Shape Analysis via Second-Order Bi-Abduction (Technical Report)

Quang Loc Le[1], Cristian Gherghina[2], Shengchao Qin[3], and Wei-Ngan Chin [1]

(1) Department of Computer Science, National University of Singapore
(2) Singapore University of Design and Technology (3) Teesside University

Abstract. We present a new modular shape analysis that can synthesize heap memory specification on a per method basis. We rely on a *second-order bi-abduction* mechanism that can give interpretations to unknown shape predicates. There are several novel features in our shape analysis. Firstly, it is grounded on second-order bi-abduction. Secondly, we distinguish unknown pre-predicates in pre-conditions, from unknown post-predicates in post-condition; since the former may be strengthened, while the latter may be weakened. Thirdly, we provide a new *heap guard* mechanism to support more precise preconditions for heap specification. Lastly, we formalise a set of derivation and normalization rules to give concise definitions for unknown predicates. Our approach has been proven sound and is implemented on top of an existing automated verification system. We show its versatility in synthesizing a wide range of intricate shape specifications.

1 Introduction

An important challenge for automatic program verifiers lies in inferring shapes describing abstractions for data structures used by each method. In the context of heap manipulating programs, determining the shape abstraction is crucial for proving memory safety and is a precursor to supporting functional correctness.

However, discovering shape abstractions can be rather challenging, as linked data structures span a wide variety of forms, from singly-linked lists, doubly-linked lists, circular lists, to tree-like data structures. Previous shape analysis proposals have made great progress in solving this problem. However, the prevailing approach relies on using a predefined vocabulary of shape definitions (typically limited to singly-linked list segments) and trying to determine if any of the pre-defined shapes fit the data structures used. This works well with programs that use simpler shapes, but would fail for programs which use more intricate data structures. An example is the method below (written in C and adapted from [19]) to build a tree whose leaf nodes are linked as a list.

```
struct tree { struct tree* parent; struct tree* l; struct tree* r; struct tree* next}
struct tree* tll(struct tree* x, struct tree* p, struct tree* t)
{ x->parent = p;
  if (x->r==NULL) { x->next=t; return x; }
  else{  struct tree* lm = tll(x->r, x, t); return tll(x->l, x, lm); } }
```

Our approach to modular shape analysis would introduce an unknown pre-predicate H (as the pre-condition), and an unknown post-predicate G (as the post-condition), as shown below, where res is the method's result.

```
requires  H(x, p, t)      ensures  G(x, p, res, t)
```

Using Hoare-style verification and a new second-order bi-abduction entailment procedure, we would derive a set of relational assumptions for the two unknown predicates. These derived assumptions are to ensure memory safety, and can be systematically transformed into concise predicate definitions for the unknown predicates, such as:

$$\begin{array}{rl} \mathtt{H(x,p,t)} \equiv & \mathtt{x}{\mapsto}\mathtt{tree}(\mathcal{D}_\mathtt{p},\mathcal{D}_\mathtt{l},\mathtt{r},\mathcal{D}_\mathtt{n}) \wedge \mathtt{r{=}NULL} \\ & \vee\ \mathtt{x}{\mapsto}\mathtt{tree}(\mathcal{D}_\mathtt{p},\mathtt{l},\mathtt{r},\mathcal{D}_\mathtt{n}){*}\mathtt{H(l,x,lm)}{*}\mathtt{H(r,x,t)} \wedge \mathtt{r} \neq \mathtt{NULL} \end{array}$$

$$\begin{array}{rl} \mathtt{G(x,p,res,t)} \equiv & \mathtt{x}{\mapsto}\mathtt{tree}(\mathtt{p},\mathcal{D}_\mathtt{l},\mathtt{r},\mathtt{t}) \wedge \mathtt{res{=}x}{\wedge}\mathtt{r{=}NULL} \\ & \vee\ \mathtt{x}{\mapsto}\mathtt{tree}(\mathtt{p},\mathtt{l},\mathtt{r},\mathcal{D}_\mathtt{n}){*}\mathtt{G(l,x,res,lm)}{*}\mathtt{G(r,x,lm,t)}{\wedge}\mathtt{r}{\neq}\mathtt{NULL} \end{array}$$

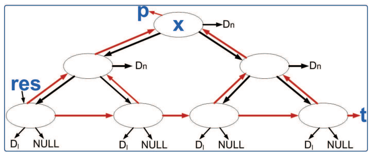

Fig. 1. An example of G(x,p,res,t)

The derived pre-predicate H captures a binary tree-like shape that would be traversed by the method. $\mathtt{x}{\mapsto}\mathtt{tree}(\mathcal{D}_\mathtt{p},\mathcal{D}_\mathtt{l},\mathtt{r},\mathcal{D}_\mathtt{n})$ denotes that x refers to a tree node with its parent,l,r and next fields being $\mathcal{D}_\mathtt{p}$, $\mathcal{D}_\mathtt{l}$, r and $\mathcal{D}_\mathtt{n}$, respectively. We use dangling references, such as $\mathcal{D}_\mathtt{l}$, $\mathcal{D}_\mathtt{p}$, $\mathcal{D}_\mathtt{n}$, as generic markers that denote field pointers that are not traversed by the method. Thus no assertion can be made on any of the $\mathcal{D}$ pointers. The post-predicate G, illustrated in Fig 1, adds parent field links for all nodes, and next field links for just the leaves. [1]
Current shape analysis mechanisms [12,4,6] are unable to infer pre/post specifications that ensure memory-safety for such complex examples. In this paper, we propose a fresh approach to shape analysis that can synthesize, from scratch, a set of shape abstractions that ensure memory safety. The central concept behind our proposal is the use of *unknown predicates* (or *second-order variables*) as place holders for shape predicates that are to be synthesized directly from proof obligations gathered by our verification process. Our proposal is based on a novel *bi-abductive entailment* that supports *second-order* variables. The core of the new entailment procedure generates a set of relational assumptions on unknown predicates to ensure memory safety. These assumptions are then refined into predicate definitions, by predicate *derivation* and *normalization* steps.

By building the generation of the required *relational assumptions* over unknown predicates directly into the new entailment checker, we were able to integrate our shape analysis into an existing program verifier with changes made only to the entailment process, rather than the program verification/analysis itself. Our proposed shape analysis thus applies an almost standard set of Hoare rules in constructing proof obligations which are discharged through the use of a new *second-order* bi-abductive entailment.

This paper makes the following four primary contributions.

- A novel *second-order bi-abduction* guided by an annotation scheme to infer relational assumptions (over unknown predicates) as part of Hoare-style verification.
- A set of formal rules for *deriving* and *normalizing* each unknown predicate definition from the relational assumptions with heap guard conditions.
- A *sound* and *modular* shape analysis, that is applied on a per method basis[2].
- Our implementation and experiments on *shape inference*, closely integrated into an automated verification system. The tool is available for online use (see Appendix

[1] Note that new links formed by the method are colored in red.

[2] Most existing shape analyses require either global analyses or re-verification after analysis. For example, bi-abduction in [6] requires its method's inferred pre-condition to be re-verified due to its use of over-approximation on heap pre-condition which can be unsound.

C to download a virtual machine including S2's source code) at
`http://loris-7.ddns.comp.nus.edu.sg/~project/s2/beta`

2 Logic Syntax for Shape Specification

Separation logic is an extension of Hoare logic for reasoning with heap-based programs [20,27]. We outline below the fragment underlying the proposed analysis:

Disj. formula	Φ	$::= \Delta \mid \Phi_1 \vee \Phi_2$
Guarded Disj.	Φ^g	$::= \Delta \mid (\Delta \;\texttt{@}\; (\kappa\wedge\pi)) \mid {\Phi^g}_1 \vee {\Phi^g}_2$
Conj. formula	Δ	$::= \exists\bar{v}\cdot(\kappa\wedge\pi)$
Spatial formula	κ	$::= \texttt{emp} \mid \top \mid v{\mapsto}c(\bar{v}) \mid \texttt{P}(\bar{v}) \mid \texttt{U}(\bar{v}) \mid \kappa_1{*}\kappa_2$
Pure formula	π	$::= \alpha \mid \neg\alpha \mid \pi_1\wedge\pi_2$
Var (Dis)Equality	α	$::= v \mid v_1{=}v_2 \mid v{=}\texttt{NULL} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{NULL}$
Pred. Defn.	$\texttt{P}^{\texttt{def}}$	$::= \texttt{P}(\bar{v}) \equiv \Phi^g$
Pred. Dict.	Γ	$::= \{\texttt{P}_1^{\texttt{def}}, \ldots, \texttt{P}_n^{\texttt{def}}\}$

$\texttt{P} \in$ Known Predicates $\quad \texttt{U} \in$ Unknown Predicates
$c \in$ Data Nodes $\quad v \in$ Variables $\quad \bar{v} \equiv v_1 \ldots v_n$

We introduce $\Delta \;\texttt{@}\; (\kappa\wedge\pi)$, a special syntactic form called *guarded heap* that capture a heap context $\kappa\wedge\pi$ in which Δ holds. Thus, $\Delta \;\texttt{@}\; (\kappa\wedge\pi)$ holds for heap configurations that satisfy Δ and that can be extended such that they satisfy $\Delta * \kappa\wedge\pi$. In Sec.5 we will describe its use in allowing our shape inference to incorporate path sensitive information in the synthesized predicates. The assertion language is also extended with the following formula for describing heaps: emp denoting the empty heap; $\top$ denoting an arbitrary heap (pointed by dangling reference); points-to assertion, $x{\mapsto}c(\bar{v})$, specifying the heap in which x points to a data structure of type c whose fields contain the values $\bar{v}$; known predicate, $\texttt{P}(\bar{v})$, which holds for heaps in which the shape of the memory locations reachable from $\bar{v}$ can be described by the P predicate; unknown predicates, $\texttt{U}(\bar{v})$, with no prior given definitions. Separation conjunction $\kappa_1{*}\kappa_2$ holds for heaps that can be partitioned in two disjoint components satisfying κ_1 and κ_2, respectively. The pure formula captures only pointer equality and disequality. We allow a special constant NULL to denote a pointer which does not point to any heap location. Known predicates $\texttt{P}(\bar{v})$ are defined inductively through disjunctive formulas Φ^g. Their definitions are either user-given or synthesised by our analysis. We will use Γ to denote the repository (or set) of available predicate definitions. Through our analysis, we shall construct an inductive definition for each unknown predicate, where possible. Unknown predicates that have *not* been instantiated would not have any definition. They denote data fields that are not accessed by their methods, and would be marked as *dangling pointers*.

3 Overview of Our Approach

Our approach comprises three main steps: (i) inferring relational assumptions for unknown predicates via Hoare-style verification, (ii) deriving predicates from relational assumptions, (iii) normalizing predicates. For (i), a key machinery is the entailment procedure that must work with second-order variables (unknown predicates). Previous bi-abduction entailment proposals, pioneered by [6], would take an antecedent $\Delta_{\mathtt{ante}}$ and a consequent $\Delta_{\mathtt{conseq}}$ and return a frame residue $\Delta_{\mathtt{frame}}$ and the precondition $\Delta_{\mathtt{pre}}$, such that the following holds: $\Delta_{\mathtt{pre}}{*}\Delta_{\mathtt{ante}} \models \Delta_{\mathtt{conseq}}{*}\Delta_{\mathtt{frame}}$. Here, all four components use separation logic formulas based on known predicates with prior definitions.

Taking a different tact, we start with an existing entailment procedure for separation logic with user-defined predicates, and extend it to accept formulas with second-order variables such that given an antecedent $\Delta_{\mathtt{ante}}$ and a consequent $\Delta_{\mathtt{conseq}}$ the resulting entailment procedure infers both the frame residue $\Delta_{\mathtt{frame}}$ and a set (or conjunction) of relational assumptions (on unknowns) of the form $\mathcal{R} = \bigwedge_{i=1}^{n}(\Delta_i \Rightarrow \Phi^g{}_i)$ such that:

$$\mathcal{R} \wedge \Delta_{\mathtt{ante}} \models \Delta_{\mathtt{conseq}}{*}\Delta_{\mathtt{frame}}$$

The inferred $\mathcal{R}$ ensures the entailment's validity. We shall use the following notation $\Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} \leadsto (\mathcal{R},\ \Delta_{\mathtt{frame}})$ for this second-order bi-abduction process.

There are two scenarios to consider for unknown predicates: (1) $\Delta_{\mathtt{ante}}$ contains an *unknown* predicate instance that matched with a points-to or known predicate in $\Delta_{\mathtt{conseq}}$; (2) $\Delta_{\mathtt{conseq}}$ contains an *unknown* predicate instance. An example of the first scenario is:

$$\mathtt{U(x)} \vdash \mathtt{x{\mapsto}snode(n)} \leadsto (\mathtt{U(x){\Rightarrow}x{\mapsto}snode(n){*}U_0(n)},\ \ \mathtt{U_0(n)})$$

Here, we generated a relational assumption to denote an *unfolding* (or instantiation) for the unknown predicate U to a heap node snode followed by another unknown $\mathtt{U_0(n)}$ predicate. The data structure snode is defined as struct snode { struct snode* next}. A simple example of the second scenario is shown next.

$$\mathtt{x{\mapsto}snode(NULL){*}y{\mapsto}snode(NULL)} \vdash \mathtt{U_1(x)} \leadsto (\mathtt{x{\mapsto}snode(NULL){\Rightarrow}U_1(x)},\ \mathtt{y{\mapsto}snode(NULL)})$$

The generated relational assumption depicts a *folding* process for unknown $\mathtt{U_1(x)}$ which captures a heap state traversed from the pointer x. Both folding and unfolding of unknown predicates are crucial for second-order bi-abduction. To make it work properly for unknown predicates with multiple parameters, we shall later provide a novel #-annotation scheme to guide these processes. For the moment, we shall use this annotation scheme implicitly. Consider the following method which traverses a singly-linked list and converts it to a doubly-linked list (let us ignore the states $\alpha_1, .., \alpha_5$ for now):

```
struct node { struct node* prev; struct node* next}
void sll2dll(struct node* x, struct node* q)
{(α1) if (x==NULL) (α2) return; (α3) x->prev = q; (α4) sll2dll(x->next, x); (α5)}
```

To synthesize the shape specification for this method, we introduce two unknown predicates, H for the pre-condition and G for the post-condition, as below.

$$\mathtt{requires}\quad \mathtt{H(x,q)} \qquad \mathtt{ensures}\quad \mathtt{G(x,q)}$$

We then apply code verification using these pre/post specifications with unknown predicates and attempt to collect a set of relational assumptions (over the unknown predicates) that must hold to ensure memory-safety. These assumptions would also ensure that the pre-condition of each method call is satisfied, and that the coresponding post-condition is ensured at the end of the method body. For example, our analysis can infer four relational assumptions for the sll2dll method as shown in Fig. 2(a).

These relational assumptions include two new unknown predicates, $\mathtt{H_p}$ and $\mathtt{H_n}$, created during the code verification process. All relational assumptions are of the form

$(\mathtt{A1}).\mathtt{H}(\mathtt{x},\mathtt{q}) \wedge \mathtt{x}{=}\mathtt{NULL} \Rightarrow \mathtt{G}(\mathtt{x},\mathtt{q})$

$(\mathtt{A2}).\mathtt{H}(\mathtt{x},\mathtt{q}) \wedge \mathtt{x}{\neq}\mathtt{NULL} \Rightarrow$

$\quad \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{x_p},\mathtt{x_n}){*}\mathtt{H_p}(\mathtt{x_p},\mathtt{q}){*}\mathtt{H_n}(\mathtt{x_n},\mathtt{q})$

$(\mathtt{A3}).\mathtt{H_n}(\mathtt{x_n},\mathtt{q}) \Rightarrow \mathtt{H}(\mathtt{x_n},\mathtt{x})\ @\ \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{q},\mathtt{x_n})$

$(\mathtt{A4}).\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{q},\mathtt{x_n}){*}\mathtt{G}(\mathtt{x_n},\mathtt{x}) \Rightarrow \mathtt{G}(\mathtt{x},\mathtt{q})$

(a)

(α_1). $\mathtt{H}(\mathtt{x},\mathtt{q})$

(α_2). $\mathtt{H}(\mathtt{x},\mathtt{q}){\wedge}\mathtt{x}{=}\mathtt{NULL}$

(α_3). $\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{x_p},\mathtt{x_n}){*}\mathtt{H_p}(\mathtt{x_p},\mathtt{q}){*}\mathtt{H_n}(\mathtt{x_n},\mathtt{q}){\wedge}\mathtt{x}{\neq}\mathtt{NULL}$

(α_4). $\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{q},\mathtt{x_n}){*}\mathtt{H_p}(\mathtt{x_p},\mathtt{q}){*}\mathtt{H_n}(\mathtt{x_n},\mathtt{q}){\wedge}\mathtt{x}{\neq}\mathtt{NULL}$

(α_5). $\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{q},\mathtt{x_n}){*}\mathtt{H_p}(\mathtt{x_p},\mathtt{q}){*}\mathtt{G}(\mathtt{x_n},\mathtt{x}){\wedge}\mathtt{x}{\neq}\mathtt{NULL}$

(b)

Fig. 2. Relational assumptions (a) and program states (b) for `sll2dll`

$\Delta_{\mathtt{lhs}}{\Rightarrow}\Delta_{\mathtt{rhs}}$, except for (A3) which has the form $\Delta_{\mathtt{lhs}}{\Rightarrow}\Delta_{\mathtt{rhs}}\ @\ \Delta_{\mathtt{g}}$ where $\Delta_{\mathtt{g}}$ denotes a heap guard condition. Such heap guard condition allows more precise pre-conditions to be synthesized (e.g. $\mathtt{H_n}$ in (A3)), and is shorthand for $\Delta_{\mathtt{lhs}}{*}\Delta_{\mathtt{g}}{\Rightarrow}\Delta_{\mathtt{rhs}}{*}\Delta_{\mathtt{g}}$.

Let us look at how relational assumptions are inferred. At the start of the method, we have (α_1), shown in Fig. 2 (b), as our program state. Upon exit from the then branch, the verification requires that the postcondition $\mathtt{G}(\mathtt{x},\mathtt{q})$ be established by the program state (α_2), generating the relational assumption (A1) via the following entailment:

$$(\alpha_2) \vdash \mathtt{G}(\mathtt{x},\mathtt{q}) \rightsquigarrow (\mathtt{A1},\ \mathtt{emp} \wedge \mathtt{x}{=}\mathtt{NULL}) \qquad \text{(E1)}$$

To get ready for the field access `x->prev`, the following entailment is invoked to unfold the unknown `H` predicate to a heap node, generating the relational assumption (A2):

$$\mathtt{H}(\mathtt{x},\mathtt{q}){\wedge}\mathtt{x}{\neq}\mathtt{NULL} \vdash \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{x_p},\mathtt{x_n}) \rightsquigarrow (\mathtt{A2},\ \mathtt{H_p}(\mathtt{x_p},\mathtt{q}){*}\mathtt{H_n}(\mathtt{x_n},\mathtt{q}) \wedge \mathtt{x}{\neq}\mathtt{NULL}) \qquad \text{(E2)}$$

Two new unknown predicates $\mathtt{H_p}$ and $\mathtt{H_n}$ are added to capture the `prev` ($\mathtt{x_p}$) and `next` ($\mathtt{x_n}$) fields of `x` (i.e. they represent heaps referred to by $\mathtt{x_p}$ and $\mathtt{x_n}$ respectively). After binding, the verification now reaches the state (α_3), which is then changed to (α_4) by the field update `x->prev = q`. Relational assumption (A3) is inferred from proving the precondition $\mathtt{H}(\mathtt{x_n},\mathtt{x})$ of the recursive call `sll2dll(x->next, x)` at the program state (α_4):

$$(\alpha_4) \vdash \mathtt{H}(\mathtt{x_n},\mathtt{x}) \rightsquigarrow (\mathtt{A3},\ \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{q},\mathtt{x_n}){*}\mathtt{H_p}(\mathtt{x_p},\mathtt{q}){\wedge}\mathtt{x}{\neq}\mathtt{NULL}) \qquad \text{(E3)}$$

Note that the heap guard $\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{q},\mathtt{x_n})$ from (α_4) is recorded in (A3), and is crucial for predicate derivation. The program state at the end of the recursive call, (α_5), is required to establish the post-condition $\mathtt{G}(\mathtt{x},\mathtt{q})$, generating the relational assumption (A4):

$$(\alpha_5) \vdash \mathtt{G}(\mathtt{x},\mathtt{q}) \rightsquigarrow (\mathtt{A4},\ \mathtt{H_p}(\mathtt{x_p},\mathtt{q}){\wedge}\mathtt{x}{\neq}\mathtt{NULL}) \qquad \text{(E4)}$$

These relational assumptions are automatically inferred symbolically during code verification. Our next step (ii) uses a predicate derivation procedure to transform (by either equivalence-preserving or abductive steps) the set of relational assumptions into a set of predicate definitions. Sec. 5 gives more details on predicate derivation. For our `sll2dll` example, we initially derive the following predicate definitions (for `H` and `G`):

$$\mathtt{H}(\mathtt{x},\mathtt{q}) \equiv \mathtt{emp} \wedge \mathtt{x}{=}\mathtt{NULL} \vee \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{x_p},\mathtt{x_n}) * \mathtt{H_p}(\mathtt{x_p},\mathtt{q}) * \mathtt{H}(\mathtt{x_n},\mathtt{x})$$
$$\mathtt{G}(\mathtt{x},\mathtt{q}) \equiv \mathtt{emp} \wedge \mathtt{x}{=}\mathtt{NULL} \vee \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{q},\mathtt{x_n}) * \mathtt{G}(\mathtt{x_n},\mathtt{x})$$

In the last step (iii), we use a normalization procedure to simplify the definition of predicate `H`. Since $\mathtt{H_p}$ is discovered as a dangling predicate, the special variable $\mathcal{D}_\mathtt{p}$ corresponds to a *dangling reference* introduced: $\mathtt{H}(\mathtt{x},\mathtt{q}) \equiv \mathtt{emp}{\wedge}\mathtt{x}{=}\mathtt{NULL}{\vee}\mathtt{x}{\mapsto}\mathtt{node}(\mathcal{D}_\mathtt{p},\mathtt{x_n}){*}$ $\mathtt{H}(\mathtt{x_n},\mathtt{x})$. Furthermore, we can synthesize a more concise $\mathtt{H_2}$ from `H` by eliminating its useless q parameter: $\mathtt{H(x,q)} \equiv \mathtt{H_2(x)}$ and $\mathtt{H_2(x)} \equiv \mathtt{emp} \wedge \mathtt{x{=}NULL} \vee \mathtt{x{\mapsto}node}(\mathcal{D}_p, \mathtt{x_n}) * \mathtt{H_2(x_n)}$.

The `tll` example. Let us revisit the `tll` example shown in Sec 1. To synthesize the shape specification for this method, we introduce two unknown predicates, `H` for the pre-condition and `G` for the post-condition, as mentioned earlier.

$$\mathtt{requires}\quad \mathtt{H(x,p,t)} \qquad\qquad \mathtt{ensures}\quad \mathtt{G(x,p,res,t)}$$

We then apply code verification using these pre/post specifications with unknown predicates and attempt to collect a set of relational assumptions (over the unknown predicates) that must hold to ensure memory-safety. These assumptions would also ensure that the pre-condition of each method call is satisfied, and that the post-condition is ensured at the end of its method body.

For example, our analysis can infer the following five relational assumptions for the `tll` method:

$(1).\mathtt{H(x,p,t)} \Rightarrow \mathtt{x{\mapsto}tree(x_p,l,r,n)} * \mathtt{H_p(x_p,p,t)} * \mathtt{H_l(l,p,t)} * \mathtt{H_r(r,p,t)} * \mathtt{H_n(n,p,t)}$
$(2).\mathtt{H_r(r,p,t)} \wedge \mathtt{r{\neq}NULL} \;@\; \mathtt{x{\mapsto}tree(p,l,r,n)} \Rightarrow \mathtt{H(r,x,t)}$
$(3).\mathtt{H_l(l,p,t)} \Rightarrow \mathtt{H(l,x,lm)} \;@\; (\mathtt{x{\mapsto}tree(p,l,r,n)} \wedge \mathtt{r{\neq}NULL})$
$(4).\mathtt{H_l(l,p,t)} * \mathtt{H_r(r,p,t)} * \mathtt{x{\mapsto}tree(p,l,r,t)} \wedge \mathtt{r{=}NULL} \wedge \mathtt{res{=}x} \Rightarrow \mathtt{G(x,p,res,t)}$
$(5).\mathtt{H_n(n,p,t)} * \mathtt{x{\mapsto}tree(p,l,r,n)} * \mathtt{G(r,x,lm,t)} * \mathtt{G(l,x,res,lm)} \wedge$
$\qquad \mathtt{r{\neq}NULL} \Rightarrow \mathtt{G(x,p,res,t)}$

Our relational assumptions include four new unknown predicates, $\mathtt{H_p}, \mathtt{H_l}, \mathtt{H_r}$ and $\mathtt{H_n}$, that were created during the code verification process.

Let us look at how relational assumptions are inferred. For illustration, we annotate program states into the `tll` example as follows:

```
struct tree* tll(struct tree* x, struct tree* p, struct tree* t) {
 (S1)x->parent = p;
 (S2)if (x->r==NULL) {(S3) x->next=t; return x; (S6)}
    else{(S4) struct tree* lm = tll(x->r, x, t);(S5) return tll(x->l, x, lm); (S7) }
 }
```

(S1). $\mathtt{H(x,p,t)}$

(S2). $\mathtt{x{\mapsto}tree(p,l,r,n)} * \mathtt{H_l(l,p,t)} * \mathtt{H_r(r,p,t)} * \mathtt{H_n(n,p,t)} * \mathtt{H_p(x_P,p,t)}$

(S3). $\mathtt{x{\mapsto}tree(p,l,r,n)} * \mathtt{H_l(l,p,t)} * \mathtt{H_r(r,p,t)} * \mathtt{H_n(n,p,t)} \wedge r{=}\mathtt{NULL}$

(S4). $\mathtt{x{\mapsto}tree(p,l,r,n)} * \mathtt{H_l(l,p,t)} * \mathtt{H_r(r,p,t)} * \mathtt{H_n(n,p,t)} \wedge r{\neq}\mathtt{NULL}$

(S5). $\mathtt{x{\mapsto}tree(p,l,r,n)} * \mathtt{H_l(l,p,t)} * \mathtt{G(r,x,lm,t)} * \mathtt{H_n(n,p,t)} \wedge r{\neq}\mathtt{NULL}$

(S6). $\mathtt{x{\mapsto}tree(p,l,r,t)} * \mathtt{H_l(l,p,t)} * \mathtt{H_r(r,p,t)} \wedge \mathtt{r{=}NULL} \wedge \mathtt{res{=}x}$

(S7). $\mathtt{x{\mapsto}tree(p,l,r,n)} * \mathtt{G(l,x,res,lm)} * \mathtt{G(r,x,lm,t)} * \mathtt{H_n(n,p,t)} \wedge \mathtt{r{\neq}NULL}$

At the start of the method, we have (S1), shown above, as our program state. Due to a field update, `x->parent`, relational assumption (1) was inferred which lead to (S2). The conditional evaluation led to (S3) and (S4), as the program states at the start of the then-branch and else-branch, respectively. Relational assumption (2) was then inferred from proving pre-condition `H(r, x, t)` of recursive call `tll(x->r, x, t)` under (S4), yielding program state (S5). Also, (3) was inferred from proving pre-condition `H(l, x, lm)` of the second recursive call `tll(x->l, x, lm)`. When inferring (3), heap guard $\mathtt{x{\mapsto}tree(p,l,r,n)} \wedge \mathtt{r{\neq}NULL}$ from the program state (S5) was used, since

$\mathtt{H}_1(\mathtt{l},\mathtt{p},\mathtt{t}){\Rightarrow}\mathtt{H}(\mathtt{l},\mathtt{x},\mathtt{lm})$, by itself, neither capture a connected context $\mathtt{r}{\neq}\mathtt{NULL}$ from the then-branch, nor properly instantiate the back (parent) pointer $\mathtt{x}$. The program state (S6) at the end of then-branch was then used to prove post-condition $\mathtt{G}(\mathtt{x},\mathtt{p},\mathtt{res},\mathtt{t})$. This proving lead to relational assumption (4). Similarly, program state (S7) at the end of the else-branch, which assumed $\mathtt{G(r,x,lm,t)}{*}\mathtt{G(l,x,res,lm)}$ from the two recursive calls, would infer (5) when proving the post-condition of the method itself.

These relational assumptions are inferred symbolically during code verification with the help of second-order bi-abduction mechanism that we are proposing. They are also being *modularly* inferred on a per method basis, using automatically generated template pre/post conditions with unknown predicates.

Our next phase uses a predicate synthesis and normalization procedure to transform (by either equivalence-preserving or abductive steps) the set of relational assumptions into a set of predicate definitions. For our running example, we synthesize the following predicate definitions (for H and G):

$$\begin{array}{l} \mathtt{H(x,p,t)} \equiv \mathtt{x}{\mapsto}\mathtt{tree}(\mathcal{D}_\mathtt{p},\mathcal{D}_\mathtt{l},\mathtt{r},\mathcal{D}_\mathtt{n}) \wedge \mathtt{r{=}NULL} \\ \qquad\qquad \vee\ \mathtt{x}{\mapsto}\mathtt{tree}(\mathcal{D}_\mathtt{p},\mathtt{l},\mathtt{r},\mathcal{D}_\mathtt{n}){*}\mathtt{H(l,x,lm)}{*}\mathtt{H(r,x,t)} \wedge \mathtt{r} \neq \mathtt{NULL} \\ \mathtt{G(x,p,res,t)} \equiv \mathtt{x}{\mapsto}\mathtt{tree}(\mathtt{p},\mathcal{D}_\mathtt{l},\mathtt{r},\mathtt{t}) \wedge \mathtt{res{=}x}{\wedge}\mathtt{r{=}NULL} \\ \qquad\qquad \vee\ \mathtt{x}{\mapsto}\mathtt{tree}(\mathtt{p},\mathtt{l},\mathtt{r},\mathcal{D}_\mathtt{n}){*}\mathtt{G(l,x,res,lm)}{*}\mathtt{G(r,x,lm,t)}{\wedge}\mathtt{r{\neq}NULL} \end{array}$$

The variables $\mathcal{D}_\mathtt{l}$, $\mathcal{D}_\mathtt{p}$ and $\mathcal{D}_\mathtt{n}$ correspond to *dangling predicates*. Note that for memory safety, the input tree x must contain at least one node and that x->l must be non-null when x->r is non-null. These requirements are captured by our synthesized pre-predicate. Compared to some prior shape analyses, such as [12,16], which requires the entire program to be available for analysis, our approach can perform this task modularly on a per method basis instead.

Our approach currently works only for shape abstractions of tree-like data structures with forward and back pointers. (We are unable to infer specifications for graph-like or overlaid data structures yet.) These abstractions are being inferred *modularly* on a per method basis. The inferred preconditions are typically the weakest ones that would ensure memory safety, and would be applicable to all contexts of use. Furthermore, the normalization step aims to ensure concise and re-useable predicate definitions. We shall next elaborate and formalise on our second-order bi-abduction process.

4 Second-Order Bi-Abduction with an Annotation Scheme

We have seen the need for a bi-abductive entailment procedure to systematically handle unknown predicates. To cater to predicates with multiple parameters, we shall use an automatic *#-annotation* scheme to support both unfolding and folding of unknown predicates. Consider a predicate $\mathtt{U}(\mathtt{v}_1,..,\mathtt{v}_\mathtt{n},\mathtt{w}_1\#,..,\mathtt{w}_\mathtt{m}\#)$, where parameters $\mathtt{v}_1,..,\mathtt{v}_\mathtt{n}$ are unannotated and parameters $\mathtt{w}_1,..,\mathtt{w}_\mathtt{m}$ are #-annotated. From the perspective of unfolding, we permit each variable from $\mathtt{v}_1,..,\mathtt{v}_\mathtt{n}$ to be instantiated at most once (we call them *instantiatable*), while variables $\mathtt{w}_1,..,\mathtt{w}_\mathtt{m}$ are *disallowed* from instantiation (we call them *non-instantiatable*). This scheme ensures that each pointer is instantiated at most once, and avoids formulae, like $\mathtt{U}_3(\mathtt{y},\mathtt{y})$ or $\mathtt{U}_2(\mathtt{r},\mathtt{y}){*}\mathtt{U}_3(\mathtt{y},\mathtt{x}\#)$, from being formed. Such formulae, where a variable may be repeatedly instantiated, may cause a trivial FALSE pre-condition to be inferred. Though sound, it is imprecise. From the perspective of folding, we allow heap traversals to start from variables $v_1, .., v_n$ and would stop whenever references to $w_1, .., w_m$ are encountered. This allows us to properly infer segmented shape predicates and back pointers. Our annotation scheme is fully automated, as we would infer the $\#$-annotation of pre-predicates based on which parameters could be field accessed; while parameters of post-predicates are left unannotated. For our running example, since q parameter is not field accessed (in its method's body), our automatic annotation scheme would start with the following pre/post specification:

$$\texttt{requires}\ \ H(x, q\#) \qquad \texttt{ensures}\ \ G(x, q)$$

Unfold. The entailment below results in an unfolding of the unknown H predicate. It is essentially (E2) in Sec 3, except that q is marked explicitly as non-instantiatable.

$$H(x, q\#) \wedge x \neq NULL \vdash x \mapsto node(x_p, x_n) \leadsto (A2,\ \Delta_1) \qquad (E2')$$

With non-instantiatable variables explicitly annotated, the assumption (A2) becomes:

$$A2\ \equiv\ H(x, q\#) \wedge x \neq NULL \Rightarrow x \mapsto node(x_p, x_n) * H_p(x_p, q\#) * H_n(x_n, q\#)$$

As mentioned earlier, we generated a new unknown predicate for each pointer field (H_p for x_p, and H_n for x_n), so as to allow the full recovery of the shape of the data structure being traversed or built. Note that each x, x_p, x_n appears only once in unannotated forms, while the annotated $q\#$ remains annotated throughout to prevent the pointer from being instantiated. If we allow q to be instantiatable in (E2′) above, we will instead obtain:

$$H(x, q) \wedge x \neq NULL \vdash x \mapsto node(x_p, x_n) \leadsto (A2',\ \Delta_1')$$

We get $A2'\ \equiv\ H(x, q) \wedge x \neq NULL \Rightarrow x \mapsto node(x_p, x_n) * H_p(x_p, q\#) * H_n(x_n, q\#) * \underline{U_2(q, x\#)}$, where the unfolding process creates extra unknown predicate $U_2(q, x\#)$ to capture shape for q.

Our proposal for instantiating unknown predicates is also applicable when known predicates appear in the RHS. These known predicates may have parameters that act as *continuation fields* for the data structure. An example is the list segment $lseg(x, p)$ predicate where the parameter p is a continuation field.

$$\begin{aligned} ll(x) &\equiv emp \wedge x = NULL \vee x \mapsto snode(n) * ll(n) \\ lseg(x, p) &\equiv emp \wedge x = p \vee x \mapsto snode(n) * lseg(n, p) \end{aligned}$$

Where snode (defined in the previous section) denotes singly-linked list node. Note that continuation fields play the same role as fields for data nodes. Therefore, for such parameters, we also generate new unknown parameters to capture the connected data structure that may have been traversed. We illustrate this with two examples:

$$U(x) \vdash ll(x) \leadsto (U(x) \Rightarrow ll(x),\ emp) \qquad U(x) \vdash lseg(x, p) \leadsto (U(x) \Rightarrow lseg(x, q) * U_2(q),\ U_2(p))$$

The first predicate $ll(x)$ did not have a continuation field. Hence, we did not generate any extra unknown predicate. The second predicate $lseg(x, p)$ did have a continuation field p, and we generated an extra unknown predicate $U_2(p)$ to capture a possible extension of the data structure beyond this continuation field.

Fold. A second scenario that must be handled by second-order entailment involves unknown predicates in the consequent. For each unknown predicate $U_1(\bar{v}, \bar{w}\#)$ in the consequent, a corresponding assumption $\Delta \Rightarrow U_1(\bar{v}, \bar{w}\#)\ @\ \Delta_g$ is inferred where Δ contains unknown predicates with at least one instantiatable parameters from $\bar{v}$, or heaps *reachable* from $\bar{v}$ (via either any data fields or parameters of known predicates) but stopping at non-instantiatable variables $\bar{w}\#$; a residual frame is also inferred from the antecedent (but added with pure approximation of footprint heaps [9]). For example, consider the following entailment:

$$\mathtt{x}{\mapsto}\mathtt{snode(q)}{*}\mathtt{q}{\mapsto}\mathtt{snode(NULL)}{\wedge}\mathtt{q}{\neq}\mathtt{NULL} \vdash \mathtt{U_1(x, q\#)} \rightsquigarrow (\mathtt{A_{f1}},\ \Delta_1)$$

The output of this entailment is:

$$\mathtt{A_{f1}} \equiv \mathtt{x}{\mapsto}\mathtt{snode(q)}{\wedge}\mathtt{q}{\neq}\mathtt{NULL}{\Rightarrow}\mathtt{U_1(x, q\#)} \quad \Delta_1 \equiv \mathtt{q}{\mapsto}\mathtt{snode(NULL)}{\wedge}\mathtt{x}{\neq}\mathtt{NULL}{\wedge}\mathtt{x}{\neq}\mathtt{q}$$

As a comparison, let us consider the scenario where $\mathtt{q}$ is unannotated, as follows:

$$\mathtt{x}{\mapsto}\mathtt{snode(q)}{*}\mathtt{q}{\mapsto}\mathtt{snode(NULL)}{\wedge}\mathtt{q}{\neq}\mathtt{NULL} \vdash \mathtt{U_1(x, q)} \rightsquigarrow (\mathtt{A_{f2}},\ \Delta_2)$$

In this case, the output of the entailment becomes:

$$\mathtt{A_{f2}} \equiv \mathtt{x}{\mapsto}\mathtt{snode(q)}{*}\mathtt{q}{\mapsto}\mathtt{snode(NULL)} \Rightarrow \mathtt{U_1(x, q)} \quad \Delta_2 \equiv \mathtt{x}{\neq}\mathtt{NULL}{\wedge}\mathtt{q}{\neq}\mathtt{NULL}{\wedge}\mathtt{x}{\neq}\mathtt{q}$$

Moreover, the folding process also captures *known* heaps that are reachable from #-parameters as *heap guard conditions*, e.g. $\mathtt{x}{\mapsto}\mathtt{node(q, x_n)}$ in our running example ($\mathtt{E3}$):

$$\begin{array}{l}\mathtt{x}{\mapsto}\mathtt{node(q,x_n)}{*}\mathtt{H_p(x_p,q\#)}{*}\mathtt{H_n(x_n,q\#)}{\wedge}\mathtt{x}{\neq}\mathtt{NULL} \vdash \mathtt{H(x_n, x\#)} \\ \quad \rightsquigarrow (\mathtt{H_n(x_n, q\#)} \Rightarrow \mathtt{H(x_n, x\#)}\ \mathtt{@}\ \mathtt{x}{\mapsto}\mathtt{node(q, x_n)},\ \mathtt{x}{\mapsto}\mathtt{node(q,x_n)}{*}\mathtt{H_p(x_p,q\#)}{\wedge}\mathtt{x}{\neq}\mathtt{NULL}) \quad (\mathtt{E3}')\end{array}$$

Such heap guards help with capturing the relations of heap structures and recovering those relationships when necessary (e.g. back-pointer $\mathtt{x\#}$).

Formalism. Bi-abductive unfold is formalized in Fig. 3. Here, $\mathtt{slice}(\bar{w}, \pi)$ is an auxiliary function that existentially quantifies in π all free variables that are not in the set $\bar{w}$.

$$\frac{\begin{array}{c}[\underline{\text{SO-ENT-UNFOLD}}] \\ \kappa_{\mathtt{s}} \equiv \mathtt{r}{\mapsto}\mathtt{c}(\bar{\mathtt{p}}) \text{ or } \kappa_{\mathtt{s}} \equiv \mathtt{P}(\mathtt{r}, \bar{\mathtt{p}}) \\ \kappa_{\mathtt{fields}} = *_{\mathtt{p_j}\in\bar{\mathtt{p}}}\ \mathtt{U_j}(\mathtt{p_j}, \bar{\mathtt{v}}_\mathtt{i}\#, \bar{\mathtt{v}}_\mathtt{n}\#),\ \ \text{where } \mathtt{U_j}\text{: fresh preds} \\ \kappa_{\mathtt{rem}} =\ \mathtt{U_{rem}}(\bar{\mathtt{v}}_\mathtt{i}, \bar{\mathtt{v}}_\mathtt{n}\#, \mathtt{r}\#),\ \ \text{where } \mathtt{U_{rem}}\text{: a fresh pred} \\ \pi_\mathtt{a} = \mathtt{slice}(\{\mathtt{r}, \bar{\mathtt{v}}_\mathtt{i}, \bar{\mathtt{v}}_\mathtt{n}, \bar{\mathtt{p}}\}, \pi_1) \quad \pi_\mathtt{c} = \mathtt{slice}(\{\bar{\mathtt{p}}\}, \pi_2) \\ \sigma \equiv (\mathtt{U}(\mathtt{r}, \bar{\mathtt{v}}_\mathtt{i}, \bar{\mathtt{v}}_\mathtt{n}\#) \wedge \pi_\mathtt{a} \Rightarrow \kappa_\mathtt{s} * \kappa_{\mathtt{fields}} * \kappa_{\mathtt{rem}} \wedge \pi_\mathtt{c}) \\ \kappa_1 * \kappa_{fields} * \kappa_{rem} \wedge \pi_1 \vdash \kappa_2 \wedge \pi_2 \rightsquigarrow (\mathcal{R},\ \Delta_R)\end{array}}{\mathtt{U}(r, \bar{v}_i, \bar{v}_n\#) {*} \kappa_1 {\wedge} \pi_1 \vdash \kappa_s {*} \kappa_2 {\wedge} \pi_2 \rightsquigarrow (\sigma{\wedge}\mathcal{R},\ \Delta_R)}$$

Fig. 3. Bi-Abductive Unfolding.

Thus it eliminates from π all subformulas not related to $\bar{w}$ (*e.g.* $\mathtt{slice}(\{x, q\}, q{=}\mathtt{NULL}{\wedge}y{>}3)$ returns $q{=}\mathtt{NULL}$). In the first line, a RHS assertion, either a points-to assertion $\mathtt{r}{\mapsto}\mathtt{c}(\bar{\mathtt{p}})$ or a known predicate instance $\mathtt{P}(\mathtt{r}, \bar{\mathtt{p}})$ is paired through the parameter $\mathtt{r}$ with the unknown predicate $\mathtt{U}$. Second, the unknown predicates $\mathtt{U_j}$ are generated for the data fields/parameters of $\kappa_\mathtt{s}$. Third, the unknown predicate $\mathtt{U_{rem}}$ is generated for the instantiatable parameters $\bar{\mathtt{v}}_\mathtt{i}$ of $\mathtt{U}$. The fourth and fifth lines compute relevant pure formulas and generate the assumption, respectively. Finally, the unknown predicates $\kappa_{\mathtt{fields}}$ and $\kappa_{\mathtt{rem}}$ are combined in the residue of LHS to continue discharging the remaining formula in RHS.

Bi-abductive fold is formalized in Fig. 4. The function $\mathtt{reach}(\bar{\mathtt{w}}, \kappa_1{\wedge}\pi_1, \bar{\mathtt{z}}\#)$ extracts portions from the antecedent heap (κ_1) that are (1) unknown predicates containing at least one instantiatable parameter from $\bar{\mathtt{w}}$; or (2) point-to or known predicates reachable from $\bar{\mathtt{w}}$, but not reachable from $\bar{\mathtt{z}}$. In our running example (the entailment ($\mathtt{E3}'$) on last page), the function $\mathtt{reach}(\{\mathtt{x_n}\}, \mathtt{x}{\mapsto}\mathtt{node(q, x_n)}{*}\mathtt{H_p(x_p, q\#)}{*}\mathtt{H_n(x_n, q\#)}{\wedge}\mathtt{x}{\neq}\mathtt{NULL}, \{\mathtt{x\#}\})$ is used to obtain $\mathtt{H_n(x_n, q\#)}$. More detail on this function is in the appendix. The $\mathtt{heaps}(\Delta)$ function enumerates all known predicate instances (of the form $\mathtt{P}(\bar{\mathtt{v}})$) and points-to instances (of the form $\mathtt{r}{\mapsto}\mathtt{c}(\bar{\mathtt{v}})$)) in Δ. The function *root*(κ) is defined as: *root*$(\mathtt{r}{\mapsto}\mathtt{c}(\bar{\mathtt{v}})){=}\{\mathtt{r}\}$, *root*$(\mathtt{P}(\mathtt{r}, \bar{\mathtt{v}})) = \{\mathtt{r}\}$. In the first line, heaps of LHS are separated into the assumption

$$\frac{\begin{array}{c}[\underline{\text{SO-ENT-FOLD}}] \\ \kappa_{11}{=}\mathtt{reach}(\bar{\mathtt{w}}, \kappa_1{\wedge}\pi_1, \bar{\mathtt{z}}\#) \quad \exists\kappa_{12} \cdot \kappa_1{=}\kappa_{11}{*}\kappa_{12} \\ \kappa_\mathtt{g} = *\{\kappa \mid \kappa{\in}\mathtt{heaps}(\kappa_{12}){\wedge}\mathit{root}(\kappa){\subseteq}\bar{\mathtt{z}}\} \quad \bar{\mathtt{r}}{=}\bigcup_{\kappa\in\kappa_\mathtt{g}} \mathit{root}(\kappa) \\ \sigma \equiv (\kappa_{11}{\wedge}\mathtt{slice}(\bar{\mathtt{w}}, \pi_1) \Rightarrow \mathtt{U_c}(\bar{\mathtt{w}}, \bar{\mathtt{z}}\#)\ \mathtt{@}\ \kappa_\mathtt{g}{\wedge}\mathtt{slice}(\bar{\mathtt{r}}, \pi_1)) \\ \kappa_{12} \wedge \pi_1 \vdash \kappa_2 \wedge \pi_2 \rightsquigarrow (\mathcal{R},\ \Delta_R)\end{array}}{\kappa_1 \wedge \pi_1 \vdash \mathtt{U_c}(\bar{w}, \bar{z}\#) * \kappa_2 \wedge \pi_2 \rightsquigarrow (\sigma{\wedge}\mathcal{R},\ \Delta_R)}$$

Fig. 4. Bi-Abductive Folding.

κ_{11} and the residue κ_{12}. Second, heap guards (and their root pointers) are inferred based on κ_{12} and the #-annotated parameters

$\bar{\mathtt{z}}$. The assumption is generated in the third line and finally, the residual heap is used to discharge the remaining heaps of RHS.

Hoare Rules. We shall now present Hoare rules to show how second-order entailment is used there. For simplicity, we consider a core imperative language (Fig. 5) that supports heap-based data structures (*datat*) and methods (*meth*).

$$\begin{array}{lll}
\mathtt{Prog} & ::= \mathit{datat}^* \ \mathit{meth}^* \quad \mathit{datat} \ ::= \mathtt{data}\, c\,\{\,\mathit{field}^*\,\} \\
\mathit{field} & ::= t\ v \quad t ::= \mathtt{int} \mid \mathtt{bool} \mid \mathtt{void} \mid \mathtt{c} \mid \ldots \\
\mathit{meth} & ::= t\ \mathit{mn}\ (([\mathtt{ref}]\ t\ v)^*)\ \ \Phi_{pr}\ \ \Phi_{po};\ \{e\} \\
e & ::= \mathtt{NULL} \mid k^{\tau} \mid v \mid v.f \mid v{=}e \mid v.f{=}e \mid \mathtt{new}\ c(v^*) \\
 & \quad \mid e_1; e_2 \mid t\ v;\ e \mid \mathit{mn}(v^*) \mid \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2
\end{array}$$

Fig. 5. The Core Language

A method declaration includes a header with pre-/post-condition and its body. Methods can have call-by-reference parameters (prefixed with `ref`). Loops, including nested loops, are transformed to tail-recursive methods with `ref` parameters to capture mutable variables. To support shape analysis, code verification is formalized as a proof of quadruple: $\vdash \{\Delta_{\mathtt{pre}}\}\ e\ \{\mathcal{R}, \Delta_{\mathtt{post}}\}$, where $\mathcal{R}$ accumulates the set of relational assumptions generated by the entailment procedure. The specification may contain unknown predicates in preconditions and postconditions. We list in Fig. 6 the rules for field access, method calls and method declaration. Note that primed variable (e.g. $\mathtt{x}'$) denotes the latest value (of the program variable $\mathtt{x}$). The formula $\Delta_1 *_{\bar{\mathtt{v}}} \Delta_2$ denotes $\exists \bar{\mathtt{r}} \cdot ([\bar{\mathtt{r}}/\bar{\mathtt{v}}']\Delta_1) * ([\bar{\mathtt{r}}/\bar{\mathtt{v}}]\Delta_2)$ (see [9]).

$$\frac{\begin{array}{c}\underline{\textbf{SA-CALL}} \\ \mathtt{t_0}\ \mathit{mn}\ ((\mathtt{ref}\ \mathtt{t_i}\ \mathtt{v_i})_{\mathtt{i=1}}^{\mathtt{m-1}}, (\mathtt{t_j}\ \mathtt{v_j})_{\mathtt{j=m}}^{\mathtt{n}})\ \ \Phi_{pr}\ \ \Phi_{po};\ \{e\} \in \mathtt{Prog} \\ \rho{=}[\mathtt{v}_\mathtt{k}'/\mathtt{v_k}]_{\mathtt{k=1}}^{\mathtt{n}} \quad \Phi_{pr}' = \rho(\Phi_{pr}) \quad \mathtt{W}{=}\{\mathtt{v_1}, .., \mathtt{v_{m-1}}\} \quad \mathtt{V}{=}\{\mathtt{v_m}, .., \mathtt{v_n}\} \\ \Delta \vdash \Phi_{pr}' \leadsto (\mathcal{R},\ \Delta_2) \quad \Delta_3{=}(\Delta_2 \wedge \bigwedge_{\mathtt{i=m}}^{\mathtt{n}}(\mathtt{v}_\mathtt{i}' = \mathtt{v_i}))\ \ *_{\mathtt{V}\cup\mathtt{W}}\ \Phi_{po}\end{array}}{\vdash \{\Delta\}\ \mathit{mn}(\mathtt{v_1}, .., \mathtt{v_{m-1}}, \mathtt{v_m}, .., \mathtt{v_n})\ \{\mathcal{R}, \Delta_3\}}$$

$$\frac{\begin{array}{c}\underline{\textbf{SA-FLD-RD}} \\ \mathtt{data}\ \mathtt{c}\ \{\mathtt{t_1}\ \mathtt{f_1}, .., \mathtt{t_n}\ \mathtt{f_n}\} \in \mathtt{Prog} \\ \Delta_1 \vdash \mathtt{x}' {\mapsto} \mathtt{c}(\mathtt{v_1..v_n}) \leadsto (\mathcal{R},\ \Delta_3) \\ \Delta_4{=}\exists \mathtt{v_1..v_n} \cdot (\Delta_3 {*} \mathtt{x}' {\mapsto} \mathtt{c}(\mathtt{v_1..v_n}) {\wedge} \mathtt{res}{=}\mathtt{v_i})\end{array}}{\vdash \{\Delta_1\}\ \mathtt{x.f_i}\ \{\mathcal{R}, \Delta_4\}} \qquad \frac{\begin{array}{c}\underline{\textbf{SA-METH}} \\ \vdash \{\Phi_{pr} {\wedge} \bigwedge(\mathtt{u}'{=}\mathtt{u})^*\}\ e\ \{\mathcal{R}_1, \Delta_1\} \\ \Delta_1 \vdash \Phi_{po} \leadsto (\mathcal{R}_2,\ \Delta_2) \\ \Gamma = \mathtt{solve}(\mathcal{R}_1 {\cup} \mathcal{R}_2)\end{array}}{\mathtt{t_0}\ \mathit{mn}\ ((\mathtt{t}\ \mathtt{u})^*)\ \ \Phi_{pr}\ \Phi_{po}\ \{e\}}$$

Fig. 6. Several Hoare Rules

The key outcome is that if a solution for the set of relational assumptions $\mathcal{R}$ can be found, the program is memory-safe and all the methods abide by their specifications. Furthermore, we propose a bottom-up verification process which is able to incrementally build suitable predicate instantiations one method at a time by solving the collected relational assumptions $\mathcal{R}$ progressively. The predicate definition synthesis (`solve`) consists of two separate operations : predicate synthesis, PRED_SYN, and predicate normalization, PRED_NORM. That is $\mathtt{solve}(\mathcal{R})$ = PRED_NORM(PRED_SYN($\mathcal{R}$)). After the method is successfully verified, the resulting predicate definitions Γ provide an interpretation for the unknown predicates appearing in the specifications such that memory safety is guaranteed. By returning Γ, the method verification allows the inferred definitions and specifications to be consistently reused in the verification of the remaining methods.

5 Derivation of Shape Predicates

Once the relational assumptions have been inferred, we proceed to apply a series of refinement steps to derive predicate definitions for each pre- and post-predicate. Fig. 7 outlines our strategy for predicate synthesis. We use the $\lfloor\textbf{syn-}*\rfloor$ notation to name refinement rules. For space reasons, we describe some rules and leave the rest to the appendix B. Steps that are left out include: (i) *sort-group* to decide on the transformation order of relational assumptions; (ii) $\lfloor\textbf{syn-Pre/Post-Obl}\rfloor$ to process some relational assumptions as proof obligations. For example, if the result of the recursive method is field-accessed after the recursive call, the post-predicate would appear as an unknown predicate for heap instantiation. This must be processed as an entailment obligation, after the definition of its post-predicate has been derived; (iii) *inline* to unfold synthesized predicates in the remaining assumptions.

```
function PRED_SYN( $\mathcal{R}$ )
  $\Gamma \leftarrow \emptyset$
  $\mathcal{R} \leftarrow$ exhaustively apply $\lfloor\textbf{syn-base}\rfloor$ on $\mathcal{R}$
  $\mathcal{R}_{\texttt{pre}}, \mathcal{R}_{\texttt{post}} \leftarrow$ sort-group($\mathcal{R}$)
  while $\mathcal{R}_{\texttt{pre}}{\neq}\emptyset$ do
    $\texttt{U}^{\texttt{pre}}, \sigma \leftarrow$ pick unknown & assumption in $\mathcal{R}_{\texttt{pre}}$
    $\texttt{U}^{\texttt{pre}}_{\texttt{def}} \leftarrow$ apply $\lfloor\textbf{syn-case}\rfloor$, $\lfloor\textbf{syn-group-pre}\rfloor$, and
        $\lfloor\textbf{syn-pre-def}\rfloor$ on $\sigma$
    $\mathcal{R}_{\texttt{pre}}, \mathcal{R}_{\texttt{post}} \leftarrow$ inline $\texttt{U}^{\texttt{pre}}_{\texttt{def}}$ in $(\mathcal{R}_{\texttt{pre}} \backslash \sigma)$, $\mathcal{R}_{\texttt{post}}$
    discharge $\texttt{U}^{\texttt{pre}}$ obligations
    $\Gamma \leftarrow \Gamma \cup \{\texttt{U}^{\texttt{pre}}_{\texttt{def}}\}$
  end while
  while $\mathcal{R}_{\texttt{post}}{\neq}\emptyset$ do
    $\texttt{U}^{\texttt{post}}, \sigma \leftarrow$ pick unknown & assumption in $\mathcal{R}_{\texttt{post}}$
    $\texttt{U}^{\texttt{post}}_{\texttt{def}} \leftarrow$ apply $\lfloor\textbf{syn-group-post}\rfloor$, $\lfloor\textbf{syn-post-def}\rfloor$ on $\sigma$
    discharge $\texttt{U}^{\texttt{post}}$ obligations
    $\mathcal{R}_{\texttt{post}} \leftarrow \mathcal{R}_{\texttt{post}} \setminus \sigma$        $\Gamma \leftarrow \Gamma \cup \{\texttt{U}^{\texttt{post}}_{\texttt{def}}\}$
  end while
  return $\Gamma$
end function
```

Fig. 7. Shape Derivation Outline

5.1 Base Splitting of Pre/Post-Predicates

We first deal with relational assumptions of the form $\texttt{U}^{\texttt{pre}}(\ldots){*}\Delta \Rightarrow \texttt{U}^{\texttt{post}}(\ldots)$, which capture constraints on both a pre-predicate and a post-predicate. To allow greater flexibility in applying specialized techniques for pre-predicates or post-predicates, we split the assumption into two assumptions such that pre-predicate $\texttt{U}^{\texttt{pre}}$ is separated from post-predicate $\texttt{U}^{\texttt{post}}$. Base splitting can be formalized as follows:

$$\frac{\begin{array}{c}\lfloor\textbf{syn-base}\rfloor\\ \sigma:\ \texttt{U}^{\texttt{pre}}(\bar{\texttt{x}}){*}\kappa{\wedge}\pi \Rightarrow \texttt{U}^{\texttt{post}}(\bar{\texttt{y}}) \quad \sigma_1:\ \texttt{U}^{\texttt{pre}}(\bar{\texttt{x}}){\wedge}\texttt{slice}(\bar{\texttt{x}},\pi){\Rightarrow}\texttt{emp} \quad \sigma_2:\ \kappa{\wedge}\pi \Rightarrow \texttt{U}^{\texttt{post}}(\bar{\texttt{y}})\\ \kappa_{\texttt{g}}{=}{*}\{\kappa_1 \mid \kappa_1{\in}\texttt{heaps}(\kappa){\wedge}\mathit{pars}(\kappa_1){\cap}\bar{\texttt{x}}{\neq}\emptyset\} \qquad \bar{\texttt{w}}{=}\bigcup\{\mathit{pars}(\kappa_1) \mid \kappa_1{\in}\kappa_{\texttt{g}}\}\\ \sigma_3:\ \texttt{U}^{\texttt{pre}}(\bar{\texttt{x}}){\Rightarrow}\texttt{U}^{\texttt{fr}}(\bar{\texttt{x}})\ @\ \kappa_{\texttt{g}}{\wedge}\texttt{slice}(\bar{\texttt{x}}{\cup}\bar{\texttt{w}},\pi) \qquad \sigma_4:\ \texttt{U}^{\texttt{fr}}(\bar{\texttt{x}}) \Rightarrow \top\end{array}}{\texttt{if is\_base}(\bar{\texttt{x}},\pi){=}\texttt{true then } (\sigma_1{\wedge}\sigma_2) \texttt{ else } (\sigma{\wedge}\sigma_3{\wedge}\sigma_4)}$$

The premise contains an assumption (σ) which could be split. The conclusion captures the new relational assumptions. There are two scenarios:
(1) The first scenario takes place when the test $\texttt{is\_base}(\bar{\texttt{x}},\pi)$ holds. It signifies that π contains a base case formula for some pointer(s) in $\bar{\texttt{x}}$. Note that $\texttt{is\_base}(\bar{\texttt{x}},\pi)$ holds if and only if $(\exists\ \texttt{v}{\in}\bar{\texttt{x}}.\ \pi \vdash \texttt{v}{=}\texttt{NULL})$ or $(\exists \texttt{v}_1,\texttt{v}_2{\in}\bar{\texttt{x}}.\pi \vdash \texttt{v}_1{=}\texttt{v}_2)$. In such a situation, the assumption σ is split into σ_1 and σ_2. This reflects the observation that a pre-predicate

guard will likely constrain the pre-predicate to a base-case with empty heap. This scenario happens in our running example where the assumption $(\mathtt{A1})$ is split to:

$$(\mathtt{A1a}).\ \mathtt{H(x,q)} \wedge \mathtt{x{=}NULL} \Rightarrow \mathtt{emp} \qquad (\mathtt{A1b}).\ \mathtt{emp} \wedge \mathtt{x{=}NULL} \Rightarrow \mathtt{G(x,q)}$$

(2) If the test $\mathtt{is\_base}(\bar{\mathtt{x}}, \pi)$ fails, there is no base case information available for us to instantiate $\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}})$. The assumption σ is not split and kept in the result. To have a more precise derivation, we would also record the fact that $\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}})$ has no instantiation under the current context. To do this, in the second line we record in $\kappa_{\mathtt{g}}$ such a heap context (related to $\bar{\mathtt{x}}$), extract in $\bar{\mathtt{w}}$ related pointers from the context, and introduce a fresh unknown predicate $\mathtt{U}^{\mathtt{fr}}$ as the instantiation for $\mathtt{U}^{\mathtt{pre}}$, as indicated by the assumption σ_3 in the third line. Note the heap guard specifies the context under which such an assumption holds. We also add σ_4 into the result, where the new predicate $\mathtt{U}^{\mathtt{fr}}$ is instantiated to the aforementioned memory locations (encapsulated by $\top$). Assumptions of the form $\mathtt{U}^{\mathtt{fr}}(\mathtt{p}) \Rightarrow \top$ are being used to denote dangling pointers. We also note that introducing the dangling predicate $\mathtt{U}^{\mathtt{fr}}$ into the guarded assumption σ_3 is essential to help relate non-traversed pointer fields between the pre-predicate $\mathtt{U}^{\mathtt{pre}}$ and the post-predicate $\mathtt{U}^{\mathtt{post}}$. The function *pars*$(\kappa)$ (the 2nd line) retrieves parameters: *pars*$(\mathtt{r}{\mapsto}\mathtt{c}(\bar{\mathtt{v}})) = \bar{\mathtt{v}}$, *pars*$(\mathtt{P}(\mathtt{r}, \bar{\mathtt{v}})) = \bar{\mathtt{v}}$.

As an example, consider splitting $(\sigma_5) : \mathtt{U}^{\mathtt{pre}}(\mathtt{p}){*}\mathtt{x}{\mapsto}\mathtt{node(p,n)}{\wedge}\mathtt{n{=}NULL} \Rightarrow \mathtt{U}^{\mathtt{post}}(\mathtt{x})$. The test $\mathtt{is\_base}(\{\mathtt{p}\}, \mathtt{n{=}NULL})$ fails. In addition to (σ_5), the splitting returns also

$$(\sigma_6) : \mathtt{U}^{\mathtt{pre}}(\mathtt{p}) \Rightarrow \mathtt{U}^{\mathtt{fr}}(\mathtt{p})\ @\ (\mathtt{x}{\mapsto}\mathtt{node(p,n)}{\wedge}\mathtt{n{=}NULL}) \qquad (\sigma_7) : \mathtt{U}^{\mathtt{fr}}(\mathtt{p}) \Rightarrow \top$$

For the $\mathtt{tll}$ example in Sec 3, the $[\underline{\textbf{syn-base}}]$ transformation can be applied to assumption (4) yielding the following three new assumptions:

$$(4a)\ \mathtt{res}{\mapsto}\mathtt{tree(p,l,r,t)}{*}\mathtt{H_l(l,p,t)}{\wedge}\mathtt{r{=}NULL}{\wedge}\mathtt{res{=}x} \Rightarrow \mathtt{G(x,p,res,t)}$$
$$(4b)\ \mathtt{H_r(r,p,t)} \wedge \mathtt{r{=}NULL} \Rightarrow \mathtt{emp}$$
$$(4c)\ \mathtt{H_l(l,p,t)} \Rightarrow \mathtt{H^f_l(l,p,t)}\ @\ (\mathtt{res}{\mapsto}\mathtt{tree(p,l,r,t)}{\wedge}\mathtt{r{=}NULL})$$
$$(4d)\ \mathtt{H^f_l(l,p,t)} \Rightarrow \top$$

Pre-predicate $\mathtt{H_r}$ captures $\mathtt{r{=}NULL}$ as its base-case split. Pre-predicate $\mathtt{H_l}$ uses a heap guard for its base context, and $\top$ to denote an un-accessed dangling heap residue encapsulated in $\mathtt{H^f_l}$.

5.2 Deriving Pre-Predicates

Pre-predicates typically appear in relational assumptions under pure guards π, of the form $\mathtt{U}^{\mathtt{pre}}(\ldots){\wedge}\pi \Rightarrow \Delta$. To derive definitions for these pre-predicates, the first step is to transform relational assumptions that overlap on their guards by forcing a case analysis that generates a set of relational assumptions with disjoint guard conditions:

$$\frac{\begin{array}{c}[\underline{\textbf{syn-case}}]\\ \mathtt{U}(\bar{x}){\wedge}\pi_1{\Rightarrow}\Delta_1\ @\ \Delta_{1g} \quad \mathtt{U}(\bar{x}){\wedge}\pi_2{\Rightarrow}\Delta_2\ @\ \Delta_{2g} \quad \pi_1{\wedge}\pi_2{\not\Rightarrow}\mathtt{FALSE} \\ \Delta_1{\wedge}\Delta_2{\Rightarrow}^{\bar{x}}_{\wedge}\Delta_3 \quad \Delta_{1g}{\wedge}\Delta_{2g}{\Rightarrow}^{\bar{x}}_{\wedge}\Delta_{3g} \quad \mathtt{SAT}(\Delta_{3g})\end{array}}{\mathtt{U}(\bar{x}){\wedge}\pi_1{\wedge}\neg\pi_2{\Rightarrow}\Delta_1\ @\ \Delta_{3g} \qquad \mathtt{U}(\bar{x}){\wedge}\pi_2{\wedge}\neg\pi_1{\Rightarrow}\Delta_2\ @\ \Delta_{3g} \qquad \mathtt{U}(\bar{x}){\wedge}\pi_1{\wedge}\pi_2{\Rightarrow}\Delta_3\ @\ \Delta_{3g}}$$

For brevity, we assume a renaming of free variables to allow $\bar{\mathtt{x}}$ to be used as arguments in both assumptions. Furthermore, we use the $\Rightarrow^{\bar{\mathtt{x}}}_{\wedge}$ operator to denote a normalization of overlapping conjunction, $\Delta_1{\wedge}\Delta_2$ [27]. Informally, in order for $\Delta_1 \wedge \Delta_2$ to hold, it is necessary that the shapes described by Δ_1 and Δ_2 agree when describing the same memory locations. Normalization thus determines the overlapping locations, Δ_c such that $\Delta_1{=}\Delta_c{*}\Delta_1'$ and $\Delta_2{=}\Delta_c{*}\Delta_2'$ and returns $\Delta_c{*}\Delta_1'{*}\Delta_2'$. We leave a formal definition of $\Rightarrow_{\wedge}^{\bar{x}}$ to the appendix B.5. Once all the relational assumptions for a given pre-predicate have been transformed such that the pure guards do not overlap, we may proceed to combine them using the rule $\lfloor$**syn-group-pre**$\rfloor$ shown below. We shall perform this exhaustively until a single relational assumption for U is derived. If the assumption RHS is independent of any post-predicate, it becomes the unknown pre-predicate definition, as shown in the rule $\lfloor$**syn-pre-def**$\rfloor$ below.

$$\frac{\texttt{U}(\bar{x}){\wedge}\pi_1 \Rightarrow \Phi_1^g \quad \texttt{U}(\bar{x}){\wedge}\pi_2 \Rightarrow \Phi_2^g \quad \pi_1{\wedge}\pi_2 \Rightarrow \texttt{FALSE}}{\texttt{U}(\bar{x}) \wedge (\pi_1{\vee}\pi_2) \Rightarrow \Phi_1^g{\wedge}\pi_1 \vee \Phi_2^g{\wedge}\pi_2}\ \lfloor\textbf{syn-group-pre}\rfloor \qquad \frac{\texttt{U}^{\texttt{pre}}(\bar{x}){\Rightarrow}\Phi^g \qquad \texttt{no\_post}(\Phi^g)}{\texttt{U}^{\texttt{pre}}(\bar{x}) \equiv \Phi^g}\ \lfloor\textbf{syn-pre-def}\rfloor$$

For the sll2dll example, applying the $\lfloor$**syn-group-pre**$\rfloor$ rule to (A2) and (A1a) yields:

$$(\texttt{A5}).\ \texttt{H}(\texttt{x},\texttt{q}) \Rightarrow \texttt{x}{\mapsto}\texttt{node}(\texttt{x}_\texttt{p},\texttt{x}_\texttt{n}){*}\texttt{H}_\texttt{p}(\texttt{x}_\texttt{p},\texttt{q}){*}\texttt{H}_\texttt{n}(\texttt{x}_\texttt{n},\texttt{q}) \ \vee\ \texttt{emp} \wedge \texttt{x}{=}\texttt{NULL}$$

For the tll example, by the above rules, (2) and (4b) yield:

$$(6).\ \texttt{H}_\texttt{r}(\texttt{r,p,t}) \equiv \texttt{H}(\texttt{r,x,t}){\wedge}\texttt{r}{\neq}\texttt{NULL}\ @\ \texttt{x}{\mapsto}\texttt{tree}(\texttt{p,l,r,n}) \vee \texttt{emp}{\wedge}\texttt{r}{=}\texttt{NULL}$$

Similarly, (3) and (4c) derives $\texttt{H}_\texttt{l}$:.

$$\begin{array}{l}(7).\ \texttt{H}_\texttt{l}(\texttt{l,p,t}) \equiv \texttt{H}(\texttt{l,x,lm})\ @\ \texttt{x}{\mapsto}\texttt{tree}(\texttt{p,l,r,n}){\wedge}\texttt{r}{\neq}\texttt{NULL}\\ \qquad\qquad\qquad \vee\ \texttt{H}_\texttt{l}^\texttt{f}(\texttt{l,p,t})\ @\ \texttt{x}{\mapsto}\texttt{tree}(\texttt{p,l,r,n}){\wedge}\texttt{r}{=}\texttt{NULL}\end{array}$$

This is then trivially converted into a definition for its pre-predicate, without any weakening, thus ensuring soundness of our pre-conditions.

5.3 Deriving Post-Predicates

We start the derivation for a post-predicate after all pre-predicates have been derived. We can incrementally group each pair of relational assumptions on a post-predicate via the $\lfloor$**syn-group-post**$\rfloor$ rule shown below. By exhaustively applying $\lfloor$**syn-group-post**$\rfloor$ rule all assumptions relating to predicate $\texttt{U}^{\texttt{post}}$ get condensed into an assumption of the form: $\Delta_1 \vee \ldots \vee \Delta_n \Rightarrow \texttt{U}^{\texttt{post}}(\bar{x})$. This may then be used to confirm the post-predicate by generating the predicate definition via the $\lfloor$**syn-post-def**$\rfloor$ rule.

$$\frac{\Delta_a \Rightarrow \texttt{U}^{\texttt{post}}(\bar{x}) \qquad \Delta_b \Rightarrow \texttt{U}^{\texttt{post}}(\bar{x})}{\Delta_a \vee \Delta_b \Rightarrow \texttt{U}^{\texttt{post}}(\bar{x})}\ \lfloor\textbf{syn-group-post}\rfloor \qquad \frac{\Delta_1 \vee \ldots \vee \Delta_n \Rightarrow \texttt{U}^{\texttt{post}}(\bar{x})}{\texttt{U}^{\texttt{post}}(\bar{x})\ \equiv\ \Delta_1 \vee \ldots \vee \Delta_n}\ \lfloor\textbf{syn-post-def}\rfloor$$

Using these rules, we can combine (A4) and (A1b) in the sll2dll example to obtain:

$$\texttt{G}(\texttt{x},\texttt{q})\ \equiv\ \texttt{emp} \wedge \texttt{x}{=}\texttt{NULL}\ \vee\ \texttt{x}{\mapsto}\texttt{node}(\texttt{q},\texttt{x}_\texttt{n}) * \texttt{G}(\texttt{x}_\texttt{n},\texttt{x})$$

Using these rules, we can combine (4a) and (5) in the tll example to give:

$$\begin{array}{l}\texttt{G}(\texttt{x,p,res,t}) \equiv \texttt{x}{\mapsto}\texttt{tree}(\texttt{p,l,r,t}){*}\texttt{H}_\texttt{l}(\texttt{l},\texttt{x}_\texttt{h},\texttt{t}_\texttt{h}) \wedge \texttt{res}{=}\texttt{x}{\wedge}\texttt{r}{=}\texttt{NULL}\\ \vee \texttt{x}{\mapsto}\texttt{tree}(\texttt{p,l,r,n}){*}\texttt{G}(\texttt{r,x,l}_\texttt{m}\texttt{,t}){*}\texttt{G}(\texttt{l,x,res,l}_\texttt{m}){*}\texttt{H}_\texttt{n}(\texttt{n},\texttt{x}_\texttt{h},\texttt{t}_\texttt{h}){\wedge}\texttt{r}{\neq}\texttt{NULL}\end{array}$$

5.4 Predicate Normalization for Concise Definitions

After we have synthesized suitable predicate definitions, we proceed with predicate normalization to convert each predicate definition to its most concise form. Our current method, PRED_NORM, uses four key steps: (i) eliminate dangling predicates, (ii) eliminate useless parameters, (iii) re-use predicate definitions and (iv) perform predicate splitting. We briefly explain the normalization steps and leave details in the appendix B.7. The first step deals with dangling predicates which do not have any definition. Though it is safe to drop such predicates (by frame rule), our normalization procedure replaces them by special variables, to help capture linking information between pre- and post-conditions. The second step eliminates predicate arguments that are not used in their synthesized definitions, with the help of second-order entailment. The third step leverage on our entailment procedure to conduct an equivalence proof to try to match a newly inferred definition with a definition previously provided or inferred. Lastly, to increase the chance for such predicate reuse, we allow predicates to be split into smaller predicates. This is again done with the help of second-order entailment procedure, allowing us to undertake such normalization tasks soundly and easily.

6 Soundness Lemmas and Theorem

Here we briefly state several key soundness results, and leave the proof details to the appendix E. For brevity, we introduce the notation $\mathcal{R}(\Gamma)$ to denote a set of predicate instantiations $\Gamma{=}\{U_1(\bar{v}_1){\equiv}\Delta_1, ..U_n(\bar{v}_n){\equiv}\Delta_n\}$ satisfying the set of assumptions $\mathcal{R}$. That is, for all assumptions $\Delta \Rightarrow \Phi^g \in \mathcal{R}$, (i) Γ contains a predicate instantiation for each unknown predicate appearing in Δ and Φ^g; (ii) by interpreting all unknown predicates according to Γ, then it is provable that Δ implies Φ^g, written as $\Gamma : \Delta \vdash \Phi^g$.

Soundness of bi-abductive entailment. Abduction soundness requires that if all the relational assumptions generated are satisfiable, then the entailment is valid.

Lemma 1. *Given the entailment judgement* $\Delta_a \vdash \Delta_c \rightsquigarrow (\mathcal{R},\ \Delta_f)$*, if there exists* Γ *such that* $\mathcal{R}(\Gamma)$*, then the entailment* $\Gamma : \Delta_a \vdash \Delta_c * \Delta_f$ *holds.*

Derivation soundness. For derivation soundness, if a set of predicate definitions is constructed then those definitions must satisfy the initial set of assumptions. We argue that (i) assumption refinement does not introduce spurious instantiations, (ii) the generated predicates satisfy the refined assumptions, (iii) normalization is meaning preserving.

Lemma 2. *Given a set of relational assumptions* $\mathcal{R}$*, let* $\mathcal{R}'$ *be the set obtained by applying any of the refinement steps, then for any* Γ *such that* $\mathcal{R}'(\Gamma)$*, we have* $\mathcal{R}(\Gamma)$.

Lemma 3. *If* $\mathcal{R}$ *contains only one pre-assumption on predicate* U^{pre},$U^{pre}(\bar{v}){\Rightarrow}\Phi^g$ *and if our algorithm returns a solution* Γ*, then* $(U^{pre}(\bar{v}){\equiv}\Phi^g){\in}\ \Gamma$*. Similarly, if* $\mathcal{R}$ *has a sole post-assumption on* U^{post}*,* $\Phi{\Rightarrow}U^{post}$ *and if solution* Γ *is returned, then* $(U^{post}(\bar{v}){\equiv}\Phi){\in}\ \Gamma$.

Lemma 4. *Given a set of assumptions* $\mathcal{R}$*, if* PRED_SYN($\mathcal{R}$) *returns a solution* Γ *then* $\mathcal{R}(\Gamma)$*. Furthermore, if* PRED_NORM(Γ) *returns a solution* Γ' *then* $\mathcal{R}(\Gamma')$.

Theorem 6.1 (Soundness) *If* $\Delta_a \vdash \Delta_c \rightsquigarrow (\mathcal{R},\ \Delta_f)$ *and* Γ=PRED_NORM(PRED_SYN($\mathcal{R}$)) *then* $\Gamma : \Delta_a \vdash \Delta_c * \Delta_f$.

7 Implementation and Experimental Results

We have implemented the proposed shape analysis within HIP [9], a separation logic verification system. The resulting verifier, called S2, uses an available CIL-based [26] translator [3] from C to the expression-oriented core language. Our analysis modularly infers the pre/post specification for each method. It attempts to provide the weakest possible precondition to ensure memory safety (from null dereferencing and memory leaks), and the strongest possible post-condition on heap usage patterns, where possible.
Expressivity. We have explored the generality and efficiency of the proposed analysis through a number of small but challenging examples. We have evaluated programs which manipulate lists, trees and combinations (e.g. `tll`: trees whose leaves are chained in a linked list). The experiments were performed on a machine with the Intel i7-960 (3.2GHz) processor and 16 GB of RAM. Table 1 presents our experimental results. For each test, we list the name of the manipulated data structure and the effect of the verified code under the `Example` column. Here we used SLL,DLL,CLL,CDLL for singly-, doubly-, cyclic-singly-, cyclic-doubly- linked lists. SLL + head/tail for an SLL where each element points to the SLL's head/tail. SLL of 0/1 SLLs uses an SLL nested in a SLL of size 0 or 1, CSLL of CSLLs for CSLL nested in CSLL, CDLL of 5CSLLs for an CDLL where each node is a source of five CSLL, and CDLL of CSLLs$_2$ for CDLL where each node is a nested CSLL. The skip lists subscript denotes the number of skip pointers. The swl procedure implements list traversal following the DeutschSchorr-Waite style. `rose-trees` are trees with nodes that are allowed to have variable number of children, typically stored as linked lists, and `mcf` trees [16] are rose-tree variants where children are stored in doubly-linked lists with sibling and parent pointers. In order to

Example	w/o norm.		w/ norm.		Veri.	Example	w/o norm.		w/ norm.		Veri.
	size	Syn.	size	Syn.			size	Syn.	size	Syn.	
SLL (delete)	9	0.23	2	0.29	0.22	CSLL (t)	8	0.22	5	0.23	0.24
SLL (reverse)	20	0.21	8	0.22	0.2	CSLL of CSLLs (c)	18	0.24	4	0.23	0.22
SLL (insert)	13	0.2	11	0.21	0.2	SLL2DLL	18	0.19	2	0.2	0.18
SLL (setTail)	7	0.16	2	0.18	0.16	DLL (check)	8	0.21	2	0.23	0.19
SLL (get-last)	20	0.7	17	0.75	0.21	DLL (append)	11	0.2	8	0.2	0.2
SLL-sorted (c)	11	0.26	2	0.27	0.22	CDLL (c)	23	0.22	8	0.26	0.21
SLL (bubblesort)	13	0.28	9	0.36	0.26	CDLL of 5CSLLs (c)	28	0.39	4	0.66	1.3
SLL (insertsort)	15	0.3	11	0.3	0.27	CDLL of CSLLs$_2$ (c)	29	0.33	4	0.44	0.29
SLL (zip)	20	0.27	2	0.32	0.24	btree (search)	33	0.23	2	0.24	0.23
SLL-zip-leq	20	0.27	2	0.27	0.25	btree-parent (t)	11	0.23	2	0.29	0.24
SLL + head (c)	12	0.24	2	0.71	0.2	rose-tree (c)	14	0.28	14	0.3	0.23
SLL + tail (c)	10	0.19	2	0.72	0.18	swl (t)	19	0.23	13	0.27	22
skip-list$_2$ (c)	9	0.28	1	0.32	0.25	mcf (c)	19	0.26	17	0.28	0.26
skip-list$_3$ (c)	9	0.36	1	0.46	0.3	tll (t)	21	0.23	2	0.25	0.21
SLL of 0/1 SLLs	8	0.25	1	0.26	0.23	tll (c)	21	0.29	2	0.32	0.19
CSLL (c)	17	0.18	2	0.23	0.21	tll (set-parent)	39	0.24	2	0.35	0.24

Table 1. Experimental Results (**c** for *check* and **t** for *traverse*)

[3] Our translation preserves the semantics of source programs, subject to CIL's limitations.

evaluate the performance of our shape synthesis, we re-verified the source programs against the inferred specifications and listed the verification time (in seconds) in the `Veri.` column and the synthesis times in column `Syn.`. In total, the specification inference took 8.37s while the re-verification[4] took 8.25s.

The experiments showed that our tool can handle fairly complex recursive methods, like trees with linked leaves. It can synthesize shape abstractions for a large variety of data structures; from list and tree variants to combinations. Furthermore, the tool can infer shapes with mutual-recursive definitions, like the `rose-trees` and `mcf` trees.

The normalization phase aims to simplify inferred shape predicates. To evaluate its effectiveness, we performed the synthesis on two scenarios: without (w/o) and with (w/) normalization. The number of conjuncts in the synthesized shapes is captured with *size* column. The results show that normalization is helpful; it reduces by 68% (169/533) the size of synthesized predicates with a time overhead of 27% (8.37s/10.62s).

Larger Experiments. We evaluated S2 on real source code from the Glib open source library [1]. Glib is a cross-platform C library including non-GUI code from the GTK+ toolkit and the GNOME desktop environment. We focused our experiments on the files which implemented heap data structures, i.e. SLL (gslist.c), DLL (glist.c), balanced binary trees (gtree.c) and N-ary trees (gnode.c). In Fig.8 we list for each file number of lines of code (excluding comments) LOC, number of procedures (while/for loops) #Proc (#Loop). $\#\surd$ describes the number of procedures and loops for which S2 inferred specifications that guarantee memory safety. S2 can infer specifications that guarantee memory safety for 89% of procedures and loops (192/216).[5]

	LOC	#Proc	#Loop	$\#\surd$	Syn. (sec)
gslist.c	698	33	18	47	11.73
glist.c	784	35	19	49	7.43
gtree.c	1204	36	14	44	3.69
gnode.c	1128	37	27	52	16.34

Fig. 8. Experiments on Glib Programs

Limitations. Our present proposal cannot handle graphs and overlaid data structures since our instantiation mechanism always expands into tree-like data structures with back pointers. This is a key limitation of our approach. For an example, of such a limitation, see appendix F. For future work, we also intend to combine shape analysis with other analyses domains, in order to capture more expressive specifications, beyond memory safety.

8 Related Work and Conclusion

A significant body of research has been devoted to shape analysis. Most proposals are orthogonal to our work as they focus on determining shapes based on a fixed set of shape domains. For instance, the analysis in [25] can infer shape and certain numerical properties but is limited to the linked list domain. The analyses from [31,11,4,15,3,23] are tailored to variants of lists and a fixed family of list interleavings. Likewise, Calcagno

[4] Due to our use of sound inference mechanisms, re-verification is not strictly required. We perform it here to illustrate the benefit of integrating inference within a verification framework.

[5] Our current implementation does not support array data structures. Hence, some procedures like `g_tree_insert_internal` cannot be verified.

et al. [7] describes an analysis for determining lock invariants with only linked lists. Lee et al. [21] presents a shape analysis specifically tailored to overlaid data structures. In the matching logic framework, a set of predicates is typically assumed for program verification [30]. The work [2] extends this with specification inference. However, it currently does not deal with the inference of inductive data structure abstractions.

The proposal by Magill et al. [25] is able to infer numerical properties, but it is still parametric in the shape domain. Similarly, the separation logic bi-abduction described in [6,17] assumes a set of built-in or user-defined predicates. Xisa, a tool presented by Rival et. al. [8], works on programs with more varied shapes as long as structural invariant checkers, which play the role of shape definitions, are provided. A later extension [29] also considers shape summaries for procedures with the additional help of global analysis. Other similarly parameterized analysis includes [13]. In comparison, our approach is built upon the foundation of second-order bi-abductive entailment, and is able to infer unknown predicates from scratch or guided by user-supplied assertions. This set-up is therefore highly flexible, as we could support a mix of inference and verification, due to our integration into an existing verification system.

With respect to fully automatic analyses, there are [5], [16] and the Forester system [18]. Although very expressive in terms of the inferred shape classes, the analysis proposed by Guo et al. [16] relies on a heavy formalism and depends wholly on the shape construction patterns being present in the code. They describe a global analysis that requires program slicing techniques to shrink the analyzed code and to avoid noise on the analysis. Furthermore, the soundness of their inference could not be guaranteed; therefore a re-verification of the inferred invariants is required. Brotherston and Gorogiannis [5] propose a novel way to synthesize inductive predicates by ensuring both memory safety and termination. However, their proposal is currently limited to a simple imperative language without methods. A completely different approach is presented in the Forrester system [18] where a fully automated shape synthesis is described in terms of graph transformations over forest automata. Their approach is based on learning techniques that can discover suitable forest automata by incrementally constructing shape abstractions called boxes. However, their proposal is currently restricted both in terms of the analysed programs, e.g. recursion is not yet supported, and in terms of the inferred shapes, as recursive nested boxes (needed by `tll`) are not supported.

In the TVLA tradition, [28] describes an interprocedural shape analysis for cut-free programs. The approach explores the interaction between framing and the reachability-based representation. Other approaches to shape analysis include grammar-based inference, e.g. [22] which relies on inferred grammars to define the recursive backbone of the shape predicates. Although [22] is able to handle various types of structures, e.g. trees and dlls, it is limited to structures with only one argument for back pointers. [24] employs inductive logic programming (ILP) to infer recursive pure predicates. While, it might be possible to apply a similar approach to shape inference, there has not yet been any such effort. Furthermore, we believe a targeted approach would be able to easily cater for the more intricate shapes. Since ILP has been shown to effectively synthesize recursive predicates, it would be interesting to explore an integration of ILP with our proposal for inferring recursive predicates of both shape and pure properties. A recent work [14] that aims to automatically construct verification tools has implemented various proof rules for reachability and termination properties however it does not focus on the synthesis of shape abstractions. In an orthogonal direction, [10] presents an analysis for constructing precise and compact method summaries. Unfortunately, both these works lack the ability to handle recursive data structures.

Conclusion We have presented a novel approach to *modular* shape analysis that can automatically synthesize, from scratch, a set of shape abstractions that are needed for ensuring memory safety. This capability is premised on our decision to build *shape predicate inference* capability directly into a new second-order bi-abductive entailment procedure. Second-order variables are placeholders for unknown predicates that can be synthesized from proof obligations gathered by Hoare-style verification. Thus, the soundness of our inference is based on the soundness of the entailment procedure itself, and is not subjected to a re-verification process. Our proposal for shape analysis has been structured into three key stages: (i) gathering of relational assumptions on unknown shape predicates; (ii) synthesis of predicate definitions via derivation; and (iii) normalization steps to provide concise shape definitions. We have also implemented a prototype of our inference system into an existing verification infrastructure, and have evaluated on a range of examples with complex heap usage patterns.

Acknowledgement We thank Quang-Trung Ta for his C front-end integration. We gratefully acknowledge the support of research grant MOE2013-T2-2-146.

References

1. Glib-2.38.2. https://developer.gnome.org/glib/, 2013. [Online; accessed 13-Nov-2013].
2. M. Alpuente, M. A. Feliú, and A. Villanueva. Automatic inference of specifications using matching logic. In *PEPM*, pages 127–136, 2013.
3. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192, 2007.
4. J. Berdine, B. Cook, and S. Ishtiaq. SLAYER: memory safety for systems-level code. In *CAV*, pages 178–183, 2011.
5. J. Brotherston and N. Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. Technical Report RN/13/14, University College London, 2013.
6. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.
7. C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive resource invariant synthesis. In *APLAS*, pages 259–274, 2009.
8. B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.
9. 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. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.
10. I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–577, 2011.
11. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.
12. 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.
13. A. Gotsman, J. Berdine, and B. Cook. Interprocedural Shape Analysis with Separated Heap Abstractions. In *SAS*, pages 240–260, 2006.

14. S. Grebenshchikov, Nuno P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.
15. B. S. Gulavani, S. Chakraborty, S. Ramalingam, and A. V. Nori. Bottom-up shape analysis. In *SAS*, pages 188–204, 2009.
16. B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *ACM PLDI*, pages 256–265, 2007.
17. G. He, S. Qin, W.-N Chin, and F. Craciun. Automated specification discovery via user-defined predicates. In *ICFEM*, 2013.
18. L. Holik, O. Lengál, A. Rogalewicz, J. Simácek, and T. Vojnar. Fully automated shape analysis based on forest automata. In *CAV'13*, pages 740–755, 2013.
19. R. Iosif, A. Rogalewicz, and J. Simácek. The tree width of separation logic with recursive definitions. In *CADE*, pages 21–38, 2013.
20. S. Ishtiaq and P. W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *ACM POPL*, London, January 2001.
21. O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *CAV*, 2011.
22. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, pages 124–140, 2005.
23. T. Lev-Ami, M. Sagiv, T. Reps, and S. Gulwani. Backward analysis for inferring quantified preconditions. Technical Report TR-2007-12-01, Tel Aviv University, 2007.
24. Alexey Loginov, Thomas Reps, and Mooly Sagiv. Abstraction refinement via inductive learning. In Kousha Etessami and SriramK. Rajamani, editors, *CAV*, volume 3576, pages 519–533. 2005.
25. S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, pages 211–222, 2010.
26. G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC'02*, pages 213–228, 2002.
27. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, pages 55–74, 2002.
28. N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS*, pages 284–302, 2005.
29. X. Rival and B.-Y. E. Chang. Calling context abstraction with shapes. In *POPL*, pages 173–186, 2011.
30. G. Rosu and A. Stefanescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012.
31. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.

A Second-Order Bi-Abduction: Auxiliary functions

$\mathtt{reach}(\mathcal{S}, \kappa{\wedge}\pi, \mathcal{I})$ "carves" from κ the abstractions of the memory locations reachable from pointer variables in the set $\mathcal{S}$ without traversing variables in $\mathcal{I}$. We also locate a set of heap guards to help with the instantiation of parameters $\bar{\mathtt{w}}_{\#}$ that are not included in $\bar{\mathtt{x}}$ but are connected via heap locations. Guard formula $\kappa_{\mathtt{g}}{\wedge}\pi_{\mathtt{g}}$ is the formula that related to the mismatched RHS but has not been consumed by the assumption. Finally, we have:

$$\mathsf{reach}(\mathcal{S}, \mathtt{v}{\mapsto}\mathtt{c}(\bar{\mathtt{w}}){*}\kappa{\wedge}\pi, \mathcal{I}) =$$
$$\begin{cases} \mathtt{v}{\mapsto}\mathtt{c}(\bar{\mathtt{w}}){*}\mathsf{reach}(\mathcal{S}, \kappa{\wedge}\pi, \mathcal{I}) & \forall \mathtt{v_i}{\in}\{\mathtt{v}\}{\cup}\bar{\mathtt{w}} \cdot \mathtt{ptr}(\mathtt{v_i}) \Rightarrow \mathtt{bounded}(\mathtt{v_i}, \mathtt{v}{\mapsto}\mathtt{c}(\bar{\mathtt{w}}){*}\kappa{\wedge}\pi, \mathcal{I}) \\ \mathtt{P}(\bar{\mathtt{w}}){*}\mathsf{reach}(\mathcal{S}, \kappa \wedge \pi, \mathcal{I}) & \forall \mathtt{v_i} \in \bar{\mathtt{w}} \cdot \mathtt{ptr}(\mathtt{v_i}) \Rightarrow \mathtt{bounded}(\mathtt{v_i}, \mathtt{P}(\bar{\mathtt{w}}){*}\kappa \wedge \pi, \mathcal{I}) \\ \mathtt{emp} & \text{otherwise} \end{cases}$$

Similarly we define $\mathtt{bounded}(\mathtt{v},\kappa\wedge\pi,\mathcal{I})$ to hold iff the shape pointed to by $\mathtt{v}$ is precisely defined by the frontier $\mathcal{I}$. That is, either of the conditions below holds:

1. $\mathtt{v}$ is on the frontier: $\mathtt{v}\in\mathcal{I}$
2. $\mathtt{v}$ is null: $\pi\Rightarrow\mathtt{v}=\mathtt{NULL}$
3. $\mathtt{v}$ is a dangling variable
4. $\mathtt{v}{\mapsto}\mathtt{c}(\bar{\mathtt{w}})\in\kappa$ **and** $\forall\mathtt{w_i}\in\bar{\mathtt{w}}\cdot\mathtt{ptr}(\mathtt{w_i})\wedge\mathtt{bounded}(\mathtt{w_i},\kappa\wedge\pi,\mathcal{I}\cup\{\mathtt{v}\})$
5. $\mathtt{U}(\bar{\mathtt{w}})\in\kappa$, $\mathtt{v}\in\bar{\mathtt{w}}$ and a definition for predicate $\mathtt{U}$ has already been inferred.

B Further Refinement/Normalization Steps

B.1 Assumption Sorting and Partitioning

In order to allow for a convenient instantiation order, we sort and group together each set of relational assumptions pertaining to the same predicate, through *sort-group($\mathcal{R}$)*. Intuitively, the call to *sort-group($\mathcal{R}$)* will sort assumptions relevant to each predicate according to the following pattern:

1. $\mathtt{H}(\ldots)\wedge\pi\ @\ \Delta\Rightarrow\mathtt{H}_2(\ldots)$
2. $\mathtt{H}(\ldots)\Rightarrow\mathtt{H}_2(\ldots)$
3. $\mathtt{H}(\ldots)*\Delta\Rightarrow\mathtt{H}_2(\ldots)$
4. $\mathtt{H}(\ldots)\Rightarrow\Delta$
5. $\mathtt{H}(\ldots)\wedge\pi\Rightarrow\Delta$
6. $\Delta\Rightarrow\mathtt{G}(\ldots)$

The first four forms could be used directly for in-lining if they are not self-recursive. Guarded assumptions (even if disjunctive but not self-recursive) are given high priority to facilitate their early removal by inlining. The fifth form leads to disjunctive recursive formula, and is not inlined for conciseness reason. As mentioned, post-predicates are only processed after all the pre-predicates have been synthesized. In order to decrease the number of assumptions that need to be considered during the derivation, we will try to partition the assumptions relevant to a predicate into assumptions that will be used in the synthesise process and assumptions that will be treated as proof obligations to be discharged after the predicate synthesis. Note that we order pre-predicates before post-predicates, and would synthesize the simpler predicates, before the more complex ones. Also, the processing of post-obligation for a post-predicate is done after the synthesis of the respective post-predicate. This could generate extra assumptions for other unsythesised predicates.

B.2 Obligations for Pre-Predicates

On a similar note, newly created memory locations that were neither abstracted by a pre-predicate nor a post-predicate, could be passed as arguments to a recursive call. One example of an assumption resulting from such a scenario is:

$$\mathtt{x}{\mapsto}\mathtt{snode}(\mathtt{n})*\mathtt{H_n}(\mathtt{n})\Rightarrow\mathtt{H}(\mathtt{x})$$

Catering for such assumptions follows on a similar scenario as the one for post-predicate obligations: handling will be delayed until a definition for the $\mathtt{H}$ predicate has been synthesized which in turn can be used by our second order bi-abductive entailment to generate constraints for the remaining pre-predicates, e.g. $\mathtt{H_n}$. That is:

$$\frac{\begin{array}{c}[\textbf{syn-Pre-Obl}]\\ \Delta\Rightarrow\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{v}})\qquad(\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{v}}){\equiv}\Delta_{\mathtt{U}^{\mathtt{pre}}})\in\Gamma\qquad\Delta\vdash\Delta_{\mathtt{U}^{\mathtt{pre}}}\leadsto(\mathcal{R},\ \Delta_{\mathtt{f}})\end{array}}{\mathcal{R}}$$

B.3 Obligation for Post-Predicates

Memory locations abstracted by post-predicates may be further accessed after a recursive call. This may lead to relational assumptions of the following form.

$$\mathtt{U}^{\mathtt{post}}(\ldots) \wedge \pi \Rightarrow \varDelta$$

We regard this as an obligation that has to be proven, and at the same time it could also be used to infer the definition of unknown post-predicates that were generated. As an example, consider the following post-predicate relational obligation:

$$\mathtt{G}(\mathtt{r},\mathtt{x}){\wedge}\mathtt{x}{\neq}\mathtt{NULL} \Rightarrow \mathtt{r}{\mapsto}\mathtt{snode}(\mathtt{r_n}) * \mathtt{G_r}(\mathtt{r_n},\mathtt{x}\#){*}\mathtt{G_x}(\mathtt{x},\mathtt{r}\#)$$

This obligation introduces two extra unknown post-predicates $\mathtt{G_r}$ and $\mathtt{G_x}$. In order to synthesize definitions for them, it is possible to leverage on earlier synthesized definitions for $\mathtt{G}$. Thus, if such a definition exists, say:

$$\mathtt{G}(\mathtt{r},\mathtt{x}) \equiv \mathtt{x}{\mapsto}\mathtt{snode}(\mathtt{NULL}) \wedge \mathtt{r}{=}\mathtt{x} \;\vee\; \mathtt{x}{\mapsto}\mathtt{snode}(\mathtt{x_n}) * \mathtt{G_x}(\mathtt{x_n},\mathtt{r}) * \mathtt{r}{\mapsto}\mathtt{snode}(\mathtt{NULL})$$

We can re-use our second-order entailment to prove the earlier post-predicate obligation, which generates the following set of relational assumptions on $\mathtt{G_r}$ and $\mathtt{G_x}$.

$$\begin{array}{ll}
\mathtt{r}{=}\mathtt{NULL} & \Rightarrow \mathtt{G_r}(\mathtt{r},\_) \\
\mathtt{x}{=}\mathtt{r} \wedge \mathtt{x}{\neq}\mathtt{NULL} & \Rightarrow \mathtt{G_x}(\mathtt{x},\mathtt{r}) \\
\mathtt{x}{\mapsto}\mathtt{snode}(\mathtt{x_n}) * \mathtt{G_x}(\mathtt{x_n},\mathtt{r}) & \Rightarrow \mathtt{G_x}(\mathtt{x},\mathtt{r})
\end{array}$$

Subjecting them to synthesis for post-predicate yields:

$$\begin{array}{l}
\mathtt{G_r}(\mathtt{r},\_) \equiv \mathtt{r}{=}\mathtt{NULL} \\
\mathtt{G_x}(\mathtt{x},\mathtt{r}) \equiv \mathtt{x}{=}\mathtt{r} \wedge \mathtt{x}{\neq}\mathtt{NULL} \;\vee\; \mathtt{x}{\mapsto}\mathtt{snode}(\mathtt{x_n}) * \mathtt{G_x}(\mathtt{x_n},\mathtt{r})
\end{array}$$

More formally,

$$\frac{\begin{array}{c}[\underline{\textbf{syn-Post-Obl}}] \\ \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{v}}){\wedge}\pi \Rightarrow \varDelta \quad (\mathtt{U}^{\mathtt{post}}(\bar{\mathtt{v}}){\equiv}\varDelta_{\mathtt{U}^{\mathtt{post}}}) \in \varGamma \quad \varDelta_{\mathtt{U}^{\mathtt{post}}}{\wedge}\pi \vdash \varDelta \leadsto (\mathcal{R},\ \varDelta_{\mathtt{f}})\end{array}}{\mathcal{R}}$$

B.4 Predicate Inlining

As derived predicates may contain heap guards, we may remove them by inlining predicate occurrences with the relevant heap context.

$$\frac{\begin{array}{c}[\underline{\textbf{syn-inline}}] \\
\mathtt{U}^{\mathtt{pre}}_{\mathtt{d}}(\bar{x}) \equiv \varDelta_1 \ \texttt{@}\ (\kappa_1{\wedge}\pi_1) \vee \ldots \vee \varDelta_n \ \texttt{@}\ (\kappa_n{\wedge}\pi_n) \\
\mathtt{U}^{\mathtt{pre}}(\bar{x}){\wedge}\pi_a \Rightarrow (\mathtt{U}^{\mathtt{pre}}_{\mathtt{d}}(\bar{x}) * \kappa{\wedge}\pi) \ \texttt{@}\ (\kappa_g{\wedge}\pi_g) \\
\mathcal{S}_1 = \{\varDelta_i{*}\varDelta_r{*}\kappa_i \mid \kappa{\wedge}\pi \vdash \kappa_i{\wedge}\pi_i \leadsto (\emptyset,\ \varDelta_r)\} \\
\mathcal{S}_2 = \{\kappa{*}\varDelta_i{\wedge}\pi \ \texttt{@}\ (\kappa_i{\wedge}\pi_i) \mid \mathtt{SAT}(\kappa{\wedge}\kappa_i{\wedge}\pi{\wedge}\pi_i), \kappa{\wedge}\pi{\not\vdash}\kappa_i{\wedge}\pi_i\}
\end{array}}{\mathtt{U}^{\mathtt{pre}}(\bar{x}){\wedge}\pi_a{\Rightarrow} \bigvee_{\varDelta\in\mathcal{S}_1\cup\mathcal{S}_2} (\varDelta \ \texttt{@}\ (\kappa_g{\wedge}\pi_g))}$$

Formally, such predicate inlining can be carried out by the $[\underline{\textbf{syn-inline}}]$ rule. Inlining serves two purposes: (i) allow instantiation of back pointers with the use of heap guards; (ii) minimize the number of predicates a definition relies on. As an eager optimization, the inlining discards infeasible disjuncts in which the context contradicts the guard and also drops the guard where it is already satisfied.

B.5 Conjunctive Unification

When describing the pre-predicate derivation, we observed that there is a need for a normalization operation for formulas $\Delta_1 \wedge \Delta_2$ to ensure the result is within the logic fragment described in Sec. 2. We obtain this normalization through a *conjunctive unification* step, $\Rightarrow_{\wedge}^{\bar{\mathtt{v}}}$. Informally, in order for $\Delta_1 \wedge \Delta_2$ to be satisfiable, to describe at least one feasible heap, it is necessary that the shapes described by Δ_1 and Δ_2 agree when describing the same memory locations. Based on this observation, it is possible to construct a possibly stronger approximation for $\Delta_1 \wedge \Delta_2$ expressed in our logic fragment by unifying the common heap locations as follows:

$$\pi \wedge \Delta \Rightarrow_{\wedge}^{\bar{v}} (\Delta \wedge \pi, []) \qquad \Delta \wedge \pi \Rightarrow_{\wedge}^{\bar{v}} (\Delta \wedge \pi, [])$$

$$\frac{\Delta_1 \wedge \Delta_3 \Rightarrow_{\wedge}^{\bar{v}} (\Delta_5, \mathcal{S}_1) \qquad \Delta_2 \wedge \Delta_4 \Rightarrow_{\wedge}^{\bar{v}} (\Delta_6, \mathcal{S}_2)}{(\Delta_1 \vee \Delta_2) \wedge (\Delta_3 \vee \Delta_4) \Rightarrow_{\wedge}^{\bar{v}} (\Delta_5 \vee \Delta_6, \mathcal{S}_1 \cup \mathcal{S}_2)}$$

$$\frac{\begin{array}{c}\texttt{P is known} \qquad x{\in}\bar{v} \qquad \bar{y} \cap \bar{v} = \emptyset \qquad \rho{=}[\bar{y} \mapsto \bar{z}] \\ \Delta_1 \wedge \rho\, \Delta_2 \Rightarrow_{\wedge}^{\bar{v}\cup\bar{z}} (\Delta_3, \mathcal{S})\end{array}}{P(x,\bar{z}) * \Delta_1 \wedge P(x,\bar{y}) * \Delta_2 \Rightarrow_{\wedge}^{\bar{v}} (P(x,\bar{z}) * \Delta_3, \mathcal{S})}$$

$$\frac{\begin{array}{c}\mathtt{U}_1, \mathtt{U}_2\ \texttt{are dangling} \quad x \in \bar{v} \quad \rho = [\bar{y} \mapsto \bar{z}] \quad \bar{y} \cap \bar{v} = \emptyset \\ \Delta_1 \wedge \rho\, \Delta_2 \Rightarrow_{\wedge}^{\bar{v}\cup\bar{z}} (\Delta_3, \mathcal{S}) \qquad \mathcal{S}_1 = \mathcal{S} {\cup} \{\mathtt{U}_1{\equiv}\mathtt{U}_2\}\end{array}}{\mathtt{U}_1(x,\bar{z}){*}\Delta_1 \wedge \mathtt{U}_2(x,\bar{y}){*}\Delta_2 \Rightarrow_{\wedge}^{\bar{v}} (\mathtt{U}_1(x,\bar{z}){*}\Delta_3, \mathcal{S}_1)}$$

$$\frac{x{\in}\bar{v} \quad \rho = [\bar{y} \mapsto \bar{z}] \quad \bar{y}{\cap}\bar{v}{=}\emptyset \quad \Delta_1 \wedge \rho\, \Delta_2 \Rightarrow_{\wedge}^{\bar{v}\cup\bar{z}} (\Delta_3, \mathcal{S})}{x{\mapsto}c(\bar{z}) * \Delta_1 \wedge x{\mapsto}c(\bar{y}) * \Delta_2 \Rightarrow_{\wedge}^{\bar{v}} (x{\mapsto}c(\bar{z}) * \Delta_3, \mathcal{S})}$$

To streamline the unification process, the $\Rightarrow_{\wedge}$ operation is parameterized with a set of variables $\bar{\mathtt{v}}$ which denotes the set of possibly common memory locations. We seed this parameter initially with the set of arguments of the predicate under construction.

In the process of unifying predicate instances or heap nodes, the transformation on one hand modifies the formulas by applying a substitution of the arguments, and on the other constructs extra constraints on dangling predicates, thus strengthening the result. We observe however that this is not affecting the soundness of the result as this strengthening is restricted to pre-condition predicate definitions. By imposing an equality constraint on dangling predicates the resulting definitions become more concise, with fewer extra predicates being synthesized.

Naturally, this strengthening *may* lead to a contradiction which we will consider as a failure of the shape analysis, due to contradictory scenarios. For example:

$$\mathtt{x{=}NULL} \wedge \mathtt{x{\mapsto}node(p,n)} \Rightarrow_{\wedge}^{\mathtt{x}} (\mathtt{FALSE}, [])$$

B.6 Disjunctive Unification

We propose to apply *disjunctive unification* to derive more concise definitions for post-predicate. The aim here is to factor out common constraints on disjunctive branches of a given post-predicate, so that common heap terms in disjunct can be abstracted:

$$\mathtt{x{\mapsto}c(\bar{y})} * \mathtt{\Delta_1} \vee \mathtt{x{\mapsto}c(\bar{y})} * \mathtt{\Delta_2} \Rightarrow \mathtt{U(x,\bar{a})} \Rightarrow_{\vee} \begin{cases} \mathtt{x{\mapsto}c(\bar{y}){*}R(x,\bar{a},\bar{y})} \Rightarrow \mathtt{U(x,\bar{a})} \\ \mathtt{\Delta_1} \vee \mathtt{\Delta_2} \Rightarrow \mathtt{R(x,\bar{a},\bar{y})} \end{cases}$$

B.7 Normalization of Shape Predicates

After we have derived suitable predicate definitions, we proceed with normalization to convert each predicate definition to its most concise form. Our current method uses four key steps:

```
function PRED_NORM($\Gamma$)
  $\Gamma_1 \leftarrow$ process-dangling-and-unused-preds $\Gamma$
  $\Gamma_2 \leftarrow$ eliminate-useless-parameters $\Gamma_1$
  $\Gamma_3 \leftarrow$ perform-predicate splitting on $\Gamma_2$
  return reuse-predicates $\Gamma_3$
end function
```

Detecting and Eliminating Dangling Predicates We have seen how relational assumptions are soundly transformed into predicate definitions. However, it is still possible for some pre-predicates not to have any definition. As mentioned in Sec. 3, these *dangling* predicates denote fields that were not accessed. Though it is safe to drop such predicates (by frame rule), we keep them to capture linking information between pre- and post-conditions.

In this predicate normalization step, we associate each dangling predicate $\mathtt{U}(\mathtt{x}, ..)$ encapsulating a pointer that is instantiated and not an argument of the current method, with a logical variable $\mathcal{D}_{\mathtt{U}}$ denoting such a predicate instance. With this extra notation, in effect, we are making explicit that the addresses pointed to by such fields have neither been read nor written to during the execution of its method. Thus, these logical variables appear in the precondition, and after the execution, in the postcondition. Notice that the marking of pointers as dangling is guided by the context and thus reflects the access patterns. We can formalize these steps as follows:

$$\frac{\mathtt{U}(\bar{y}) \equiv (\mathtt{U}_d(x,\bar{v}){*}\kappa{\wedge}\pi) \,@\, (\kappa_g{\wedge}\pi_g) \;\vee\; \Phi^g \quad \mathtt{x} \notin \bar{\mathtt{y}} \quad \mathtt{U}_{\mathtt{d}}(\mathtt{x},\bar{\mathtt{v}}) \equiv \top}{\mathtt{U}(\bar{y}) \equiv ([x{\mapsto}\mathcal{D}_{\mathtt{U}_d}](\kappa{\wedge}\pi)) \,@\, (\kappa_g{\wedge}\pi_g) \vee \Phi^g}$$

Eliminating Useless Parameters We observe that there are cases in which predicate arguments are not used in the synthesized definitions. For our `sll2dll` example (Sec. 3), the second parameter `q` in the derived pre-predicate is redundant. For the `tll` example outlined in Sec.1, the last two parameters of the derived pre-predicate are redundant.

In order to simplify the definitions and improve predicate reuse, we propose to detect and eliminate such arguments. For a given predicate definition, $\mathtt{P}(\bar{\mathtt{x}}) \equiv \Delta$, we can discover if any constraint in the predicate body involves a parameter through a standard, sound flow analysis. Once a set of candidate arguments, $\bar{\mathtt{z}}$, has been identified, we construct a new unknown predicate $\mathtt{U}_{\mathtt{z}}(\bar{\mathtt{x}}')$ where $\bar{\mathtt{x}}' = \bar{\mathtt{x}} \setminus \bar{\mathtt{z}}$ which can then be instantiated by running the bi-abductive entailment check on the entailment $\mathtt{P}'(\bar{\mathtt{x}}) \vdash \mathtt{U}_{\mathtt{z}}(\bar{\mathtt{x}}')$, where $\mathtt{P}'(\bar{\mathtt{x}}) \equiv \exists \bar{\mathtt{z}}.\Delta$. This would gather the necessary set of assumptions on $\mathtt{U}_{\mathtt{z}}$ that can provide a definition for the new predicate without the useless argument(s). Followed by a check that with the inferred definition $\mathtt{U}_{\mathtt{z}}(\bar{\mathtt{x}})' \vdash \mathtt{P}'(\bar{\mathtt{x}})$. Thus, ideally, the resulting definitions would follow: $\mathtt{P}(\bar{\mathtt{x}}) \equiv \mathtt{U}_{\mathtt{z}}(\bar{\mathtt{x}}'), \mathtt{U}_{\mathtt{z}}(\bar{\mathtt{x}}') \equiv \Delta'$ Using this step, we can obtain a simpler pre-predicate for `tll` (Sec 1):

$\mathtt{H}(\mathtt{x},\mathtt{p},\mathtt{t}) \equiv \mathtt{H}_{\mathtt{f}}(\mathtt{x})$
$\mathtt{H}_{\mathtt{f}}(\mathtt{x}) \equiv \mathtt{x}{\mapsto}\mathtt{tree}(\mathcal{D}_{\mathtt{p}},\mathcal{D}_{\mathtt{l}},\mathtt{r},\mathcal{D}_{\mathtt{n}}) \wedge \mathtt{r}{=}\mathtt{NULL} \vee \mathtt{x}{\mapsto}\mathtt{tree}(\mathcal{D}_{\mathtt{p}},\mathtt{l},\mathtt{r},\mathcal{D}_{\mathtt{n}}){*}\mathtt{H}_{\mathtt{f}}(\mathtt{l}){*}\mathtt{H}_{\mathtt{f}}(\mathtt{r}) \wedge \mathtt{r} \neq \mathtt{NULL}$

Reusing Predicates In order to derive more concise predicate definitions, we propose an equivalence detection step that would try and match a newly inferred definition with a definition previously provided or inferred. We leverage on our second-order entailment prover to perform this task, but limit its folding steps to identity matching of predicates that may possibly be equivalent. We also use an analysis to pre-determine those predicates that are unlikely to be equivalent, or have already been processed as such. For any two synthesized predicates $\mathtt{U1}(\bar{\mathtt{v}})$ and $\mathtt{U2}(\bar{\mathtt{w}})$, we first align their parameters, and then prove two entailments *unroll*$[\mathtt{U1}(\bar{\mathtt{v}})] \vdash \mathtt{U2}(\bar{\mathtt{w}})$ and *unroll*$[\mathtt{U2}(\bar{\mathtt{w}})] \vdash \mathtt{U1}(\bar{\mathtt{v}})$. (Each *unroll* replaces a predicate instance by its definition. It ensures that our inductive proof is well-founded.) If both entailments fail, we assert the pair of predicates to be disequal. If only one of the entailments succeeds, we assert a *predicate subsumption* has been detected. If both succeed and return a further set of possibly equivalence pairs, we proceed to prove the equivalence of pairs from the new set. When no more pairs of possibly equivalent predicates are found, we assert $\mathtt{U1}(\bar{\mathtt{v}}) \leftrightarrow \mathtt{U2}(\bar{\mathtt{v}})$ to indicate the equivalence of the sets of pairs of predicates that we have just proven.

Predicate Splitting Here we illustrate a *split-predicate* tactic to derive, where possible, lemmas of the following form: $\mathtt{U}(\mathtt{x},\mathtt{y}) \rightarrow \mathtt{U}_1(\mathtt{x}){*}\mathtt{U}_2(\mathtt{y})$ which denotes known facts about valid implication over heap formula that can be used by the entailment checker. For example, consider a predicate that captures $\mathtt{x}$ and $\mathtt{y}$, two linked lists of same length:

$$\mathtt{twosame}(\mathtt{x},\mathtt{y}) \equiv \mathtt{x}{=}\mathtt{NULL}{\wedge}\mathtt{y}{=}\mathtt{NULL}{\vee}\mathtt{x}{\mapsto}\mathtt{snode}(\mathtt{x_n}){*}\mathtt{y}{\mapsto}\mathtt{snode}(\mathtt{y_n}){*}\mathtt{twosame}(\mathtt{x_n},\mathtt{y_n})$$

To explore such splitting, we can follow the example of useless parameter elimination where our second-order bi-abductive entailment is again used to infer definitions for unknown predicates, $\mathtt{U}_1$ and $\mathtt{U}_2$. Once these definitions are derived, we can even use the same entailment check to determine if the converse implication $\mathtt{twosame}(\mathtt{x},\mathtt{y}) \leftarrow \mathtt{U}_1(\mathtt{x}){*}\mathtt{U}_2(\mathtt{y})$ holds. For this example, we can only derive [6]:

$$\begin{array}{lll}
\mathtt{twosame}(\mathtt{x},\mathtt{y}) & \rightarrow & \mathtt{U}_1(\mathtt{x}){*}\mathtt{U}_2(\mathtt{y}) \\
\mathtt{U}_1(\mathtt{x}) & \equiv & \mathtt{x}{=}\mathtt{NULL} \vee \mathtt{x}{\mapsto}\mathtt{snode}(\mathtt{n}) * \mathtt{U}_1(\mathtt{n}) \\
\mathtt{U}_2(\mathtt{x}) & \equiv & \mathtt{x}{=}\mathtt{NULL} \vee \mathtt{x}{\mapsto}\mathtt{snode}(\mathtt{n}) * \mathtt{U}_2(\mathtt{n})
\end{array}$$

Now, two resulting predicates $\mathtt{U}_1$ and $\mathtt{U}_2$ have more chances to be matched/reused with given library predicates (e.g. `ll` in Sec. 4).

C S2 Virtual Machine (S2-VM)

There are two options to reproduce the results in Sec. 7:

1. Testing programs through the tool's website (by clicking the URL given in Sec 1).
2. Setting up and conducting the benchmark on a virtual machine S2-VM as suggested in what follows.

[6] We only derive a weakening lemma that can be applied to post-condition, but not to pre-condition. For safely splitting pre-condition of `twosame` which captures two lists of the same length, we will need to extend our inference to capture size properties on lists.

C.1 Set up The Virtual Machine

- Download it by clicking the URL below. Note to copy-and-paste the link may not work correctly (in which case, please ensure the character ~ before `project` is shown properly in the browser).
  - `http://loris-7.ddns.comp.nus.edu.sg/~project/s2/beta/s2.ova`
  - checksum of **s2.ova** file: `5874796047da26ce53569930d3ae183e`
  - virtual machine player: Oracle VM VirtualBox 4.3.6 (or VMware Player 6.0.1)
- Open Terminal in Desktop of S2-VM. Set the working directory to `Desktop/s2`:

```
> cd Desktop/s2
```

- Build S2 from the source code:

```
> make hip
```

- Copy s2 binary into cav14/bin folder:

```
> cp hip cav14/bin
```

C.2 Reproduce the Experimental Results in S2-VM

Assume the current directory of the terminal is **Desktop/s2**

- Set the directory to `cav14` (which contains all examples in the Experiments):

```
> cd cav14
```

- Please refer to README file for the structure of the **cav14** folder.
- Synthesize small benchmark in Table 1 (Sec 7) without normalization:

```
> cd small
> ./run-wo.sh
> cd ..
```

- Synthesize small benchmark in Table 1 (Sec 7) with normalization:

```
> cd small
> ./run-norm.sh
> cd ..
```

- Verify small benchmark in Table 1(Sec 7):

```
> cd small/veri
> ./veri.sh
> cd ../..
```

- Analyze Glib Programs:

```
> cd glib
> ./run-glib.sh
> cd ..
```

D Two More Examples

We provide further illustrations of our proposal through two examples to highlight key features of our shape inference mechanism.

```
void append(struct node * x, struct node * y)
 requires H(x, y#) ensures G(x, y#)
  {if (x->next) append(x->next, y);
   else { x->next = y; y->prev = x; } }
```

The `append` method joins two doubly-linked lists. To guide the shape synthesis, the initial stub specification is pre-analysed with # annotations. Thus, by the same process described in previous sections the following relational assumptions are inferred:

1 $\mathtt{H(x,y\#)} \Rightarrow \mathtt{x{\mapsto}node(x_p,x_n){*}H_p(x_p,y\#){*}H_n(x_n,y\#){*}H_y(y,x\#)}$
2 $\mathtt{H_n(x_n,y\#){*}H_y(y,x\#){\wedge}x_n{\neq}NULL} \Rightarrow \mathtt{H(x_n,y\#)}$
3 $\mathtt{H_y(y,x\#)} \Rightarrow \mathtt{y{\mapsto}node(y_p,y_n){*}H_{yp}(y_p,x\#){*}H_{yn}(y_n,x\#)}$
4 $\mathtt{H_n(x_n,y\#){\wedge}x_n{=}NULL} \Rightarrow \mathtt{emp}$
5 $\mathtt{H_p(x_p,y\#){*}x{\mapsto}node(x_p,x_n){*}G(x_n,y\#){\wedge}x_n{\neq}NULL} \Rightarrow \mathtt{G(x,y\#)}$
6 $\mathtt{x{\mapsto}node(x_p,y){*}y{\mapsto}node(x,y_n){*}H_{yn}(y_n,x\#){*}H_p(x_p,y\#)}{\Rightarrow}\mathtt{G(x,y\#)}$

We can then synthesize the following predicate definitions:

$$\begin{aligned}
&\mathtt{H(x,y)}{\equiv}\mathtt{x{\mapsto}node}(\mathcal{D}_p,\mathtt{x_n}){*}\mathtt{H_n(x_n)}{*}\mathtt{y{\mapsto}node}(\mathcal{D}_{yp},\mathcal{D}_{yn})\\
&\mathtt{H_n(x_n)}{\equiv}\mathtt{emp}{\wedge}\mathtt{x_n{=}NULL} \vee\ \mathtt{x_n{\mapsto}node}(\mathcal{D}_p,\mathtt{x_{nn}}){*}\mathtt{H_n(x_{nn})}\\
&\mathtt{G(x,y)}{\equiv}\mathtt{x{\mapsto}node}(\mathcal{D}_p,\mathtt{y}){*}\mathtt{y{\mapsto}node}(\mathtt{x},\mathcal{D}_{yn})\\
&\qquad\quad \vee\ \mathtt{x{\mapsto}node}(\mathcal{D}_p,\mathtt{x_n}){*}\mathtt{G(x_n,y)}{\wedge}\mathtt{x_n{\neq}NULL}
\end{aligned}$$

Our shape inference mechanism manages to infer a precise (weak) pre-condition which only requires a singly-linked list for the first parameter, and a single node for the second parameter without enforcing unnecessary constraints on the rest of the locations reachable from the second parameter. Furthermore, the derived post-predicate describes a non-empty recursive list segment joined with the structure described by the second parameter. Through the use of dangling references, the derived specification permits cyclic data structures for the second parameter, and moreover guarantees that only its first node is being changed. This more precise pre/post specification subsumes the specification which uses two doubly-linked lists for the two parameters.

To illustrate a more complex data structure, consider the following mutual-recursive methods to validate a rose tree, whose children are linked via a doubly-linked list with parent pointers,

```
struct mtree {int val; struct mnode* children; }
struct mnode { struct mtree* child; struct mnode* prev;
               struct mnode* next; struct mtree* parent; }
bool c_tree (struct mtree* t)
 requires H1(t) ensures G1(t) ∧ res;
{  if (t->children == null) return true;
   else return c_child(t->children, NULL, t); }

bool c_child(struct mnode* l, struct mnode* prv,
                 struct mtree* par)
requires H2(l, prv, par) ensures G2(l, prv, par) ∧ res;
{ if (l == null) return true;
  else if (l->parent == par && l->prev == prv)
         return c_child(l->next, l, par) && c_tree(l->child);
       else return false; }
```

This checker code is special in that we are using it to validate some expected data structure. We use it here primarily for evaluating the precision of our synthesis method. Notice the use of mixed constraints[7] e.g. $\mathtt{G_1(t)}\wedge\mathtt{res}$, which requires both inference and verification to work together. This comes naturally from our integration of *second-order bi-abduction* into an existing separation logic verifier with proving capability. Essentially, this code is checking that each tree node contains a pointer to a null-terminated doubly-linked list, with pointers to parent node. Our approach is able to derive the following precise and concise predicate definitions. We achieve this feat by using normalization techniques which unify disjuncts and semantically-equivalent predicates, where possible.

$\mathtt{H_1(t)}{\equiv}\mathtt{t}{\mapsto}\mathtt{mtree(v,c)}{*}\mathtt{H_2(c,NULL,t)}$

$\mathtt{H_2(l,b,p)} \equiv \mathtt{emp}{\wedge}\mathtt{l}{=}\mathtt{NULL}{\vee}\mathtt{H_2(n_1,l,p)}{*}\mathtt{l}{\mapsto}\mathtt{mnode(c_1,b,n_1,p)}{*}\mathtt{c_1}{\mapsto}\mathtt{mtree(v,c)}{*}\mathtt{H_2(c,NULL,c_1)}$

$\mathtt{G_1(t)}{\equiv}\mathtt{H_1(t)}$

$\mathtt{G_2(l,b,p)}{\equiv}\mathtt{H_2(l,b,p)}$

E Expanded Soundness

Proof for Lemma 1 We will show that for all $\Delta_{\mathtt{ante}}$ and $\Delta_{\mathtt{conseq}}$ such that

$$\Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} \leadsto (\mathcal{R},\ \Delta_{\mathtt{frame}})$$

and $\Gamma{=}\{\mathtt{U_1}(\bar{\mathtt{v}}_1){\equiv}\Delta_1, ..\mathtt{U_n}(\bar{\mathtt{v}}_n){\equiv}\Delta_n\}$, a set of instantiations for unknown predicates such that $\mathcal{R}(\Gamma)$ then the entailment $\Gamma{:}\Delta_{\mathtt{ante}} \ \vdash\ \Delta_{\mathtt{conseq}}{*}\Delta_{\mathtt{frame}}$ holds. We will show by structural induction on $\Delta_{\mathtt{conseq}}$.

[7] C languages uses integer for boolean values, which gets translated to boolean type in our core language.

Due to the construction of the $\Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} \leadsto (\mathcal{R},\ \Delta_{\mathtt{frame}})$ procedure as an extension of an existing entailment procedure with frame inference, for all $\Delta_{\mathtt{ante}}$ and $\Delta_{\mathtt{conseq}}$ not involving unknown predicates $\mathcal{R} = \mathtt{true}$ and $\Gamma{:}\Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}}{*}\Delta_{\mathtt{frame}}$.

Bellow we consider the cases that actually involve unknown predicates. These cases fall under two categories:

– $\Delta_{\mathtt{ante}} = U(\mathtt{r}, \bar{\mathtt{v}}_{\mathtt{i}}, \bar{\mathtt{v}}_{\mathtt{n}}\#){*}\kappa_1{\wedge}\pi_1$ and $\Delta_{\mathtt{conseq}} = \kappa_s{*}\kappa_2{\wedge}\pi_2$ where $\kappa_{\mathtt{s}} \equiv \mathtt{r}{\mapsto}\mathtt{c}(\bar{\mathtt{d}}, \bar{\mathtt{p}})$ or $\kappa_{\mathtt{s}} \equiv \mathtt{P}(\mathtt{r}, \bar{\mathtt{d}}, \bar{\mathtt{p}})$. By hypothesis $\Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} \leadsto (\mathcal{R},\ \Delta_{\mathtt{frame}})$. Then, as described in Sec.6 the [SO-ENT-UNFOLD] step must hold ensuring the following assertion holds:

$$\kappa_1{*}\Delta_{\mathtt{dangl}}{*}\Delta_{\mathtt{rem}}{\wedge}\pi_1 \vdash \kappa_2{\wedge}\pi_2 \leadsto (\mathcal{R}',\ \Delta_{\mathtt{frame}})$$

where $\mathcal{R} = \mathcal{R}' \wedge (\mathtt{U}(\mathtt{r}, \bar{\mathtt{v}}_{\mathtt{i}}, \bar{\mathtt{v}}_{\mathtt{n}}\#) \wedge \pi_{\mathtt{a}} \Rightarrow \kappa_{\mathtt{s}} * \Delta_{\mathtt{dangl}} * \Delta_{\mathtt{rem}} \wedge \pi_{\mathtt{c}})$ It follows from the structural induction hypothesis that:

$$\Gamma{:}\kappa_1{*}\Delta_{\mathtt{dangl}}{*}\Delta_{\mathtt{rem}}{\wedge}\pi_1 \vdash (\kappa_2{\wedge}\pi_2) * \Delta_{\mathtt{frame}} \tag{1}$$

From $\mathcal{R}(\Gamma)$ it follows that

$$\Gamma{:}\mathtt{U}(\mathtt{r}{,}\bar{\mathtt{v}}_{\mathtt{i}}{,}\bar{\mathtt{v}}_{\mathtt{n}}\#){\wedge}\pi_{\mathtt{a}}{\vdash}\kappa_{\mathtt{s}}{*}\Delta_{\mathtt{dangl}}{*}\Delta_{\mathtt{rem}}{\wedge}\pi_{\mathtt{c}} \tag{2}$$

From equations 1 and 2 it follows that $\Gamma{:}\Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} * \Delta_{\mathtt{frame}}$.

– $\Delta_{\mathtt{ante}}{=}\kappa_1{\wedge}\pi_1$ and $\Delta_{\mathtt{conseq}}{=}\mathtt{U}_{\mathtt{c}}(\bar{\mathtt{w}}{,}\bar{\mathtt{z}}\#){*}\kappa_2{\wedge}\pi_2$.
Let $\pi_r = \mathtt{slice}(\bar{r}, \pi_1)$ and $\pi_w = \mathtt{slice}(\bar{w}, \pi_1)$.
By hypothesis :$\Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} \leadsto (\mathcal{R},\ \Delta_{\mathtt{frame}})$.
Then, as described in Sec.6 the [SO-ENT-FOLD] step must hold ensuring the following assertions hold:
  - $\kappa_1 = \kappa_{11}{*}\kappa_{12}$
  - $\kappa_{12} \wedge \pi_1 \vdash \kappa_2 \wedge \pi_2 \leadsto (\mathcal{R}',\ \Delta_{\mathtt{frame}})$ which by structural induction leads to $\Gamma{:}\kappa_{12} \wedge \pi_1 \vdash (\kappa_2 \wedge \pi_2) * \Delta_{\mathtt{frame}}$
  - $\mathcal{R}{=}(\kappa_{11}{\wedge}\pi_{\mathtt{w}}{\Rightarrow}\mathtt{U}_{\mathtt{c}}(\bar{\mathtt{w}}, \bar{\mathtt{z}}\#) \ @\ \kappa_{\mathtt{g}}{\wedge}\pi_{\mathtt{r}})\ {\wedge}\mathcal{R}'$ which by $\mathcal{R}(\Gamma)$ leads to: $\Gamma{:}\kappa_{11}{\wedge}\pi_{\mathtt{w}} \vdash \mathtt{U}_{\mathtt{c}}(\bar{\mathtt{w}}, \bar{\mathtt{z}}\#) \ @\ \kappa_{\mathtt{g}}{\wedge}\pi_{\mathtt{r}}$ Note that by the definition in Sec.2 for guarded assumptions, it follows that $\Delta \ @\ (\kappa_g{\wedge}\pi_r){*}\kappa_g{\wedge}\pi_r$ is equivalent with $\Delta{*}(\kappa_g{\wedge}\pi_r)$.

From the above three assertions it follows that $\Gamma{:}\Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} * \Delta_{\mathtt{frame}}$

$\square$

Proof for Lemma 2 We will show that given a set of relational assumptions $\mathcal{R}$ and one of the synthesis rules is applied to obtain $\mathcal{R}'$ then if exists Γ such that $\mathcal{R}'(\Gamma)$ then $\mathcal{R}(\Gamma)$.

– If [syn-base] was applied to $\mathcal{R} \wedge (\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){*}\kappa{\wedge}\pi{\Rightarrow}\mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}}))$ then the resulting assumptions are either:
  1. $\mathcal{R}'{=}\mathcal{R} \wedge (\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){\wedge}\pi'{\Rightarrow}\mathtt{emp}) \wedge (\kappa{\wedge}\pi \Rightarrow \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}}))$
     OR
  2. $\mathcal{R}'{=}\mathcal{R} \wedge (\mathtt{U}^{\mathtt{fr}}(\bar{\mathtt{x}}) \Rightarrow \top)\ \wedge\ (\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){\Rightarrow}\mathtt{U}^{\mathtt{fr}}(\bar{\mathtt{x}}) \ @\ (\kappa_{\mathtt{g}}{\wedge}\pi_{\mathtt{g}})){\wedge}$ $(\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){*}\kappa{\wedge}\pi \Rightarrow \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}}))$ where $\mathtt{U}^{\mathtt{fr}}$ is a fresh unknown predicate

If there exists Γ such that $\mathcal{R}'(\Gamma)$ then by definition, using the Γ interpretation for the unknown predicates then either:

1. $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}})\wedge\pi' \vdash \mathtt{emp}$ and $\Gamma{:}\kappa\wedge\pi \vdash \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}})$ and since by construction $\pi\vdash\pi'$ it follows that $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){*}\kappa\wedge\pi \vdash \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}})$ thus $\mathcal{R}(\Gamma)$.
2. $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){*}\kappa\wedge\pi \vdash \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}})$ and $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}) \vdash \mathtt{U}^{\mathtt{fr}}(\bar{\mathtt{x}}) \mathbin{@} (\kappa_{\mathtt{g}}\wedge\pi_{\mathtt{g}})$ and $\Gamma{:}\mathtt{U}^{\mathtt{fr}}(\bar{\mathtt{x}}) \vdash \top$ leading to $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){*}\kappa\wedge\pi \vdash \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}}) * \top \mathbin{@} (\kappa_g\wedge\pi_g)$ which by construction of $\kappa_g\wedge\pi_g$ and σ_2 leads to $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){*}\kappa\wedge\pi \vdash \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}})$ and thus $\mathcal{R}(\Gamma)$.

– If $\boxed{\textbf{syn-case}}$ was applied to: $\mathcal{R} \wedge (\mathtt{U}(\bar{\mathtt{x}})\wedge\pi_1{\Rightarrow}\Delta_1 \mathbin{@} \Delta_{1\mathtt{g}})\wedge(\mathtt{U}(\bar{\mathtt{x}})\wedge\pi_2{\Rightarrow}\Delta_2 \mathbin{@} \Delta_{2\mathtt{g}})$. To generate:

$$\mathcal{R}' = \mathcal{R} \wedge (\mathtt{U}(\bar{x})\wedge\pi_1\wedge\neg\pi_2{\Rightarrow}\Delta_1 \mathbin{@} \Delta_{3g})\wedge \\ (\mathtt{U}(\bar{x})\wedge\pi_2\wedge\neg\pi_1{\Rightarrow}\Delta_2 \mathbin{@} \Delta_{3g}) \wedge\ (\mathtt{U}(\bar{x})\wedge\pi_1\wedge\pi_2{\Rightarrow}\Delta_3 \mathbin{@} \Delta_{3g})$$

with $\pi_1\wedge\pi_2{\not\Rightarrow}\mathtt{FALSE}$ and $\Delta_1\wedge\Delta_2{\Rightarrow}^{\bar{\mathtt{x}}}_{\wedge}\Delta_3$ and $\Delta_{1g}\wedge\Delta_{2g}{\Rightarrow}^{\bar{x}}_{\wedge}\Delta_{3g}$ and exists a Γ such that $\mathcal{R}'(\Gamma)$. From $\mathcal{R}'(\Gamma)$ it follows that:

$$\Gamma{:}\mathtt{U}(\bar{x})\wedge\pi_1\wedge\neg\pi_2 \vdash \Delta_1 \mathbin{@} \Delta_{3g} \quad \Gamma{:}\mathtt{U}(\bar{x})\wedge\pi_2\wedge\neg\pi_1 \vdash \Delta_2 \mathbin{@} \Delta_{3g} \\ \Gamma{:}\mathtt{U}(\bar{x})\wedge\pi_1\wedge\pi_2 \vdash \Delta_3 \mathbin{@} \Delta_{3g}$$

We need to show that: $\Gamma{:}\mathtt{U}(\bar{x})\wedge\pi_1 \vdash \Delta_1 \mathbin{@} \Delta_{1g}$ and $\Gamma{:}\mathtt{U}(\bar{x})\wedge\pi_2 \vdash \Delta_2 \mathbin{@} \Delta_{2g}$.
Note that by the definition of the conjunctive unification, it follows that if $\Delta_1\wedge\Delta_2{\Rightarrow}^{\bar{x}}_{\wedge}\Delta_3$ then $\Gamma{:}\Delta_3 \vdash \Delta_1$ and $\Gamma{:}\Delta_3 \vdash \Delta_2$. Thus from $\Gamma{:}\mathtt{U}(\bar{\mathtt{x}})\wedge\pi_1\wedge\pi_2 \vdash \Delta_3 \mathbin{@} \Delta_{3\mathtt{g}}$ and follows that: $\Gamma{:}\mathtt{U}(\bar{x})\wedge\pi_1\wedge\pi_2 \vdash \Delta_1 \mathbin{@} \Delta_{3g}$ and $\Gamma{:}\mathtt{U}(\bar{x})\wedge\pi_1\wedge\pi_2 \vdash \Delta_2 \mathbin{@} \Delta_{3g}$
Thus it follows: $\Gamma{:}\mathtt{U}(\bar{x})\wedge\pi_1\wedge(\neg\pi_2 \vee \pi_2) \vdash \Delta_1 \mathbin{@} \Delta_{3g}$
which simplifies to $\Gamma{:}\mathtt{U}(\bar{x})\wedge\pi_1 \vdash \Delta_1 \mathbin{@} \Delta_{3g}$ which by the construction of Δ_{3g} leads to $\Gamma{:}\mathtt{U}(\bar{x})\wedge\pi_1 \vdash \Delta_1 \mathbin{@} \Delta_{1g}$. Similarly we obtain: $\Gamma{:}\mathtt{U}(\bar{x})\wedge\pi_2 \vdash \Delta_2 \mathbin{@} \Delta_{2g}$.

– $\boxed{\textbf{syn-group-pre}}$ The proof obligation reduces to: if there exists Γ such that $\Gamma{:}\mathtt{U}(\bar{v}) \wedge (\pi_1{\vee}\pi_2) \vdash \Phi^g{}_1\wedge\pi_1 \vee \Phi^g{}_2\wedge\pi_2$ and $\pi_1\wedge\pi_2 \vdash \mathtt{FALSE}$ then $\Gamma{:}\mathtt{U}(\bar{v})\wedge\pi_1 \vdash \Phi^g{}_1$ and $\Gamma{:}\mathtt{U}(\bar{v})\wedge\pi_2 \vdash \Phi^g{}_2$. It follows that $\Gamma{:}\mathtt{U}(\bar{v})\wedge\pi_1 \vdash \Phi^g{}_1\wedge\pi_1{\vee}\Phi^g{}_2\wedge\pi_2$ and $\Gamma{:}\mathtt{U}(\bar{v})\wedge\pi_2 \vdash \Phi^g{}_1\wedge\pi_1{\vee}\Phi^g{}_2\wedge\pi_2$. And since $\pi_1\wedge\pi_2 \vdash \mathtt{FALSE}$ it follows that $\Gamma{:}\mathtt{U}(\bar{v})\wedge\pi_1 \vdash \Phi^g{}_1\wedge\pi_1$ and $\Gamma{:}\mathtt{U}(\bar{v})\wedge\pi_2 \vdash \Phi^g{}_2\wedge\pi_2$.

– $\boxed{\textbf{syn-group-post}}$ It follows trivially that $\Gamma{:}\Delta_a \vdash \mathtt{U}^{\mathtt{post}}(\bar{v})$ and $\Gamma{:}\Delta_b \vdash \mathtt{U}^{\mathtt{post}}(\bar{v})$ from $\Gamma{:}\Delta_a \vee \Delta_b \vdash \mathtt{U}^{\mathtt{post}}(\bar{v})$.

– if $\boxed{\textbf{syn-inline}}$ was applied to : $\mathcal{R}\wedge(\mathtt{U}^{\mathtt{pre}}(\bar{x})\wedge\pi_a \Rightarrow (\mathtt{U}^{\mathtt{pre}}_{\mathtt{d}}(\bar{x}) * \kappa\wedge\pi) \mathbin{@} (\kappa_g\wedge\pi_g))$ resulting in the assumption set: $\mathcal{R}' = \mathcal{R}\wedge(\mathtt{U}^{\mathtt{pre}}(\bar{x})\wedge\pi_a{\Rightarrow} \bigvee_{\Delta_i\in\mathcal{S}_1\cup\mathcal{S}_2} (\Delta_i \mathbin{@} (\kappa_g\wedge\pi_g)))$ when

$$\mathtt{U}^{\mathtt{pre}}_{\mathtt{d}}(\bar{x}) \equiv \Delta_1 \mathbin{@} (\kappa_1\wedge\pi_1) \vee \ldots \vee \Delta_n \mathbin{@} (\kappa_n\wedge\pi_n) \\ \mathcal{S}_1 = \{\Delta_i{*}\Delta_r{*}\kappa_i \mid \kappa\wedge\pi \vdash \kappa_i\wedge\pi_i \rightsquigarrow (\emptyset,\ \Delta_r)\} \\ \mathcal{S}_2 = \{\kappa{*}\Delta_i\wedge\pi \mathbin{@} (\kappa_i\wedge\pi_i) \mid \mathtt{SAT}(\kappa\wedge\kappa_i\wedge\pi\wedge\pi_i), \kappa\wedge\pi{\not\vdash}\kappa_i\wedge\pi_i\}$$

We need to prove that if exists Γ such that $\mathcal{R}'(\Gamma)$ then, $\mathcal{R}(\Gamma)$. That is: $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{x})\wedge\pi_a \vdash (\mathtt{U}^{\mathtt{pre}}_{\mathtt{d}}(\bar{x}) * \kappa\wedge\pi) \mathbin{@} (\kappa_g\wedge\pi_g)$. Which by using the $\mathtt{U}^{\mathtt{pre}}_{\mathtt{d}}$ definition translates in having to prove: $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}})\wedge\pi_{\mathtt{a}} \vdash \bigvee_{\mathtt{i}\in 1\ldots\mathtt{n}}((\Delta_{\mathtt{i}} \mathbin{@} (\kappa_{\mathtt{i}}\wedge\pi_{\mathtt{i}}) * \kappa\wedge\pi) \mathbin{@} (\kappa_{\mathtt{g}}\wedge\pi_{\mathtt{g}}))$
From $\mathcal{R}'(\Gamma)$ it follows that: $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{x})\wedge\pi_a{\vdash} \bigvee_{\Delta_i\in\mathcal{S}_1\cup\mathcal{S}_2} (((\kappa\wedge\pi){*}\Delta_i) \mathbin{@} (\kappa_g\wedge\pi_g))$.
We will show that:

$$\bigvee_{\Delta_i\in\mathcal{S}_1\cup\mathcal{S}_2} (((\kappa\wedge\pi){*}\Delta_i) \mathbin{@} (\kappa_g\wedge\pi_g)) \equiv \bigvee_{i\in 1\ldots n} (\Delta_i \mathbin{@} (\kappa_i\wedge\pi_i) * \kappa\wedge\pi) \mathbin{@} (\kappa_g\wedge\pi_g)$$

Observe that by the definition of the guard assertion, $(\Delta_i \ @ \ (\kappa_i \wedge \pi_i) * \kappa \wedge \pi)$ a RHS disjunction where the guard $\kappa_{\mathtt{i}} \wedge \pi_{\mathtt{i}}$ contradicts the context $\kappa \wedge \pi$ is equivalent to FALSE and thus can be discarded, leaving only disjuncts that do not contradict the context. Note that by construction, $\mathcal{S}_1 \cup \mathcal{S}_2$ denotes exactly that set. Furthermore, by the definition of the guarded assumption, assertions $(\Delta_i \ @ \ (\kappa_i \wedge \pi_i) * \kappa \wedge \pi)$ in which $\kappa \wedge \pi \vdash \kappa_{\mathtt{i}} \wedge \pi_{\mathtt{i}}$ can be reduced to $\Delta_i * \kappa \wedge \pi$. Observe that the result of the application of the above two equivalence preserving simplification steps on the RHS is identical to the LHS. Thus the required disjunction equivalence holds.

Proof for Lemma 3 Follows from the observation that there are only two rules generating predicate definitions: [**syn-pre-def**] and [**syn-post-def**]. Each applicable only if there exists only one assumption corresponding to the predicate that is currently being derived. Each rule generates exactly the predicate definition that would satisfy the unique assumption.

Proof for Lemma 4 First, we observe that the algorithm in Fig.7 finishes only when all assumptions have been catered for: assumptions used for synthesis have been reduced to a unique assumption which becomes the predicate definition; assumptions not included in the synthesis are discharged by an entailment step. Thus by the previous lemmas and the soundness of the underlying entailment checker the resulting definitions satisfy all the initial assumptions.

Second, we will show that all normalization steps are meaning preserving:

– Dangling elimination: We need to show that if $\mathtt{U}_d(x, \bar{v}) \equiv \top$ and $\mathtt{x} \notin \bar{\mathtt{y}}$ then:

$$\mathtt{U}(\bar{y}) \equiv ([x {\mapsto} \mathcal{D}_{\mathtt{U}_d}](\kappa \wedge \pi)) \ @ \ (\kappa_g \wedge \pi_g) \vee \Phi^g$$

is equivalent to:

$$\mathtt{U}(\bar{y}) \equiv (\mathtt{U}_d(x, \bar{v}) {*} \kappa \wedge \pi) \ @ \ (\kappa_g \wedge \pi_g) \ \vee \ \Phi^g$$

We first observe that variables local to the predicate definition, not part of the predicate arguments, are implicitly existentially quantified.As mentioned, we use the $\mathcal{D}_{\mathtt{U}_d}$ notation as a visual aid, to identify an instance of the predicate $\mathtt{U}_d$ whose root pointer $\mathtt{x}$ is reachable but has been neither read nor written to. By expanding the notation, the equivalence to be proven becomes:

$$\exists x.(\mathtt{U}_d(x, \bar{v}) {*} \kappa \wedge \pi) {\equiv} \exists \mathtt{f}_{\mathtt{v}}.[\mathcal{D}_{\mathtt{U_d}} \mapsto \mathtt{f}_{\mathtt{v}}]([\mathtt{x} {\mapsto} \mathcal{D}_{\mathtt{U_d}}](\kappa \wedge \pi)) {*} \mathtt{U_d}(\mathtt{f}_{\mathtt{v2}}, \ldots)$$

By applying the predicate definitions:

$$\exists \mathtt{x}.(\top {*} \kappa \wedge \pi) {\equiv} \exists \mathtt{f}_{\mathtt{v}}.[\mathcal{D}_{\mathtt{U_d}} \mapsto \mathtt{f}_{\mathtt{v}}]([\mathtt{x} {\mapsto} \mathcal{D}_{\mathtt{U_d}}](\kappa \wedge \pi)) {*} \top$$

Which holds trivially.

– Eliminating useless parameters: We need to show that if at this step a predicate $\mathtt{P}(\bar{\mathtt{x}}) \equiv \Delta_1$ is distilled into $\mathtt{Q}(\bar{\mathtt{x}}') \equiv \Delta_2$ then $\mathtt{P}'(\bar{\mathtt{x}}) \equiv \mathtt{Q}(\bar{\mathtt{x}}')$ that is, Δ_1 holds iff Δ_2 holds. By construction $\bar{\mathtt{x}}' = \bar{\mathtt{x}} \setminus \bar{\mathtt{z}}$ and $\exists \bar{z}.\Delta_1 \vdash \Delta_2$ and also $\Delta_2 \vdash \exists \bar{z}.\Delta_1$ which leads to $\Delta_2 \equiv \exists \bar{z}.\Delta_1$ and by the soundness of the flow analysis used to detect that variables $\bar{z}$ are not used in Δ_1 it follows that $\Delta_1 \equiv \exists \bar{z}.\Delta_1$.

– Re-using predicates: We need to show that if at this step a predicate $\mathtt{P}(\bar{\mathtt{x}}) \equiv \Delta_1$ is found to be equivalent with $\mathtt{Q}(\bar{\mathtt{x}}') \equiv \Delta_2$ then Δ_1 holds iff Δ_2 holds. By the premise of this normalization step, $\Delta_1 \vdash \mathtt{Q}(\bar{\mathtt{x}}')$ and also $\Delta_2 \vdash \mathtt{P}(\bar{\mathtt{x}})$ which leads to $\Delta_2 \equiv \Delta_1$.
– Predicate splitting: Soundness follows from the construction. Given a predicate $\mathtt{P}(\bar{\mathtt{x}})$ we need to show that if the bi-abduction succeeds in discovering definitions for $\mathtt{U}_1(\bar{\mathtt{x}})$ and $\mathtt{U}_2(\bar{\mathtt{x}})$ such that $\mathtt{P}(\bar{\mathtt{x}}) \vdash \mathtt{U}_1(\bar{\mathtt{x}}) * \mathtt{U}_2(\bar{\mathtt{x}})$ and if the derived predicate definitions can be used to prove $\mathtt{P}(\bar{\mathtt{x}}) \equiv \mathtt{U}_1(\bar{\mathtt{x}}) * \mathtt{U}_2(\bar{\mathtt{x}})$ which follows from the soundness of the bi-abduction and of the entailment methods.

F Limitation Example

To show limitation of our current proposal, we highlight an example in Fig. 9 where we are currently unable to prove memory safety. The lists $\mathtt{l}_1$ and $\mathtt{l}_2$ are actually from overlapping heap memory since they are computed from the same list, `list`. Our current modular procedure infers two disjoint lists as pre-condition of the loop, with $\mathtt{l}_1$ being longer than $\mathtt{l}_2$ through a composite predicate (similar to the `zip` example). Precondition of this loop cannot be currently proven inside `g_slist_sort_real`. To analyse this example successfully, our current tool would have to be extended to infer immutability and size property of heap data structure, so that certain heap overlaps can be handled. In particular, we would need to infer the following more precise specification:

```
struct GSList*
g_slist_sort_real(struct GSList* list, ...)
{ struct GSList *l1, *l2;
  if (!list) return NULL;if(!list->next)
     return list;
  l1 = list; l2=list->next;
  while ((l2=l2->next) ≠ NULL){
    if ((l2=l2->next) == NULL) break;
    l1=l1->next; } //failed precondition
 l2=l1->next; ...}
```

Fig. 9. `g_slist_sort_real` from `gslist.c`

$$\mathtt{requires}\ (\mathtt{lseg}(\mathtt{l}_2,\mathtt{NULL},\mathtt{n})@\mathtt{L} \wedge \mathtt{lseg}(\mathtt{l}_1,\mathtt{q},\mathtt{n}-1)@\mathtt{L}) \wedge \mathtt{n}>0$$
$$\mathtt{ensures}\ \mathtt{l}_2'=\mathtt{NULL} \wedge \mathtt{l}_1'=\mathtt{q}$$

Here, list segments of $\mathtt{l}_1$ and $\mathtt{l}_2$ of lengths $\mathtt{n}$ and $\mathtt{n}-1$ are overlapping but accessed in read-mode via the $@\mathtt{L}$ annotation. $\mathtt{l}_1'$ and $\mathtt{l}_2'$ captures the updated variables at exit of loop. Furthermore, there are other examples which rely on both shape and pure properties (e.g. sortedness or size) for memory safety.