Shape Analysis via Second-Order Bi-Abduction

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:

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

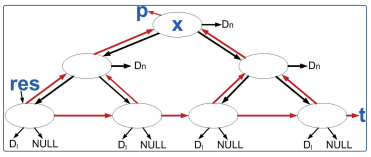

Fig. 1. An example of G(x,p,res,t)

The derived pre-predicate H captures a binary tree-like shape that would be traversed by the method. $\texttt{x}{\mapsto}\texttt{tree}(\mathcal{D}_\texttt{p},\mathcal{D}_\texttt{l},\texttt{r},\mathcal{D}_\texttt{n})$ denotes that x refers to a tree node with its parent,l,r and next fields being $\mathcal{D}_\texttt{p}$, $\mathcal{D}_\texttt{l}$, r and $\mathcal{D}_\texttt{n}$, respectively. We use dangling references, such as $\mathcal{D}_\texttt{l}$, $\mathcal{D}_\texttt{p}$, $\mathcal{D}_\texttt{n}$, 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 report [21] contains more details of our tool.

[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.

2 Logic Syntax for Shape Specification

Separation logic is an extension of Hoare logic for reasoning with heap-based programs [20,28]. We outline below the fragment underlying the proposed analysis:

$$
\begin{array}{lll}
\text{Disj. formula} & \Phi & ::= \Delta \mid \Phi_1 \vee \Phi_2 \\
\text{Guarded Disj.} & \Phi^g & ::= \Delta \mid (\Delta \,@\, (\kappa \wedge \pi)) \mid {\Phi^g}_1 \vee {\Phi^g}_2 \\
\text{Conj. formula} & \Delta & ::= \exists \bar{v} \cdot (\kappa \wedge \pi) \\
\text{Spatial formula} & \kappa & ::= \mathtt{emp} \mid \top \mid v \mapsto c(\bar{v}) \mid \mathtt{P}(\bar{v}) \mid \mathtt{U}(\bar{v}) \mid \kappa_1 * \kappa_2 \\
\text{Pure formula} & \pi & ::= \alpha \mid \neg \alpha \mid \pi_1 \wedge \pi_2 \\
\text{Var (Dis)Equality} & \alpha & ::= v | v_1 = v_2 | v = \mathtt{NULL} | v_1 \neq v_2 | v \neq \mathtt{NULL} \\
\text{Pred. Defn.} & \mathtt{P}^{\mathtt{def}} & ::= \mathtt{P}(\bar{v}) \equiv \Phi^g \\
\text{Pred. Dict.} & \Gamma & ::= \{\mathtt{P}_1^{\mathtt{def}}, \ldots, \mathtt{P}_n^{\mathtt{def}}\}
\end{array}
$$

$\mathtt{P} \in$ Known Predicates $\quad \mathtt{U} \in$ Unknown Predicates

$c \in$ Data Nodes $\quad v \in$ Variables $\quad \bar{v} \equiv v_1 \ldots v_n$

We introduce $\Delta \,@\, (\kappa \wedge \pi)$, a special syntactic form called *guarded heap* that capture a heap context $\kappa \wedge \pi$ in which Δ holds. Thus, $\Delta \,@\, (\kappa \wedge \pi)$ 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, $\mathtt{P}(\bar{v})$, 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, $\mathtt{U}(\bar{v})$, 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 $\mathtt{P}(\bar{v})$ 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: $\Delta_{\mathtt{pre}} * \Delta_{\mathtt{ante}} \vDash \Delta_{\mathtt{conseq}} * \Delta_{\mathtt{frame}}$. 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 $\Delta_{\texttt{ante}}$ and a consequent $\Delta_{\texttt{conseq}}$ the resulting entailment procedure infers both the frame residue $\Delta_{\texttt{frame}}$ 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:

$$\mathcal{R} \wedge \Delta_{\texttt{ante}} \vdash \Delta_{\texttt{conseq}} * \Delta_{\texttt{frame}}$$

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

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

$$\texttt{U(x)} \vdash \texttt{x}{\mapsto}\texttt{snode(n)} \rightsquigarrow (\texttt{U(x)}{\Rightarrow}\texttt{x}{\mapsto}\texttt{snode(n)} * \texttt{U}_0\texttt{(n)},\ \texttt{U}_0\texttt{(n)})$$

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 $\texttt{U}_0\texttt{(n)}$ predicate. The data structure `snode` is defined as `struct snode` $\{$ `struct snode`* `next`$\}$. A simple example of the second scenario is shown next.

$$\texttt{x}{\mapsto}\texttt{snode(NULL)} * \texttt{y}{\mapsto}\texttt{snode(NULL)} \vdash \texttt{U}_1\texttt{(x)} \rightsquigarrow (\texttt{x}{\mapsto}\texttt{snode(NULL)}{\Rightarrow}\texttt{U}_1\texttt{(x)},\ \texttt{y}{\mapsto}\texttt{snode(NULL)})$$

The generated relational assumption depicts a *folding* process for unknown $\texttt{U}_1\texttt{(x)}$ 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.

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

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, $\texttt{H}_\texttt{p}$ and $\texttt{H}_\texttt{n}$, created during the code verification process. All relational assumptions are of the form $\Delta_{\texttt{lhs}} \Rightarrow \Delta_{\texttt{rhs}}$, except for (`A3`) which has the form $\Delta_{\texttt{lhs}} \Rightarrow \Delta_{\texttt{rhs}}$ @ $\Delta_\texttt{g}$ where $\Delta_\texttt{g}$ denotes a heap guard condition. Such heap guard condition allows more precise pre-conditions to be synthesized (e.g. $\texttt{H}_\texttt{n}$ in (`A3`)), and is shorthand for $\Delta_{\texttt{lhs}} * \Delta_\texttt{g} \Rightarrow \Delta_{\texttt{rhs}} * \Delta_\texttt{g}$.

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

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

(A2). $\mathtt{H(x,q)} \wedge \mathtt{x{\neq}NULL} \Rightarrow$
$\mathtt{x{\mapsto}node(x_p, x_n){*}H_p(x_p, q){*}H_n(x_n, q)}$

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

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

(a)

(α_1). $\mathtt{H(x,q)}$

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

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

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

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

(b)

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

verification requires that the postcondition $\mathtt{G(x,q)}$ be established by the program state (α_2), generating the relational assumption (A1) via the following entailment:

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

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):

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

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 $\mathtt{H(x_n,x)}$ of the recursive call $\mathtt{sll2dll(x{\text{->}}next, x)}$ at the program state (α_4):

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

Note that the heap guard $\mathtt{x{\mapsto}node(q, x_n)}$ 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 $\mathtt{G(x,q)}$, generating the relational assumption (A4):

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

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):

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

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 $\mathcal{D}_p$ corresponds to a *dangling reference* introduced: $\mathtt{H(x,q)} \equiv \mathtt{emp{\wedge}x{=}NULL{\vee}x{\mapsto}node}(\mathcal{D}_p, \mathtt{x_n}) * \mathtt{H(x_n, x)}$. 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 $\mathtt{H_2(x)} \equiv \mathtt{emp{\wedge}x{=}NULL{\vee}x{\mapsto}node}(\mathcal{D}_p, \mathtt{x_n}){*}\mathtt{H_2(x_n)}$.

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 $\mathtt{U}(\mathtt{v}_1,..,\mathtt{v}_n,\mathtt{w}_1\#,..,\mathtt{w}_m\#)$, where parameters $\mathtt{v}_1,..,\mathtt{v}_n$ are unannotated and parameters $\mathtt{w}_1,..,\mathtt{w}_m$ are #-annotated. From the perspective of unfolding, we permit each variable from $\mathtt{v}_1,..,\mathtt{v}_n$ to be instantiated at most once (we call them *instantiatable*), while variables $\mathtt{w}_1,..,\mathtt{w}_m$ 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 $\mathtt{v}_1,..,\mathtt{v}_n$ and would stop whenever references to $\mathtt{w}_1,..,\mathtt{w}_m$ 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:

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

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.

$$\mathtt{H}(\mathtt{x},\mathtt{q}\#)\wedge\mathtt{x}\neq\mathtt{NULL}\vdash\mathtt{x}\mapsto\mathtt{node}(\mathtt{x}_p,\mathtt{x}_n)\rightsquigarrow(\mathtt{A2},\ \Delta_1)\qquad(\mathtt{E2}')$$

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

$$\mathtt{A2}\ \equiv\ \mathtt{H}(\mathtt{x},\mathtt{q}\#)\wedge\mathtt{x}\neq\mathtt{NULL}\Rightarrow\mathtt{x}\mapsto\mathtt{node}(\mathtt{x}_p,\mathtt{x}_n)*\mathtt{H}_p(\mathtt{x}_p,\mathtt{q}\#)*\mathtt{H}_n(\mathtt{x}_n,\mathtt{q}\#)$$

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

$$\mathtt{H}(\mathtt{x},\mathtt{q})\wedge\mathtt{x}\neq\mathtt{NULL}\vdash\mathtt{x}\mapsto\mathtt{node}(\mathtt{x}_p,\mathtt{x}_n)\rightsquigarrow(\mathtt{A2}',\ \Delta_1')$$

We get $\mathtt{A2}'\ \equiv\ \mathtt{H}(\mathtt{x},\mathtt{q})\wedge\mathtt{x}\neq\mathtt{NULL}\Rightarrow\mathtt{x}\mapsto\mathtt{node}(\mathtt{x}_p,\mathtt{x}_n)*\mathtt{H}_p(\mathtt{x}_p,\mathtt{q}\#)*\mathtt{H}_n(\mathtt{x}_n,\mathtt{q}\#)*\underline{\mathtt{U}_2(\mathtt{q},\mathtt{x}\#)}$, where the unfolding process creates extra unknown predicate $\mathtt{U}_2(\mathtt{q},\mathtt{x}\#)$ 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 $\mathtt{lseg}(\mathtt{x},\mathtt{p})$ predicate where the parameter p is a continuation field.

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

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:

$$\mathtt{U(x)} \vdash \mathtt{ll(x)} \rightsquigarrow (\mathtt{U(x)} \Rightarrow \mathtt{ll(x)},\ \mathtt{emp}) \quad \mathtt{U(x)} \vdash \mathtt{lseg(x,p)} \rightsquigarrow (\mathtt{U(x)} \Rightarrow \mathtt{lseg(x,q)} * \mathtt{U_2(q)},\ \mathtt{U_2(p)})$$

The first predicate $\mathtt{ll(x)}$ did not have a continuation field. Hence, we did not generate any extra unknown predicate. The second predicate $\mathtt{lseg(x,p)}$ did have a continuation field p, and we generated an extra unknown predicate $\mathtt{U_2(p)}$ 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 $\mathtt{U_1}(\bar{\mathtt{v}}, \bar{\mathtt{w}}\#)$ in the consequent, a corresponding assumption $\Delta \Rightarrow \mathtt{U_1}(\bar{\mathtt{v}}, \bar{\mathtt{w}}\#)$ @ Δ_g is inferred where Δ contains unknown predicates with at least one instantiatable parameters from $\bar{\mathtt{v}}$, or heaps *reachable* from $\bar{\mathtt{v}}$ (via either any data fields or parameters of known predicates) but stopping at non-instantiatable variables $\bar{\mathtt{w}}\#$; 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 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 (E3):

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

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 $\bar{w}$.

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

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 $q{=}$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 r with the unknown predicate 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

U. The fourth and fifth lines compute relevant pure formulas and generate the assumption, respectively. Finally, the unknown predicates $\kappa_{\texttt{fields}}$ and $\kappa_{\texttt{rem}}$ 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 $\texttt{reach}(\bar{\texttt{w}}, \kappa_1 \wedge \pi_1, \bar{\texttt{z}}\#)$ extracts portions from the antecedent heap (κ_1) that are (1) unknown predicates containing at least one instantiatable parameter from $\bar{\texttt{w}}$; or (2) point-to or known predicates reachable from $\bar{\texttt{w}}$, but not reachable from $\bar{\texttt{z}}$. In our running example (the entailment (E3′) on last page), the function $\texttt{reach}(\{\texttt{x}_\texttt{n}\}, \texttt{x} \mapsto \texttt{node}(\texttt{q}, \texttt{x}_\texttt{n}) * \texttt{H}_\texttt{p}(\texttt{x}_\texttt{p}, \texttt{q}\#) * \texttt{H}_\texttt{n}(\texttt{x}_\texttt{n}, \texttt{q}\#) \wedge \texttt{x} \neq \texttt{NULL}, \{\texttt{x}\#\})$ is used to obtain $\texttt{H}_\texttt{n}(\texttt{x}_\texttt{n}, \texttt{q}\#)$. More detail on this function is in the report [21]. The $\texttt{heaps}(\Delta)$ function enumerates all known predicate instances (of the form $\texttt{P}(\bar{\texttt{v}})$) and points-to instances (of the form $\texttt{r} \mapsto \texttt{c}(\bar{\texttt{v}})$) in Δ. The function $root(\kappa)$ is defined as: $root(\texttt{r} \mapsto \texttt{c}(\bar{\texttt{v}})) = \{\texttt{r}\}$, $root(\texttt{P}(\texttt{r}, \bar{\texttt{v}})) = \{\texttt{r}\}$. In the first line, heaps of LHS are separated into the assumption κ_{11} and the residue κ_{12}. Second, heap guards (and their root pointers) are inferred based on κ_{12} and the #-annotated parameters $\bar{\texttt{z}}$. The assumption is generated in the third line and finally, the residual heap is used to discharge the remaining heaps of RHS.

$$\frac{\begin{array}{c}[\textsc{so-ent-fold}] \\ \kappa_{11} = \texttt{reach}(\bar{\texttt{w}}, \kappa_1 \wedge \pi_1, \bar{\texttt{z}}\#) \quad \exists \kappa_{12} \cdot \kappa_1 = \kappa_{11} * \kappa_{12} \\ \kappa_\texttt{g} = *\{\kappa \mid \kappa \in \texttt{heaps}(\kappa_{12}) \wedge root(\kappa) \subseteq \bar{\texttt{z}}\} \quad \bar{\texttt{r}} = \bigcup_{\kappa \in \kappa_\texttt{g}} root(\kappa) \\ \sigma \equiv (\kappa_{11} \wedge \texttt{slice}(\bar{\texttt{w}}, \pi_1) \Rightarrow \texttt{U}_\texttt{c}(\bar{\texttt{w}}, \bar{\texttt{z}}\#) \,@\, \kappa_\texttt{g} \wedge \texttt{slice}(\bar{\texttt{r}}, \pi_1)) \\ \kappa_{12} \wedge \pi_1 \vdash \kappa_2 \wedge \pi_2 \rightsquigarrow (\mathcal{R},\ \Delta_R)\end{array}}{\kappa_1 \wedge \pi_1 \vdash \texttt{U}_\texttt{c}(\bar{w}, \bar{z}\#) * \kappa_2 \wedge \pi_2 \rightsquigarrow (\sigma \wedge \mathcal{R},\ \Delta_R)}$$

Fig. 4. Bi-Abductive Folding.

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*).

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

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: $\vdash \{\Delta_\texttt{pre}\}\ e\ \{\mathcal{R}, \Delta_\texttt{post}\}$, 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. x′) denotes the latest value (of the program variable x). The formula $\Delta_1 *_{\bar{\texttt{v}}} \Delta_2$ denotes $\exists \bar{\texttt{r}} \cdot ([\bar{\texttt{r}}/\bar{\texttt{v}}']\Delta_1) * ([\bar{\texttt{r}}/\bar{\texttt{v}}]\Delta_2)$ (see [9]).

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 $\texttt{solve}(\mathcal{R}) = \texttt{PRED\_NORM}(\texttt{PRED\_SYN}(\mathcal{R}))$. After the method

$$\frac{\begin{array}{c}[\textbf{SA-CALL}]\\ \texttt{t}_0\ mn\ ((\texttt{ref}\ \texttt{t}_\texttt{i}\ \texttt{v}_\texttt{i})_{\texttt{i}=1}^{\texttt{m}-1}, (\texttt{t}_\texttt{j}\ \texttt{v}_\texttt{j})_{\texttt{j}=\texttt{m}}^{\texttt{n}})\ \ \Phi_{pr}\ \ \Phi_{po};\ \{e\} \in \texttt{Prog}\\ \rho{=}[\texttt{v}'_\texttt{k}/\texttt{v}_\texttt{k}]_{\texttt{k}=1}^{\texttt{n}}\quad \Phi'_{pr} = \rho(\Phi_{pr})\quad \texttt{W}{=}\{\texttt{v}_1,..,\texttt{v}_{\texttt{m}-1}\}\quad \texttt{V}{=}\{\texttt{v}_\texttt{m},..,\texttt{v}_\texttt{n}\}\\ \Delta \vdash \Phi'_{pr} \rightsquigarrow (\mathcal{R},\ \Delta_2)\quad \Delta_3{=}(\Delta_2 \wedge \bigwedge_{\texttt{i}=\texttt{m}}^{\texttt{n}}(\texttt{v}'_\texttt{i} = \texttt{v}_\texttt{i}))\ \ *_{\texttt{V}\cup\texttt{W}}\ \Phi_{po}\end{array}}{\vdash \{\Delta\}\ mn(\texttt{v}_1,..,\texttt{v}_{\texttt{m}-1},\texttt{v}_\texttt{m},..,\texttt{v}_\texttt{n})\ \{\mathcal{R},\Delta_3\}}$$

$$\frac{\begin{array}{c}[\textbf{SA-FLD-RD}]\\ \texttt{data c}\ \{\texttt{t}_1\ \texttt{f}_1,..,\texttt{t}_\texttt{n}\ \texttt{f}_\texttt{n}\} \in \texttt{Prog}\\ \Delta_1 \vdash \texttt{x}'{\mapsto}\texttt{c}(\texttt{v}_1..\texttt{v}_\texttt{n}) \rightsquigarrow (\mathcal{R},\ \Delta_3)\\ \Delta_4{=}\exists \texttt{v}_1..\texttt{v}_\texttt{n} \cdot (\Delta_3 * \texttt{x}'{\mapsto}\texttt{c}(\texttt{v}_1..\texttt{v}_\texttt{n}) \wedge \texttt{res}{=}\texttt{v}_\texttt{i})\end{array}}{\vdash \{\Delta_1\}\ \texttt{x.f}_\texttt{i}\ \{\mathcal{R},\Delta_4\}}$$

$$\frac{\begin{array}{c}[\textbf{SA-METH}]\\ \vdash \{\Phi_{pr} \wedge \bigwedge(\texttt{u}'{=}\texttt{u})^*\}\ e\ \{\mathcal{R}_1,\Delta_1\}\\ \Delta_1 \vdash \Phi_{po} \rightsquigarrow (\mathcal{R}_2,\ \Delta_2)\\ \Gamma = \texttt{solve}(\mathcal{R}_1 \cup \mathcal{R}_2)\end{array}}{\texttt{t}_0\ mn\ ((\texttt{t u})^*)\ \ \Phi_{pr}\ \Phi_{po}\ \{e\}}$$

Fig. 6. Several Hoare Rules

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 [syn-$*$] notation to name refinement rules. For space reasons, we describe some rules and leave the rest to the report [21]. Steps that are left out include: (i) *sort-group* to decide on the transformation order of relational assumptions; (ii) rules 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 [**syn-base**] 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 [**syn-case**], [**syn-group-pre**], and [**syn-pre-def**] on σ
  $\mathcal{R}_{\texttt{pre}}, \mathcal{R}_{\texttt{post}} \leftarrow$ 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 [**syn-group-post**], [**syn-post-def**] on σ
  discharge $\texttt{U}^{\texttt{post}}$ obligations
  $\mathcal{R}_{\texttt{post}} \leftarrow \mathcal{R}_{\texttt{post}} \setminus \sigma \qquad \Gamma \leftarrow \Gamma \cup \{\texttt{U}^{\texttt{post}}_{\texttt{def}}\}$
 end while
 return Γ
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 $\mathtt{U}^{\mathtt{pre}}$ is separated from post-predicate $\mathtt{U}^{\mathtt{post}}$. Base splitting can be formalized as follows:

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

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 $\mathtt{is\_base}(\bar{\mathtt{x}},\pi)$ holds. It signifies that π contains a base case formula for some pointer(s) in $\bar{\mathtt{x}}$. Note that $\mathtt{is\_base}(\bar{\mathtt{x}},\pi)$ holds if and only if ($\exists\ \mathtt{v}\in\bar{\mathtt{x}}.\ \pi\vdash\mathtt{v}=\mathtt{NULL}$) or ($\exists\mathtt{v}_1,\mathtt{v}_2\in\bar{\mathtt{x}}.\pi\vdash\mathtt{v}_1=\mathtt{v}_2$). 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:

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

(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 $\mathit{pars}(\kappa)$ (the 2nd line) retrieves parameters: $\mathit{pars}(\mathtt{r}{\mapsto}\mathtt{c}(\bar{\mathtt{v}}))=\bar{\mathtt{v}}$, $\mathit{pars}(\mathtt{P}(\mathtt{r},\bar{\mathtt{v}}))=\bar{\mathtt{v}}$.

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

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

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:

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

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, $\Delta_1 \wedge \Delta_2$ [28]. 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 $\Rightarrow^{\bar{\mathtt{x}}}_{\wedge}$ to the technical report [21]. 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 [syn-group-pre] 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 [syn-pre-def] below.

$$\frac{\mathtt{U}(\bar{x})\wedge\pi_1 \Rightarrow \Phi_1^g \quad \mathtt{U}(\bar{x})\wedge\pi_2 \Rightarrow \Phi_2^g \quad \pi_1\wedge\pi_2 \Rightarrow \mathtt{FALSE}}{\mathtt{U}(\bar{x}) \wedge (\pi_1 \vee \pi_2) \Rightarrow \Phi_1^g \wedge \pi_1 \vee \Phi_2^g \wedge \pi_2}\ [\text{syn-group-pre}] \qquad \frac{\mathtt{U}^{\mathtt{pre}}(\bar{x}) \Rightarrow \Phi^g \qquad \mathtt{no\_post}(\Phi^g)}{\mathtt{U}^{\mathtt{pre}}(\bar{x}) \equiv \Phi^g}\ [\text{syn-pre-def}]$$

For the sll2dll example, applying the [syn-group-pre] rule to (A2) and (A1a) yields:

$$(\mathtt{A5}).\ \mathtt{H}(\mathtt{x},\mathtt{q}) \Rightarrow \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{x_p},\mathtt{x_n}) * \mathtt{H_p}(\mathtt{x_p},\mathtt{q}) * \mathtt{H_n}(\mathtt{x_n},\mathtt{q}) \ \vee\ \mathtt{emp} \wedge \mathtt{x}{=}\mathtt{NULL}$$

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 [syn-group-post] rule shown below. By exhaustively applying [syn-group-post] rule all assumptions relating to predicate $\mathtt{U}^{\mathtt{post}}$ get condensed into an assumption of the form: $\Delta_1 \vee \ldots \vee \Delta_n \Rightarrow \mathtt{U}^{\mathtt{post}}(\bar{x})$. This may then be used to confirm the post-predicate by generating the predicate definition via the [syn-post-def] rule.

$$\frac{\Delta_a \Rightarrow \mathtt{U}^{\mathtt{post}}(\bar{x}) \qquad \Delta_b \Rightarrow \mathtt{U}^{\mathtt{post}}(\bar{x})}{\Delta_a \vee \Delta_b \Rightarrow \mathtt{U}^{\mathtt{post}}(\bar{x})}\ [\text{syn-group-post}] \qquad \frac{\Delta_1 \vee \ldots \vee \Delta_n \Rightarrow \mathtt{U}^{\mathtt{post}}(\bar{x})}{\mathtt{U}^{\mathtt{post}}(\bar{x}) \ \equiv\ \Delta_1 \vee \ldots \vee \Delta_n}\ [\text{syn-post-def}]$$

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

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

5.4 Predicate Normalization for More 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 report [21]. 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 report [21]. For brevity, we introduce the notation $\mathcal{R}(\Gamma)$ to denote a set of predicate instantiations $\Gamma{=}\{\mathtt{U}_1(\bar{\mathtt{v}}_1){\equiv}\Delta_1, ..\mathtt{U}_n(\bar{\mathtt{v}}_n){\equiv}\Delta_n\}$ 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 $\mathtt{U}^{\mathtt{pre}}$,$\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{v}}){\Rightarrow}\Phi^g$ and if our algorithm returns a solution Γ, then $(\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{v}}){\equiv}\Phi^g){\in}\,\Gamma$. Similarly, if $\mathcal{R}$ has a sole post-assumption on $\mathtt{U}^{\mathtt{post}}$, $\Phi{\Rightarrow}\mathtt{U}^{\mathtt{post}}$ and if solution Γ is returned, then $(\mathtt{U}^{\mathtt{post}}(\bar{\mathtt{v}}){\equiv}\Phi){\in}\,\Gamma$.*

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

Theorem 6.1 (Soundness) *If $\Delta_a \vdash \Delta_c \rightsquigarrow (\mathcal{R},\ \Delta_f)$ and $\Gamma{=}\mathtt{PRED\_NORM}(\mathtt{PRED\_SYN}(\mathcal{R}))$ 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 [27] 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-,

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

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*)

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 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

[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.

	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

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]

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, see the report [21]. 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 [26] can infer shape and certain numerical properties but is limited to the linked list domain. The analyses from [32,11,4,15,3,24] are tailored to variants of lists and a fixed family of list interleavings. Likewise, Calcagno et al. [7] describes an analysis for determining lock invariants with only linked lists. Lee et al. [22] 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 [31]. 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. [26] 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 [30] 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

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

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, [29] 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. [23] which relies on inferred grammars to define the recursive backbone of the shape predicates. Although [23] 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. [25] 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. Q.L. Le, C. Gherghina, S. Qin, and W.N. Chin. Shape analysis via second-order bi-abduction. In *Technical Report*, Soc, NUS, February 2014. http://loris-7.ddns.comp.nus.edu.sg/∼project/s2/beta/src/TRs2.pdf.
22. O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *CAV*, 2011.
23. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, pages 124–140, 2005.

24. 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.
25. 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.
26. S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, pages 211–222, 2010.
27. 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.
28. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, pages 55–74, 2002.
29. N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS*, pages 284–302, 2005.
30. X. Rival and B.-Y. E. Chang. Calling context abstraction with shapes. In *POPL*, pages 173–186, 2011.
31. G. Rosu and A. Stefanescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012.
32. 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.