_{pr}\wedge\phi_2\wedge\phi_3)\qquad \Phi^n_{po}=\rho_2(\Phi_{po}{*}\Delta_3)\end{array}}{\vdash t_0\ mn\ ((t\ u)^*)\ \texttt{infer}\ [v^*,v^*_{rel}]\ \Phi_{pr}\ \Phi_{po}\ \{c\}\rightsquigarrow\Phi^n_{pr}\ \Phi^n_{po}}\ [\underline{\textbf{INF}-[\textbf{METH}]}]$$

Fig. 3: Hoare Rules with Pure Bi-Abduction

For each method call, we must ensure that its precondition is satisfied, and then add the expected postcondition into its residual state, as illustrated in $[\underline{\textbf{INF}-[\textbf{CALL}]}]$. 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 $[\underline{\textbf{INF}-[\textbf{METH}]}]$. 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 entire set β of definitions and obligations for each uninterpreted relation. From this set we infer the pre- and post-relations via two steps described below. Take note that, given the entire set β, we retrieve the set of definitions and obligations for post-relations through functions $\texttt{def}_{\texttt{po}}$ and $\texttt{obl}_{\texttt{po}}$ respectively, while we use functions $\texttt{def}_{\texttt{pr}}$ and $\texttt{obl}_{\texttt{pr}}$ for pre-relations.

$$\begin{array}{l}\texttt{def}_{\texttt{po}}(\beta)=\{\pi^k_i{\rightarrow}v_{rel_i}(v^*_i)\mid(\pi^k_i{\rightarrow}v_{rel_i}@\texttt{po}(v^*_i))\in\beta\}\\ \texttt{obl}_{\texttt{po}}(\beta)=\{v_{rel_i}(v^*_i){\rightarrow}\alpha_j\mid(v_{rel_i}@\texttt{po}(v^*_i){\rightarrow}\alpha_j)\in\beta\}\\ \texttt{def}_{\texttt{pr}}(\beta)=\{\pi^k_i{\rightarrow}v_{rel_i}(v^*_i)\mid(\pi^k_i{\rightarrow}v_{rel_i}@\texttt{pr}(v^*_i))\in\beta\}\\ \texttt{obl}_{\texttt{pr}}(\beta)=\{v_{rel_i}(v^*_i){\rightarrow}\alpha_j\mid(v_{rel_i}@\texttt{pr}(v^*_i){\rightarrow}\alpha_j)\in\beta\}\end{array}$$

• In order to infer post-relations, function *infer_post* applies a least fixed point analysis to the set of gathered relational definitions $\texttt{def}_{\texttt{po}}(\beta)$. 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 [18] and FIXBAG [16], respectively. The call to the fixed point analysis is denoted below as $\texttt{LFP}(\texttt{def}_{\texttt{po}}(\beta))$, It takes as inputs the set of relational definitions, while returning a set of closed form constraints of the form $\alpha_i \rightarrow v_{rel_i}(v_i^*)$, where each constraint corresponds to an uninterpreted relation v_{rel_i}. Given that we aim at inferring strongest post-relations, we further consider each post-relation $v_{rel_i}(v_i^*)$ to be equal to α_i. Finally, *infer_post* returns a set of substitutions, where each uninterpreted 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.

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

• For pre-relations, our goal is to infer the weakest ones. Hence, for each pre-relation, we first calculate the conjunction of all its obligations from $\texttt{obl}_{\texttt{pr}}(\beta)$ to obtain sufficient preconditions $\texttt{pre\_fst}_{\texttt{i}}$ for base cases. To capture the precondition for a 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, which can be achieved via a top-down fixed point analysis [19]. Obviously, the parameters of an arbitrary call must also satisfy relevant relational obligations (e.g. $\texttt{pre\_fst}_{\texttt{i}}$). Finally, the approximation α_i for each relation $v_{rel_i}(v_i^*)$ must satisfy the precondition for the first call ($\texttt{pre\_fst}_{\texttt{i}}$), for an arbitrary recursive call ($\texttt{pre\_rec}_{\texttt{i}}$) and the invariant (e.g. $\texttt{a} \geq \texttt{0} \wedge \texttt{b} \geq \texttt{0}$ in Sec 4.4). The last step is to check the quality of candidate preconditions in order to keep the ones that satisfy not only the obligations but also definitions of each relation.

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

With the help of functions *infer_pre* and *infer_post*, we can finally 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 uninterpreted relations that are to be inferred, whereas ρ_1 and ρ_2 represent the substitutions obtained for pre- and post-relations, respectively.

7 Enhancing Predicates with Pure Properties

Since the user may encounter various kinds of inductive spatial predicates (from a shape analysis step), such as linked lists, doubly-linked lists, trees, etc and there may be different pure properties to enrich shape predicates such as size, height, sum, head, min/max, set of values/addresses (and their combinations), we need to use a predicate extension mechanism to systematically incorporate pure properties into heap predicates.

Our mechanism is generic in the sense that each specified property can be applied to a broad range of recursive data structures, whose underlying structures can be quite different. For example, while linked-list segment (lseg) is built upon node, doubly-linked list (dll) and tree (tree) have a different underlying structure node2:

$$\begin{array}{l}\texttt{pred lseg}\langle\texttt{root, p}\rangle \equiv \texttt{root=p} \vee \exists\, \texttt{q}\cdot(\texttt{root}\mapsto\texttt{node}\langle\_, \texttt{q}\rangle * \texttt{lseg}\langle\texttt{q, p}\rangle)\\ \texttt{pred dll}\langle\texttt{root, p}\rangle \equiv \texttt{root=null} \vee \exists\, \texttt{q}\cdot(\texttt{root}\mapsto\texttt{node2}\langle\_, \texttt{p, q}\rangle * \texttt{dll}\langle\texttt{q, root}\rangle)\\ \texttt{pred tree}\langle\texttt{root}\rangle \equiv \texttt{root=null} \vee \exists\, \texttt{q, r}\cdot(\texttt{root}\mapsto\texttt{node2}\langle\_, \texttt{q, r}\rangle * \texttt{tree}\langle\texttt{q}\rangle * \texttt{tree}\langle\texttt{r}\rangle)\\ \texttt{where}\qquad \texttt{data node2}\,\{\texttt{int val; node2 left; node2 right;}\}\end{array}$$

To enrich pure properties for various kinds of shape predicates, we first define pure properties of data structures in the form of parameterized inductive definitions such as:

$$\begin{array}{lll}\texttt{prop\_defn HEAD}[\texttt{@V}]\langle\texttt{v}\rangle & \equiv & \texttt{v=V}\\ \texttt{prop\_defn SIZE}[\texttt{@R}]\langle\texttt{n}\rangle & \equiv & \texttt{n=0} \vee \texttt{SIZE}\langle\texttt{R, m}\rangle\wedge\texttt{n=1+m}\\ \texttt{prop\_defn HEIGHT}[\texttt{@R}]\langle\texttt{n}\rangle & \equiv & \texttt{n=0} \vee \texttt{HEIGHT}\langle\texttt{R, m}\rangle\wedge\texttt{n=1+max(m)}\\ \texttt{prop\_defn SUM}[\texttt{@V, @R}]\langle\texttt{s}\rangle & \equiv & \texttt{s=0} \vee \texttt{SUM}\langle\texttt{R, r}\rangle\wedge\texttt{s=V+r}\\ \texttt{prop\_defn SET}[\texttt{@V, @R}]\langle\texttt{S}\rangle & \equiv & \texttt{S=\{\}} \vee \texttt{SET}\langle\texttt{R, S}_2\rangle\wedge\texttt{S=\{V\}}\sqcup\texttt{S}_2\\ \texttt{prop\_defn SETADDR}[\texttt{@R}]\langle\texttt{S}\rangle & \equiv & \texttt{S=\{\}} \vee \texttt{SETADDR}\langle\texttt{R, S}_2\rangle\wedge\texttt{S=\{root\}}\sqcup\texttt{S}_2\\ \texttt{prop\_defn MINP}[\texttt{@V, @R}]\langle\texttt{mi}\rangle & \equiv & \texttt{mi=min(V)} \vee \texttt{MINP}\langle\texttt{R, mi}_2\rangle\wedge\texttt{mi=min(V, mi}_2\texttt{)}\\ \texttt{prop\_defn MAXP}[\texttt{@V, @R}]\langle\texttt{mx}\rangle & \equiv & \texttt{mx=max(V)} \vee \texttt{MAXP}\langle\texttt{R, mx}_2\rangle\wedge\texttt{mx=max(V, mx}_2\texttt{)},\end{array}$$

where V, R are values extracted from parameters @V, @R respectively. For instance, to determine if n is the size of some data structure whose recursive pointer is annotated as @REC, SIZE[@REC]⟨n⟩ would check the satisfiability of the base case (i.e. n=0) and the inductive case (via recursive pointer REC). Each occurrence of the annotated value, e.g. V, is either a single occurrence or it denotes multiple occurrences, depending on its context of use. As an example, the V value in HEAD definition 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 MINP, MAXP property definition. The presence of associative operators would allow us to combine multiple occurrences into a single result. For values that denote recursive pointers, such as R in the SET definition, the occurrences of their values are determined through constraints on their defining properties, such as $\texttt{S}_2$ from $\texttt{SET}\langle\texttt{R}, \texttt{S}_2\rangle$. Here, $\texttt{S}_2$ denotes multiple occurrences as we can identify an associative operator $\sqcup$ from $\texttt{S=\{V\}}\sqcup\texttt{S}_2$ to combine them.

Using such definition, one can use the following command to incorporate size property directly to a linked-list predicate definition:

$$\begin{array}{l}\texttt{pred llN}\langle\texttt{root, n}\rangle = \texttt{extend ll}\langle\texttt{root}\rangle\ \texttt{with SIZE}[\texttt{@REC}]\langle\texttt{n}\rangle\\ \texttt{data node}\,\{\texttt{int val@VAL; node next@REC;}\}\end{array}$$

Here, the annotations @VAL, @REC are hardwired to two fields of underlying heap structure node. Based on the commands and the definition of pure properties, our system first constructs an entry (in a dictionary) for each targeting predicate. For example, there is one entry $(\texttt{llN}\langle\texttt{root, n}\rangle, (\texttt{F1, F2, Base, Rec}))$, where list of all value field annotations F1=[], list of all recursive pointer annotations F2=[@REC], base case $\texttt{Base}=\backslash[\_] \rightarrow \texttt{n=0}$ and inductive case $\texttt{Rec}=\backslash[\texttt{m}_{\texttt{REC}}] \rightarrow \texttt{n=m}_{\texttt{REC}}\texttt{+1}$ ($\texttt{m}_{\texttt{REC}}$ is the size property of corresponding

recursive pointer REC of the linked-list). Using the dictionary, we can transform the base case and inductive case of original spatial predicate ll as follows:

$$\begin{array}{r}\texttt{root=null}\ \#\texttt{Dict}\rightsquigarrow\texttt{root=null}\wedge\texttt{n=0}\\ \exists\texttt{q}\cdot(\texttt{root}\mapsto\texttt{node}\langle\_,\texttt{q}\rangle*\texttt{ll}\langle\texttt{q}\rangle)\ \#\texttt{Dict}\rightsquigarrow\exists\texttt{q},\texttt{m}\cdot(\texttt{root}\mapsto\texttt{node}\langle\_,\texttt{q}\rangle*\texttt{llN}\langle\texttt{q},\texttt{m}\rangle\wedge\texttt{n=m+1})\end{array}$$

Finally, we can synthesize llN predicate as previously shown in Sec 2.

We stress that our technique aims at a systematic way to extend shape predicates with pure properties. In fact, it can be applied for a broad range of recursive data structures. For illustration, let us consider another command to derive the predicate llHM, that captures the head v and the minimum element mi of a singly linked list pointed by root.

$$\begin{array}{c}\texttt{pred llHM}\langle\texttt{root},\texttt{v},\texttt{mi}\rangle=\texttt{extend ll}\langle\texttt{root}\rangle\ \texttt{with}\\ \texttt{HEAD}[@\texttt{VAL}]\langle\texttt{v}\rangle,\texttt{MINP}[@\texttt{VAL},@\texttt{REC}]\langle\texttt{mi}\rangle.\end{array}$$

Since these pure properties (HEAD and MINP) require a non-null heap structure, our system will first derive the predicate llnn describing a non-null singly-linked list:

$$\texttt{pred llnn}\langle\texttt{root}\rangle\equiv\exists\texttt{v}_1\cdot(\texttt{root}\mapsto\texttt{node}\langle\texttt{v}_1,\texttt{null}\rangle)\vee\exists\texttt{v}_1,\texttt{q}\cdot(\texttt{root}\mapsto\texttt{node}\langle\texttt{v}_1,\texttt{q}\rangle*\texttt{llnn}\langle\texttt{q}\rangle)$$

Next, we will have one entry in the dictionary Dict corresponding to the targeting predicate llHM:

$$\begin{array}{l}(\texttt{llHM}\langle\texttt{root},\texttt{v},\texttt{mi}\rangle,([@\texttt{VAL}],[@\texttt{REC}\#],\texttt{Base},\texttt{Rec}))\\ \texttt{where Base}=\setminus[\texttt{v}_1,\_]\rightarrow\texttt{v=v}_1\wedge\texttt{mi=v}_1\\ \texttt{and Rec}=\setminus[\texttt{v}_1,\texttt{m}_1\#]\rightarrow\texttt{v=v}_1\wedge\texttt{mi=min}(\texttt{v}_1,\texttt{fold}(\texttt{min},\texttt{m}_1\#,\infty))\end{array}$$

We use the # marker to indicate the case where the annotation has multiple occurrences. For example, [@Rec#] is a list of all recursive pointer annotations and $\texttt{m}_1\#$ is the list of MINP properties extracted from [@Rec#]. Since $\texttt{m}\#_1$ is a list of values, we use a higher-order meta-operation $\texttt{fold}(\texttt{min},\texttt{m}\#_1,\infty)$[2] to obtain the list's minimum element. 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 judgment is used to derive the definitions of new predicates. Our system will apply simultaneously the two properties HEAD and MINP to the base and inductive cases of the non-null singly-linked list definition, which results in two derivation judgments:

$$\begin{array}{r}\exists\texttt{v}_1\cdot(\texttt{root}\mapsto\texttt{node}\langle\texttt{v}_1,\texttt{null}\rangle)\ \#\texttt{Dict}\rightsquigarrow\exists\texttt{v}_1\cdot(\texttt{root}\mapsto\texttt{node}\langle\texttt{v}_1,\texttt{null}\rangle\wedge\texttt{mi=v}_1\wedge\texttt{v=v}_1)\\ \exists\texttt{v}_1,\texttt{q}\cdot(\texttt{root}\mapsto\texttt{node}\langle\texttt{v}_1,\texttt{q}\rangle*\texttt{llnn}\langle\texttt{q}\rangle)\ \#\texttt{Dict}\rightsquigarrow\exists\texttt{v}_1,\texttt{q},\texttt{v}_2,\texttt{mi}_2\cdot(\texttt{root}\mapsto\texttt{node}\langle\texttt{v}_1,\texttt{q}\rangle*\\ \texttt{llHM}\langle\texttt{q},\texttt{v}_2,\texttt{mi}_2\rangle\wedge\texttt{mi=min}(\texttt{v}_1,\texttt{mi}_2)\wedge\texttt{v=v}_1)\end{array}$$

Lastly, we obtain the following new predicate definition for llHM:

$$\begin{array}{l}\texttt{pred llHM}\langle\texttt{root},\texttt{v},\texttt{mi}\rangle\equiv\exists\texttt{v}_1\cdot(\texttt{root}\mapsto\texttt{node}\langle\texttt{v}_1,\texttt{null}\rangle\wedge\texttt{mi=v}_1\wedge\texttt{v=v}_1)\vee\\ \exists\texttt{v}_1,\texttt{q},\texttt{v}_2,\texttt{mi}_2\cdot(\texttt{root}\mapsto\texttt{node}\langle\texttt{v}_1,\texttt{q}\rangle*\texttt{llHM}\langle\texttt{q},\texttt{v}_2,\texttt{mi}_2\rangle\wedge\texttt{mi=min}(\texttt{v}_1,\texttt{mi}_2)\wedge\texttt{v=v}_1).\end{array}$$

While property extensions are user customizable, their use within our pure inference sub-system can be completely automated, as we can automatically construct predicate

[2] Similar to a List.fold_right reduction operation in OCaml

derivation commands and systematically apply them after shape analysis. We then 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.

8 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 then 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 programs.

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 2, that can be found in Appendix A. Though codes for these examples are not so complicated, they illustrate the treatment of recursion (thus, the interprocedural aspect). Therefore, the preconditions and postconditions derived can be quite intricate and would require considerable human efforts if they were constructed manually.

Ex. 2. A method where the content of its data structure is helpful to ensure its functional correctness

```
1  node del_val(node x,int a){
2    if (x == null) return x;
3    else if (x.val == a) {
4         node tmp = x.next;
5         free(x);
6         return tmp; }
7      else {
8         x.next =
9           del_val(x.next, a);
10        return x; } }
```

One interesting thing to note is that each example may require different pure properties for its correctness to be captured and verified. Using our pure bi-abduction technique, we can derive more expressive specifications, which can help ensure a higher-level correctness of programs. For instance, the size property is not enough to capture functional correctness of `del_val` method, whose source code is given in Ex. 2. Method `del_val` deletes the first node with value `a` from the linked-list pointed by `x`. Since the behavior of this method depends on the content of its list, SPECINFER need to derive `llNB` predicate that also captures a bag `B` of values stored in the list:

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

Finally, our tool infer the following specification, that guarantees the functional correctness of `del_val` method:

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

where `res` denotes the method's result.

Medium Examples. We tested our tool on a set of challenging programs that use a variety of data structures. The results are shown in Table 1 (the first eight rows), 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 start with shape specifications that are from the prior shape analysis step. The number of procedures with valid specifications (valid Hoare triples) is reported in the `V#` 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` columns. While phase 2 only focuses on quantitative properties such as size (number of nodes) and height (for trees), phase 3 aims at other functional properties. We also measure the time (in seconds) taken for verification with selective inference, in the `Time` column.

In addition to illustrating the applicability of SPECINFER in dealing with different data structures and pure properties, Table 1 reaffirms the need of pure properties for capturing program correctness. More specifically, for procedures that SPECINFER cannot infer any valid specifications, we do construct the specifications manually. However due to the restriction of properties the resulting specification can capture, we fail to do so for these procedures. For illustration, we cannot construct any valid specification for about 18% of procedures in phase 1 (using shape domain only). Even in phase 2, there is still one example, `delete_max` method in `Heaps` test, for which we cannot obtain any valid specification. This method is used to delete the root of a heap tree, thus it requires the information of the maximum element.

			Shape		Shape + Quan				Shape + Quan + Func			
Program	LOC	P#	V#	%	Add.Properties	V#	%	Time	Add.Properties	V#	%	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
`schedule`	512	19	14	75	Size	19	100	6.86				
`schedule2`	474	14	6	43	Size	14	100	10.58				
`pcidriver`	1036	29	29	100	Size	29	100	17.72				

Table 1. Specification Inference with Pure Properties for a Variety of Data Structures

Larger Examples. The last three rows from Table 1 demonstrates the applicability of SPECINFER on larger programs. The first two programs used to perform process scheduling are adopted from SIR test suites [7] while the last one is `pci_driver.c` file from Linux Device Driver. Note that for this case study we enrich shape specifications with size property only. For Linux file, although it is sufficient to use only shape prop-

erty to prove memory-safety, size property can still be useful for termination proving aspect.

9 Related Works

Specification inference makes program verification more practical by minimizing on the need for manual provision of specifications. It can also be used to support formal software documentation. 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. The footprint analysis described in [4] infers descriptions of data structures without requiring a given precondition. Furthermore, in [3], 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 [3] is a tool implementing a compositional shape analysis based on bi-abduction, which was used to check memory safety of large open source codebases [6]. A recent work [23] 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 mechanism. Our aim is to provide a systematic machinery for deriving formal specifications with more precise correctness properties.

A closely related research direction concerns the inference of both shape and numerical properties. In [12], 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 [13] is a tool for reasoning about a combination of list reasoning and arithmetic by using two separate analyses. The arithmetic support added by Thor includes stack-based integers, integers in the heap, lengths of lists. However, these current works are 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. A recent work [20] focuses 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 bi-abduction with pure properties. Our paper addresses these issues by designing a new pure bi-abduction entailment procedure, together with the handling of uninterpreted functions and relations. To utilize shape analyses' results, we also propose a predicate extension mechanism for systematically enhancing predicates with new pure properties. Another recent work [9] aims to automatically construct verification tools that implement various input proof rules for reachability and termination properties in

the form of Horn(-like) clauses. Also, on the type system side, the authors of [22,10,11] require templates in order to infer dependent types precise enough to prove a variety of safety properties such as safety of array accesses. However, in both of these works, mutable data structures are not supported. Compared to these works, our proposal can 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.

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. 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.
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. 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.
6. D. Distefano. Attacking large industrial code with bi-abductive inference. In *FMICS*, pages 1–8, 2009.
7. 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.
8. 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.
9. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.
10. M. Kawaguchi, P. M. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, pages 304–315, 2009.
11. M. Kawaguchi, P. M. Rondon, and R. Jhala. Dsolve: Safety verification via liquid types. In *CAV*, pages 123–126, 2010.
12. S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, pages 419–436, 2007.
13. 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.
14. 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.
15. P. W. O'Hearn, J. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In *CSL*, Paris, France, Sept. 2001.
16. 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.
17. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.
18. C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *ASIAN*, pages 331–345.
19. 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.

20. S. Qin, C. Luo, W.-N. Chin, and G. He. Automatically refining partial specifications for program verification. In *FM*, pages 369–385, 2011.
21. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, pages 55–74, 2002.
22. P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.
23. F. Vogels, B. Jacobs, F. Piessens, and J. Smans. Annotation inference for separation logic based verifiers. In *FMOODS/FORTE*, pages 319–333, 2011.

A Pre-/Post-Conditions Inferred for Examples

In Table 2, we summarize inferred pre/post specifications for selected list-based procedures. Each procedure can take as its inputs singly linked lists (pointed by x or y), i_th index and an integer value a. We start with specifications that are from the prior shape analysis step, and enrich the heap part with size property and multi-set/bag property. The inferred constraints are then highlighted using grayed background.

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

Table 2. Pre-/Post-Conditions Inferred for Selected List-Based Examples

B Soundness of Inference

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

Lemma 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 2 (Soundness of Hoare Logic with Specification 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 1 and 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 [5]. For Lemma 1, given that the underlying entailment procedure without inference capabilities is sound, the collection of suitable preconditions in [INF−[AND]], [INF−[PRE−DERIVE]] and [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 2 assumes preconditions of recursive methods are inferred using pre-relations. Both lemmas can be shown to hold by the following inductive proofs over their respective rules.

Proof. We first present two auxiliary propositions: Partial Substitution Law for Assertions and the Monotonicity rule [21]. 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 $FV(p) \subseteq \{v_1, \ldots, v_n\}$, 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 $FV(p)$ 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 $\{v'_1, \ldots, v'_k\} = FV(p) - \{v_1, \ldots, v_n\}$.

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 $(FV(p) - \{v_1, \ldots, v_n\}) \cup FV(e_1) \cup \ldots \cup FV(e_n) \subseteq \texttt{dom}\, s$, 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) denotes 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 $f\ x$.

Next, we will show our proof for the two mentioned lemmas:

Lemma 1. Using these auxiliary propositions, Lemma 1 can be proven by induction on entailment rules from Figure 2:

- Case $[\underline{\textbf{INF}\text{–}[\textbf{AND}]}]$:
  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:
  $[\,]\ \pi_1 \wedge \pi_1 \wedge \phi_2 \wedge \phi_3 \wedge \bigwedge_{\alpha \in \beta} \alpha \vdash \pi_2 \wedge \pi_3 \rightsquigarrow (\texttt{true}, \Delta_2 \wedge \Delta_3, \emptyset)$
  $\Leftrightarrow$ $[\,]\ \pi_1 \wedge \phi_2 \wedge \phi_3 \wedge \bigwedge_{\alpha \in \beta} \alpha \vdash \pi_2 \wedge \pi_3 \rightsquigarrow (\texttt{true}, \Delta_2 \wedge \Delta_3, \emptyset)$ with $\beta = \beta_2 \cup \beta_3$.
  Thus, Lemma 1 holds for: $[v^*]\ \pi_1 \vdash \pi_2 \wedge \pi_3 \rightsquigarrow (\phi_2 \wedge \phi_3, \Delta_2 \wedge \Delta_3, \beta_2 \cup \beta_3)$.
- Case $[\underline{\textbf{INF}\text{–}[\textbf{UNSAT}]}]$: This case holds trivially.
- Case $[\underline{\textbf{INF}\text{–}[\textbf{VALID}]}]$: This case holds trivially.
- Case $[\underline{\textbf{INF}\text{–}[\textbf{LHS}\text{–}\textbf{CONTRA}]}]$:
  We need to show that:
  $[v^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\phi, \texttt{false}, \emptyset) \wedge \phi = \forall(FV(\alpha_1) - v^*) \cdot \neg\alpha_1$
  $\Rightarrow$ $[\,]\ \alpha_1 \wedge \phi \vdash \alpha_2 \rightsquigarrow (\texttt{true}, \texttt{false}, \emptyset)$
  Suppose that the negation operator and quantifier elimination algorithms supported by provers are all sound, we need to show that with $\phi = \forall(FV(\alpha_1) - v^*) \cdot \neg\alpha_1$, we still have the contradiction between ϕ and α_1.
  In fact,
  - Case $\{v^*\} = \emptyset \Rightarrow \phi \equiv \texttt{false}$
  - Case $\{v^*\} = FV(\alpha_1) \Rightarrow \quad \phi \equiv \neg\alpha_1$
    $\Rightarrow \phi \wedge \alpha_1 \equiv \neg\alpha_1 \wedge \alpha_1 \equiv \texttt{false}$
  - Otherwise:
    * if we can sufficiently express $\neg\alpha_1$ using v^*, Lemma 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 $[\underline{\textbf{INF}\text{–}[\textbf{PRE}\text{–}\textbf{DERIVE}]}]$:
  We need to show that:
  $[v^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\phi, \alpha_1 \wedge \phi, \emptyset) \wedge \phi = \forall(FV(\alpha_1, \alpha_2) - v^*) \cdot (\neg\alpha_1 \vee \alpha_2)$

$\Rightarrow \quad [\,]\ \alpha_1 \wedge \phi \vdash \alpha_2 \rightsquigarrow (\texttt{true}\,, \alpha_1 \wedge \phi, \emptyset)$

Indeed,

- Case $\{v^*\}{=}\emptyset \Rightarrow \phi \equiv \texttt{false}$
- Case $\{v^*\}{=}FV(\alpha_1, \alpha_2) \Rightarrow \quad \phi \equiv \neg\alpha_1 \vee \alpha_2$
  $\Rightarrow \phi \wedge \alpha_1 \equiv \alpha_2 \wedge \alpha_1$
- Otherwise:
  - if we can sufficiently express $\neg\alpha_1 \vee \alpha_2$ using v^*, Lemma 1 is sound.
  - otherwise, in the worst case, we can still infer a precondition as $\forall(FV(\alpha_1, \alpha_2){-}v^*){\cdot}\neg\alpha_1$. Thus, Lemma 1 holds.

– Case [INF−[REL−DEFN]]: This case holds trivially.
– Case [INF−[REL−OBLG]]: This follows from the soundness of rule [INF−[PRE−DERIVE]].

Lemma 2. Lemma 2 can be proven by induction on Hoare rules with pure bi-abduction from Figure 3:

– Case [INF−[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:

$$[\,] \vdash \{\Delta \wedge \phi_2 \wedge \textstyle\bigwedge_{\alpha \in \beta_2} \alpha\}\ e_1\ \{\texttt{true}\,, \Delta_2, \emptyset\} \text{ and}$$

$$[\,] \vdash \{\Delta_2 \wedge \phi_3 \wedge \textstyle\bigwedge_{\alpha \in \beta_3} \alpha\}\ e_2\ \{\texttt{true}\,, \Delta_3, \emptyset\} \tag{1}$$

The Frame rule ensures that the following holds:

$$[\,] \vdash \{\Delta \wedge \phi_2 \wedge \phi_3 \wedge \textstyle\bigwedge_{\alpha \in \beta} \alpha\}\ e_1\ \{\texttt{true}\,, \Delta_2 * \phi_3 \wedge \textstyle\bigwedge_{\alpha \in \beta_3} \alpha, \emptyset\} \tag{2}$$

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 2 holds for: $[v^*] \vdash \{\Delta\}\ e_1; e_2\ \{\phi_2 \wedge \phi_3, \Delta_3, \beta_2 \cup \beta_3\}$.

– Case [INF−[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:

$$[\,] \vdash \{\Delta \wedge \phi_2 \wedge w' \wedge \textstyle\bigwedge_{\alpha \in \beta_2} \alpha\}\ e_1\ \{\texttt{true}\,, \Delta_2, \emptyset\} \tag{3}$$

$$[\,] \vdash \{\Delta \wedge \phi_3 \wedge \neg w' \wedge \textstyle\bigwedge_{\alpha \in \beta_3} \alpha\}\ e_2\ \{\texttt{true}\,, \Delta_3, \emptyset\} \tag{4}$$

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 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 $[\underline{\textbf{INF}-[\textbf{ASSIGN}]}]$:
Suppose $[v^*] \vdash \{\Delta\}\ e\ \{\phi_2, \Delta_2, \beta_2\}$.
By the inductive hypothesis, we have:
$$[\,] \vdash \{\Delta \wedge \phi_2 \wedge \textstyle\bigwedge_{\alpha \in \beta_2} \alpha\}\ e\ \{\texttt{true}\,, \Delta_2, \emptyset\}.$$
With $\Delta_3 = \exists \texttt{res} \cdot (\Delta_2 \wedge_u u' = \texttt{res})$, we obtain (similarly in [5]):
$$[\,] \vdash \{\Delta \wedge \phi_2 \wedge \textstyle\bigwedge_{\alpha \in \beta_2} \alpha\}\ u := e\ \{\texttt{true}\,, \Delta_3, \emptyset\}.$$
Thus, Lemma 2 holds for:
$$[v^*] \vdash \{\Delta\}\ u := e\ \{\phi_2, \Delta_3, \beta_2\}.$$
– Case $[\underline{\textbf{INF}-[\textbf{CALL}]}]$:
Suppose we have:
$$[v^*]\ \Delta \vdash \rho(\Phi_{pr}) \rightsquigarrow (\phi_2, \Delta_2, \beta_2) \text{ with } \rho = [v'_k/v_k]_{k=1}^{n} \text{ and}$$
$$t_0\ mn\ (\texttt{ref}\ (t_i\ v_i)_{i=1}^{m-1}, (t_j\ v_j)_{j=m}^{n})\ \ \Phi_{pr}\ \ \Phi_{po}\ \{c\} \in Prog.$$
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 [5], we also have:
$$[\,] \vdash \{\Delta \wedge \phi_2 \wedge \textstyle\bigwedge_{\alpha \in \beta_2} \alpha\}\ mn(v_1, \ldots, v_{m-1}, v_m, \ldots v_n)\ \{\texttt{true}\,, \Delta_3, \emptyset\}$$
with $W = \{v_1, .., v_{m-1}\}$, $V = \{v_m, .., v_n\}$ and $\Delta_3 = (\Delta_2 \wedge nochange(V))\ *_{V \cup W}\ \Phi_{po}$.
Thus, Lemma 2 holds for: $[v^*] \vdash \{\Delta\}\ mn(v_1, \ldots, v_{m-1}, v_m, \ldots v_n)\ \{\phi_2, \Delta_3, \beta_2\}$ □

Since both Lemma 1 and 2 have been proven to be sound, the soundness of rule $[\underline{\textbf{INF}-[\textbf{METH}]}]$ 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 [18,16] while the top-down fixpoint procedure (used in *infer_pre*) is inspired by [19].