Bi-Abduction with Pure Properties for Specification Inference

Minh-Thai Trinh[1], Quang Loc Le[1], Cristina David[2], and Wei-Ngan Chin[1]

[1] National University of Singapore
{trinhmt,locle,chinwn}@comp.nus.edu.sg
[2] University of Oxford
cristina.david@gmail.com

Abstract. Separation logic is a state-of-the-art logic for dealing with the program 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 minimum/maximum value, 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. The power of our new bi-abductive entailment procedure is significantly enhanced by its collection of proof obligations over *uninterpreted relations (functions)*. 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.

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

1 Introduction

One of the challenging areas for software verification concerns programs using heap-based data structures. To prove the correctness of such programs, in the last decade, research methodologies based on separation logic have offered good solutions [1,13,3].

Separation logic [20,14], an extension of Hoare logic, is a state-of-the-art logic for dealing with the program 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 [1,13] based on separation logic have strived towards automated modular verification against user-supplied specifications. However, manually writing specifications is a tedious and error-prone engineering process. Thus, more recent separation

C.-c. Shan (Ed.): APLAS 2013, LNCS 8301, pp. 107–123, 2013.

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] 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 recursive function in Ex. 1 that zips two lists of integers into a single one. To reduce the performance overhead of redundant null-checking, in the `zip` method there is no null-checking for `y`. As a result, the field access `y.next` at 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 becomes 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, using 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 the problem of inferring specifications with both heap and pure information. To achieve this, we have designed a new bi-abductive entailment procedure. Its power will be significantly enhanced by the collection of proof obligations over uninterpreted relations (functions).

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.

Contributions. Our contributions include the following:

- We design a new bi-abductive entailment procedure for inferring pure information for specifications of heap-manipulating programs. 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 via fixpoint analyses (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 (Section 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 [3,6], we could only obtain the following shape specification:

$$\begin{array}{l} \texttt{requires}\ \texttt{ll}\langle\texttt{x}\rangle * \texttt{ll}\langle\texttt{y}\rangle \\ \texttt{ensures}\ \ \texttt{ll}\langle\texttt{res}\rangle; \end{array}$$

$$\texttt{where}\qquad \texttt{pred}\ \texttt{ll}\langle\texttt{root}\rangle \equiv (\texttt{root}{=}\texttt{null}) \vee \exists\, \texttt{q}\cdot(\texttt{root}{\mapsto}\texttt{node}\langle\_,\texttt{q}\rangle * \texttt{ll}\langle\texttt{q}\rangle).$$

Although this specification cannot ensure memory safety for the y.next field access (at lines 7 and 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. For instance, the 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, while the second one separates the list into two parts: the head $\texttt{root}{\mapsto}\texttt{node}\langle\_,\texttt{q}\rangle$, where $\mapsto$ is 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 speaking, we cannot obtain any valid pre/post specification (valid Hoare triple) for the zip method by using only the shape domain. To prove memory safety, the specification must also capture the size of each list.

Using predicate extension mechanism, we first inject the size property (captured by n) into the ll predicate in order to derive the llN predicate as follows:

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

With the new llN predicate, we could then strengthen the specification 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. Their purpose is 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.

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

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

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

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

Using suitable fix-point analysis techniques, we can synthesize the approximations for these unknowns, which would add a pre-condition $\mathtt{a{\leq}b}$ to guarantee memory safety. Specifically, we have $\mathtt{P(a,b)} \equiv \mathtt{a{\leq}b}$, $\mathtt{Q(r,a,b)} \equiv \mathtt{r{=}a{+}b}$ and a new specification:

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

Program Termination. 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 [16] to support termination proofs. For this task, we would introduce an uninterpreted function $\mathtt{F(a,b)}$, as a possible measure, via the following termination-based specification that is synthesized right after size inference. Note that size inference is crucial for proving not only program safety but also program termination.

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

Similarly, applying our pure bi-abduction technique, we can derive the following proof obligations whose satisfaction would guarantee program termination.

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

Using suitable fixpoint analyses, we can synthesize $\mathtt{F(a,b)} \equiv \mathtt{a{-}1}$, 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 reasons, 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 whenever the method is called in a state satisfying precondition Φ_{pr} and the method terminates, the resulting state will satisfy the corresponding postcondition Φ_{po}. The specification inference process can be enabled 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.

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

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

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 free 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 (dis)equality γ, 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 postcondition resp. This information will be later used to synthesize the approximation for each uninterpreted relation in Sec. 6. The relational definitions and obligations defined in Sec. 4.4 are denoted as $\pi{\rightarrow}v_{rel}(v^*)$ and $v_{rel}(v^*){\rightarrow}\alpha$ resp. Lastly, Δ denotes a composite formula that can be normalized into the Φ form, while ϕ represents a pure formula.

4 Principles of Pure Bi-Abduction

Initial works [1,13] are typically based on an entailment system of the form $\Delta_1 \vdash \Delta_2 \leadsto \Delta_r$, 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 $\Delta_1 \vdash \Delta_2 \leadsto (\Delta_p, \Delta_r)$, whereby a precondition Δ_p, the condition for the entailment proving to succeed, may be inferred.

In this paper, we propose pure bi-abduction technique to infer pure information for pre/post specifications. To better exploit the expressiveness of separation logic, we integrate inference mechanisms directly into it and propose to use an entailment system of the following form $[v_1, .., v_n]\ \Delta_1 \vdash \Delta_2 \leadsto (\phi_p, \Delta_r, \beta_c)$. 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 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 these second-order variables. This approach is critical for capturing inductive definitions that can be refined via fix-point analyses.

We first highlight key principles employed by pure bi-abduction with examples. Later in Sec. 5, we shall present the formalization for our proposed technique.

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:

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

Predicate $\mathtt{llN}\langle\mathtt{x}, \mathtt{n}\rangle$ by itself does not entail a non-empty node. For the entailment proving 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 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 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 aim 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 the antecedent Δ_1. Although we disallow $\mathtt{false}$ to be inferred, we allow above precondition if it is not equivalent to $\mathtt{false}$. 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 should be as weak as possible, while post-relations should be as strong as possible. Our inference mechanism respects this principle, and would use it to derive the weakest pre-relations and strongest post-relations, where possible.

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 unknown relation is in the consequent instead.

Relational Obligations. They are useful in two ways. For pre-relations, they act as initial preconditions for (recursive) methods. For post-relations, they denote proof obligations that post-relations must also satisfy. We will check these obligations after we have synthesized post-relations.

As an example, consider the entailment extracted from the motivating example: $[\mathtt{P}]$ $\mathtt{a}{\geq}1{\wedge}\mathtt{b}{=}0{\wedge}\mathtt{P}(\mathtt{a},\mathtt{b}) \vdash \mathtt{b}{\neq}0$. We infer $\mathtt{P}(\mathtt{a},\mathtt{b}){\rightarrow}\mathtt{a}{\leq}0{\vee}\mathtt{b}{\neq}0$, which will denote an initial precondition 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 contradict 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. After gathering the relational definitions (both base and inductive cases), we would apply a least fixpoint procedure [17] to discover suitable closed-form definitions for post-relations. For pre-relations, while it may be possible to compute a 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. $\mathtt{REC}_\mathtt{a},\mathtt{REC}_\mathtt{b}$) to those of the first one (e.g. $\mathtt{a},\mathtt{b}$) using top-down fixpoint [18]. For example, a recursive invariant $\mathtt{rec\_inv}$ for $\mathtt{zip}$ method is $\mathtt{REC}_\mathtt{a}{\geq}0{\wedge}\mathtt{a}{\geq}1{+}\mathtt{REC}_\mathtt{a}{\wedge}\mathtt{REC}_\mathtt{a}{+}\mathtt{b}{=}\mathtt{REC}_\mathtt{b}{+}\mathtt{a}$. Next, since parameters of an arbitrary call must also satisfy relevant relational obligations, the precondition $\mathtt{pre\_rec}$ is then $\forall\mathtt{REC}_\mathtt{a},\mathtt{REC}_\mathtt{b}\cdot\ \mathtt{rec\_inv}{\rightarrow}\mathtt{pre\_fst}(\mathtt{REC}_\mathtt{a},\mathtt{REC}_\mathtt{b})$, where $\mathtt{pre\_fst}(\mathtt{a},\mathtt{b}){=}\mathtt{a}{\leq}0{\vee}\mathtt{b}{\neq}0$ is the initial condition. 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 analyses, whose important operators (i.e. narrowing) are supported in restricted domains, and is sufficient for all practical examples evaluated.

5 Formalization of Pure Bi-Abduction

Recall that our bi-abductive entailment proving procedure has the following form:

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

This new entailment procedure serves two key roles:

- For our forward verification, its goal is to reduce the entailment between separation formulas to the entailment between pure formulas by successively matching up aliased heap nodes between the antecedent and the consequent through folding, unfolding and matching [13]. When this happens, the heap formula in the antecedent is soundly approximated by returning a pure approximation of the form $\bigvee(\exists v^* {\cdot} \pi)^*$ for each given heap formula κ (using *XPure* function as in [13]).
- For our inference, its goal is to infer a precondition ϕ_3 and gather a set of constraints β_3 over specified uninterpreted relations. Along with the inferred frame Δ_3, we should be able to finally 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 interest is when both the antecedent and the consequent are heap free, and the rules in Figure 2 can in turn apply. Take note that these rules are applied in a *top-down* and *left-to-right* order.

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

$$\underline{[\textsc{inf}{-}[\textsc{unsat}]]} \qquad\qquad \underline{[\textsc{inf}{-}[\textsc{valid}]]}$$
$$\frac{\textsc{unsat}(\alpha_1)}{[v^*]\ \alpha_1 \vdash \alpha_2 \leadsto (\texttt{true}, \texttt{false}, \emptyset)} \qquad \frac{\alpha_1 \Rightarrow \alpha_2}{[v^*]\ \alpha_1 \vdash \alpha_2 \leadsto (\texttt{true}, \alpha_1, \emptyset)}$$

$$\underline{[\textsc{inf}{-}[\textsc{lhs}{-}\textsc{contra}]]} \qquad\qquad \underline{[\textsc{inf}{-}[\textsc{pre}{-}\textsc{derive}]]}$$
$$\frac{\begin{array}{c}\phi = \forall(FV(\alpha_1){-}v^*) \cdot \neg\alpha_1 \\ \textsc{unsat}(\alpha_1 \wedge \alpha_2) \quad \phi {\neq} \texttt{false}\end{array}}{[v^*]\ \alpha_1 \vdash \alpha_2 \leadsto (\phi, \texttt{false}, \emptyset)} \qquad \frac{\begin{array}{c}\phi {=} \forall(FV(\alpha_1, \alpha_2){-}v^*) \cdot (\neg\alpha_1 \vee \alpha_2) \\ \phi {\neq} \texttt{false}\end{array}}{[v^*]\ \alpha_1 \vdash \alpha_2 \leadsto (\phi, \alpha_1 \wedge \phi, \emptyset)}$$

$$\underline{[\textsc{inf}{-}[\textsc{rel}{-}\textsc{defn}]]}$$
$$[v^*, v_{rel}]\ \pi \vdash v_{rel}(u^*) \leadsto (\texttt{true}, \texttt{true}, \{\pi {\rightarrow} v_{rel}(u^*)\})$$

$$\underline{[\textsc{inf}{-}[\textsc{rel}{-}\textsc{oblg}]]}$$
$$\frac{[u^*]\ \alpha_1 \vdash \alpha_2 \leadsto (\phi_1, \Delta_1, \emptyset) \qquad [v^*]\ \alpha_1 \vdash \alpha_2 \leadsto (\phi_2, \Delta_2, \emptyset)}{[v^*, v_{rel}]\ \alpha_1 \wedge v_{rel}(u^*) \vdash \alpha_2 \leadsto (\phi_2, \Delta_1 \wedge \Delta_2, \{v_{rel}(u^*) {\rightarrow} \phi_1\})}$$

Fig. 2. Pure Bi-Abduction Rules

- Rule $[\textsc{inf}{-}[\textsc{and}]]$ breaks the conjunctive consequent into smaller components.
- Rules $[\textsc{inf}{-}[\textsc{unsat}]]$ and $[\textsc{inf}{-}[\textsc{valid}]]$ 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 the antecedent is being inferred. Note that $FV(\cdot)$ returns the set of free variables from its argument(s), while v^* is a shorthand 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 to prove 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 used 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* or *backwards* 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. Specifically, 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 satisfying 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 variables $[v^*]$ (ii) an extra precondition ϕ_2 that must be added (iii) a set of definitions and obligations β_2 on the specified 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 is empty, our new rule is simply the case that performs verification, without any inference.

Figure 3 captures a set of our Hoare rules with pure bi-abduction. Rule $\underline{[\text{INF−[SEQ]}]}$ shows how sequential composition $e_1; e_2$ is handled. The two inferred preconditions are conjunctively combined as $\phi_2 \wedge \phi_3$. Rule $\underline{[\text{INF−[IF]}]}$ deals with conditional expression. Our core language allows only boolean 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. Since the two preconditions ϕ_2, ϕ_3 come from two branches, both of them must hold for soundness; thus they are combined conjunctively in a conservative manner. Rule $\underline{[\text{INF−[ASSIGN]}]}$ handles assignment statement. We first define a

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

composition with update operator. Given a state $\varDelta_1$, a state change $\varDelta_2$, and a set of variables to be updated $X=\{x_1,\ldots,x_n\}$, the composition operator $\texttt{op}_X$ is defined as: $\varDelta_1\ \texttt{op}_X\ \varDelta_2 \overset{def}{=} \exists\, r_1..r_n \cdot (\rho_1\varDelta_1)\ \texttt{op}\ (\rho_2\varDelta_2)$, 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 $\varDelta_1$ with the corresponding initial value x_i in $\varDelta_2$ via a fresh variable r_i. The binary operator $\texttt{op}$ is either $\wedge$ or $*$. Instances of this operator will be used in the inference rules $\left[\underline{\textbf{INF}-[\textbf{ASSIGN}]}\right]$ and $\left[\underline{\textbf{INF}-[\textbf{CALL}]}\right]$. As illustrated in $\left[\underline{\textbf{INF}-[\textbf{CALL}]}\right]$, for each method call, we must ensure that its precondition is satisfied, and then add the expected postcondition into its residual state. 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 to the method's callers. Note that inference may occur during the entailment proving for the method's precondition.

$$\left[\underline{\textbf{INF}-[\textbf{SEQ}]}\right]$$

$$\frac{[v^*]\vdash\{\varDelta\}\ e_1\ \{\phi_2,\varDelta_2,\beta_2\}\qquad [v^*]\vdash\{\varDelta_2\}\ e_2\ \{\phi_3,\varDelta_3,\beta_3\}}{[v^*]\vdash\{\varDelta\}\ e_1;e_2\ \{\phi_2{\wedge}\phi_3,\varDelta_3,\beta_2{\cup}\beta_3\}}$$

$$\left[\underline{\textbf{INF}-[\textbf{IF}]}\right]$$

$$\frac{[v^*]\vdash\{\varDelta{\wedge}w'\}\ e_1\ \{\phi_2,\varDelta_2,\beta_2\}\qquad [v^*]\vdash\{\varDelta{\wedge}\neg w'\}\ e_2\ \{\phi_3,\varDelta_3,\beta_3\}}{[v^*]\vdash\{\varDelta\}\ \texttt{if}\ w\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \{\phi_2{\wedge}\phi_3,\varDelta_2{\vee}\varDelta_3,\beta_2{\cup}\beta_3\}}$$

$$\left[\underline{\textbf{INF}-[\textbf{ASSIGN}]}\right]$$

$$\frac{[v^*]\vdash\{\varDelta\}\ e\ \{\phi_2,\varDelta_2,\beta_2\}\qquad \varDelta_3{=}\exists\texttt{res}\cdot(\varDelta_2\wedge_u u'{=}\texttt{res})}{[v^*]\vdash\{\varDelta\}\ u{:=}e\ \{\phi_2,\varDelta_3,\beta_2\}}$$

$$\left[\underline{\textbf{INF}-[\textbf{CALL}]}\right]$$

$$\frac{\begin{array}{c} t_0\ \mathit{mn}\ (\texttt{ref}\ (t_i\ v_i)_{i=1}^{m-1},(t_j\ v_j)_{j=m}^{n})\ \ \varPhi_{pr}\ \ \varPhi_{po}\ \{c\}\in \mathit{Prog}\\ \rho{=}[v_k'/v_k]_{k=1}^{n}\quad \varPhi_{pr}'=\rho(\varPhi_{pr})\qquad W{=}\{v_1,\ldots,v_{m-1}\}\quad V{=}\{v_m,\ldots,v_n\}\\ [v^*]\ \varDelta\vdash\varPhi_{pr}'\leadsto(\phi_2,\varDelta_2,\beta_2)\qquad \varDelta_3{=}(\varDelta_2\wedge \mathit{nochange}(V))\ \ *_{V\cup W}\ \varPhi_{po}\end{array}}{[v^*]\vdash\{\varDelta\}\ \mathit{mn}(v_1,\ldots,v_{m-1},v_m,...v_n)\ \{\phi_2,\varDelta_3,\beta_2\}}$$

$$\left[\underline{\textbf{INF}-[\textbf{METH}]}\right]$$

$$\frac{\begin{array}{c}[v^*,v_{rel}^*]\vdash\{\varPhi_{pr}{\wedge}\bigwedge(u'{=}u)^*\}\ c\ \{\phi_2,\varDelta_2,\beta_2\}\qquad [v^*,v_{rel}^*]\ \varDelta_2\vdash\varPhi_{po}\leadsto(\phi_3,\varDelta_3,\beta_3)\\ \rho_1=\mathit{infer\_pre}(\beta_2{\cup}\beta_3)\qquad \rho_2=\mathit{infer\_post}(\beta_2{\cup}\beta_3)\\ \varPhi_{pr}^{n}=\rho_1(\varPhi_{pr}{\wedge}\phi_2{\wedge}\phi_3)\qquad \varPhi_{po}^{n}=\rho_2(\varPhi_{po}{*}\varDelta_3)\end{array}}{\vdash t_0\ \mathit{mn}\ ((t\ u)^*)\ \texttt{infer}\ [v^*,v_{rel}^*]\ \varPhi_{pr}\ \varPhi_{po}\ \{c\}\leadsto\varPhi_{pr}^{n}\ \varPhi_{po}^{n}}$$

Fig. 3. Hoare Rules with Pure Bi-Abduction

Lastly, we discuss the rule $\left[\underline{\textbf{INF}-[\textbf{METH}]}\right]$ for handling each method declaration. 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 uninterpreted relation. From this set β we infer the pre- and post-relations via two techniques described below.

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

Take note that, given the entire set β, we retrieve the set of definitions and obligations for post-relations through functions $\mathtt{def}_{\mathtt{po}}$ and $\mathtt{obl}_{\mathtt{po}}$ respectively, while we use functions $\mathtt{def}_{\mathtt{pr}}$ and $\mathtt{obl}_{\mathtt{pr}}$ for pre-relations.

• For post-relations, function *infer_post* applies a least fixed point analysis to the set of collected relational definitions $\mathtt{def}_{\mathtt{po}}(\beta)$. To compute the least fixed point over two domains used to instantiate the current framework, namely the numerical domain and the set/bag domain, we utilize FIXCALC [17] and FIXBAG [15], respectively. The call to the fixed point analysis is denoted as $\mathtt{LFP}(\mathtt{def}_{\mathtt{po}}(\beta))$. 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 uninterpreted relation $v_{rel_i}(v_i^*)$. Given that our aim is to infer the 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, whereby each unknown relation is substituted by the inferred formula, provided that this formula satisfies all the corresponding relational obligations from $\mathtt{obl}_{\mathtt{po}}(\beta)$.

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

• For pre-relations, we initially infer two kinds of precondition: one for base cases, the other for recursive calls. For base cases, we calculate the conjunction of all its obligations from $\mathtt{obl}_{\mathtt{pr}}(\beta)$ to obtain sufficient precondition $\mathtt{pre\_base_i}$ for each uninterpreted relation $v_{rel_i}(v_i^*)$. For recursive calls, we first derive the *recursive invariant* $\mathtt{rec\_inv_i}$ to relate the parameters of an arbitrary call to those of the first one. This can be achieved via a top-down fixed point analysis [18]. Because the parameters of an arbitrary call must also satisfy relevant relational obligations (e.g. $\mathtt{pre\_base_i}$), we will then be able to construct a precondition $\mathtt{pre\_rec_i}$ for each relation. An acceptable approximation α_i for each relation $v_{rel_i}(v_i^*)$ must satisfy simultaneously the precondition for base calls ($\mathtt{pre\_base_i}$), for an arbitrary recursive call ($\mathtt{pre\_rec_i}$) and the invariant $\mathtt{INV}$ (e.g. $\mathtt{a}{\geq}0 \wedge \mathtt{b}{\geq}0$ as in Sec. 4.4). The last step is to check the quality of candidate substitutions to keep the ones that satisfy not only the obligations but also definitions of each relation.

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

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 to be inferred, while ρ_1 and ρ_2 represent the substitutions obtained for pre- and post-relations, respectively.

Soundness. Soundness of inference is given in the extended version of this paper [22].

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 (see [22]). We can define these pure properties of data structures in the form of parameterized inductive definitions such as:

$$\begin{array}{lll}
\mathtt{prop\_defn\ HEAD[@V]\langle v\rangle} & \equiv & \mathtt{v{=}V} \\
\mathtt{prop\_defn\ SIZE[@R]\langle n\rangle} & \equiv & \mathtt{n{=}0}\ \vee\ \mathtt{SIZE\langle R,m\rangle{\wedge}n{=}1{+}m} \\
\mathtt{prop\_defn\ HEIGHT[@R]\langle n\rangle} & \equiv & \mathtt{n{=}0}\ \vee\ \mathtt{HEIGHT\langle R,m\rangle{\wedge}n{=}1{+}max(m)} \\
\mathtt{prop\_defn\ SUM[@V,@R]\langle s\rangle} & \equiv & \mathtt{s{=}0}\ \vee\ \mathtt{SUM\langle R,r\rangle{\wedge}s{=}V{+}r} \\
\mathtt{prop\_defn\ SET[@V,@R]\langle S\rangle} & \equiv & \mathtt{S{=}\{\}}\ \vee\ \mathtt{SET\langle R,S_2\rangle{\wedge}S{=}\{V\}{\sqcup}S_2} \\
\mathtt{prop\_defn\ SETADDR[@R]\langle S\rangle} & \equiv & \mathtt{S{=}\{\}}\ \vee\ \mathtt{SETADDR\langle R,S_2\rangle{\wedge}S{=}\{root\}{\sqcup}S_2} \\
\mathtt{prop\_defn\ MINP[@V,@R]\langle mi\rangle} & \equiv & \mathtt{mi{=}min(V)}\ \vee\ \mathtt{MINP\langle R,mi_2\rangle{\wedge}mi{=}min(V,mi_2)} \\
\mathtt{prop\_defn\ MAXP[@V,@R]\langle mx\rangle} & \equiv & \mathtt{mx{=}max(V)}\ \vee\ \mathtt{MAXP\langle R,mx_2\rangle{\wedge}mx{=}max(V,mx_2)},
\end{array}$$

where $\mathtt{V}, \mathtt{R}$ are values extracted from parameters $\mathtt{@V}$, $\mathtt{@R}$ resp. For example, to determine if $\mathtt{n}$ is the size of some data structure whose recursive pointer is annotated as $\mathtt{@REC}$, $\mathtt{SIZE[@REC]\langle n\rangle}$ would check the satisfiability of the base case (e.g. $\mathtt{n{=}0}$) and the inductive case (via recursive pointer $\mathtt{REC}$).

Using such definition, one can use the following command to incorporate size property directly to a linked-list predicate definition. Below, the annotations $\mathtt{@VAL}$, $\mathtt{@REC}$ are hardwired to two fields of the underlying heap structure $\mathtt{node}$:

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

Based on these commands and the definitions of pure properties, our system first constructs an entry (in a dictionary) for each targeting predicate. For example, there is one entry $(\mathtt{llN\langle root,n\rangle},(\mathtt{F1,F2,BC,IC}))$, where list of all value field annotations $\mathtt{F1{=}[\,]}$, list of all recursive pointer annotations $\mathtt{F2{=}[@REC]}$, base case $\mathtt{BC{=}\backslash[\_]} \rightarrow \mathtt{n{=}0}$ and inductive case $\mathtt{IC{=}\backslash[m_{REC}]} \rightarrow \mathtt{n{=}m_{REC}{+}1}$ ($\mathtt{m_{REC}}$ is the size property of corresponding recursive pointer $\mathtt{REC}$ of the linked-list). Using the dictionary, we can transform the base case and inductive case of original spatial predicate $\mathtt{ll}$ as follows:

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

Finally, we can synthesize $\mathtt{llN}$ predicate as previously shown in Sec. 2.

In short, the technique we present in this section aims at a systematic way to enrich spatial predicates with interesting pure properties. For space reasons, more complicated cases (i.e. when handling data structures with multiple links such as tree) can be found in [22]. While property extensions are user customizable, 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 then replace each heap predicate with its derived counterpart, followed by the introduction of uninterpreted pre- and post-relations before applying Hoare-style verification rules 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. Details can be found in the extended version of this paper [22]. Though codes for these examples are not too complicated, they illustrate the treatment of recursion (thus, the inter-procedural aspect). Therefore, the preconditions and postconditions derived can be quite intricate and would require considerable human efforts if they were constructed manually.[2]

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

```
node del_val(node x,int a)
  {
   if (x == null) return x;
   else if (x.val == a) {
       node tmp = x.next;
       free(x);
       return tmp; }
     else {
      x.next =
       del_val(x.next, a);
       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 the 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 needs to derive `llNB` predicate that also captures a bag `B` of values stored in the list:

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

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

$$\begin{array}{ll}\mathtt{requires} & \mathtt{llNB}\langle\mathtt{x},\mathtt{n},\mathtt{B}_1\rangle\\ \mathtt{ensures} & \mathtt{llNB}\langle\mathtt{res},\mathtt{m},\mathtt{B}_2\rangle\wedge((\mathtt{a}{\notin}\mathtt{B}_1\wedge\mathtt{B}_2{=}\mathtt{B}_1)\vee\mathtt{B}_1{=}\mathtt{B}_2\sqcup\{\mathtt{a}\});\end{array}$$

where `res` denotes the method's result.

[2] The source code of all examples can be found in our website [22].

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 program sets: `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 obtained 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 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. 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.

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

			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	18	13	72	Size	18	100	6.86				
`schedule2`	474	16	5	31	Size	16	100	10.58				
`pcidriver`	1036	29	29	100	Size	29	100	17.72				

Larger Examples. The last three rows from Table 1 demonstrate the applicability of SpecInfer on larger programs. The first two programs used to perform process scheduling are adopted from the Siemens test suite [8] 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 property to prove memory safety, size property is still useful for proving termination.

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 on some common heap 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 [5]. 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 systematic machinery for deriving formal specifications with more precise correctness properties.

A closely related research direction to ours concerns the inference of both shape and numerical properties. In [11], 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 [12] 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 and 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 our integrated analysis (or entailment procedure) that can handle both heap and pure properties simultaneously. A recent work [19] focuses on refining partial specifications, using a semi-automatic approach whereby predicate definitions are 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 [7] 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 [21,9,10] require templates in order to infer dependent types precise enough to prove a variety of safety properties such as the safety of array accesses. However, in both of these works, mutable data structures are not supported. Compared to above 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.

Acknowledgements. We would like to thank Duc-Hiep Chu for his useful comments.

References

1. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
2. Berdine, J., Cook, B., Ishtiaq, S.: SLAYER: Memory safety for systems-level code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 178–183. Springer, Heidelberg (2011)
3. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, pp. 289–300. ACM, New York (2009)
4. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Footprint analysis: A shape analysis that discovers preconditions. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 402–418. Springer, Heidelberg (2007)
5. Distefano, D.: Attacking large industrial code with bi-abductive inference. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 1–8. Springer, Heidelberg (2009)
6. Dudka, K., Peringer, P., Vojnar, T.: Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 372–378. Springer, Heidelberg (2011)
7. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI, pp. 405–416 (2012)
8. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: ICSE, pp. 191–200 (1994)
9. Kawaguchi, M., Rondon, P.M., Jhala, R.: Type-based data structure verification. In: PLDI, pp. 304–315 (2009)
10. Kawaguchi, M., Rondon, P.M., Jhala, R.: Dsolve: Safety verification via liquid types. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 123–126. Springer, Heidelberg (2010)
11. Magill, S., Berdine, J., Clarke, E., Cook, B.: Arithmetic strengthening for shape analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 419–436. Springer, Heidelberg (2007)
12. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: THOR: A tool for reasoning about shape and arithmetic. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 428–432. Springer, Heidelberg (2008)
13. Nguyen, H.H., David, C., Qin, S.C., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)

14. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local Reasoning about Programs that Alter Data Structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
15. Pham, T.-H., Trinh, M.-T., Truong, A.-H., Chin, W.-N.: FIXBAG: A fixpoint calculator for quantified bag constraints. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 656–662. Springer, Heidelberg (2011)
16. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
17. Popeea, C., Chin, W.-N.: Inferring disjunctive postconditions. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 331–345. Springer, Heidelberg (2008)
18. Popeea, C., Xu, D.N., Chin, W.-N.: A practical and precise inference and specializer for array bound checks elimination. In: PEPM, pp. 177–187 (2008)
19. Qin, S., Luo, C., Chin, W.-N., He, G.: Automatically refining partial specifications for program verification. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 369–385. Springer, Heidelberg (2011)
20. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: IEEE LICS, pp. 55–74.
21. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI, pp. 159–169 (2008)
22. Trinh, M.-T., Le, Q.L., David, C., Chin, W.-N.: Bi-Abduction with Pure Properties for Specification Inference (extended version) (2013) `loris-7.ddns.comp.nus.edu.sg/~project/SpecInfer/`
23. Vogels, F., Jacobs, B., Piessens, F., Smans, J.: Annotation inference for separation logic based verifiers. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE 2011. LNCS, vol. 6722, pp. 319–333. Springer, Heidelberg (2011)