Verifying Heap-Manipulating Programs with Unknown Procedure Calls

Shengchao Qin[1], Chenguang Luo[2⋆], Guanhua He[2], Florin Craciun[1], and Wei-Ngan Chin[3]

[1] Teesside University, Middlesbrough TS1 3BA, UK
[2] Durham University
[3] National University of Singapore

Abstract. Verification of programs with invocations to unknown procedures is a practical problem, because in many scenarios not all codes of programs to be verified are available. Those unknown calls also pose a challenge for their verification. This paper addresses this problem with an attempt to verify the full functional correctness of such programs using pointer-based data structures. Provided with a Hoare-style specification $\{\Phi_{pr}\}$ prog $\{\Phi_{po}\}$ where program prog contains calls to some unknown procedure unknown, we infer a specification $mspec_u$ for unknown from the calling contexts, such that the problem of verifying prog can be safely reduced to the problem of proving that the procedure unknown (once its code is available) meets the derived specification $mspec_u$. The expected specification $mspec_u$ for the unknown procedure unknown is automatically calculated using an abduction-based shape analysis specifically designed for a combined abstract domain. We have also done some experiments to validate the viability of our approach.

1 Introduction

While automated verification of heap-manipulating programs remains a big challenge [16], significant advances have been seen recently since the emergence of separation logic [13]. For instance, SpaceInvader [3] can verify the pointer safety of a large portion of the Linux kernel and many device drivers using shared mutable data structures; THOR [10] employs additional numerical analysis to help gain better precision for data structure properties such as list length; Hip/Sleek [11] can verify more sophisticated properties involving both shape and numerical information, such as sortedness, height-balanced and red-black properties. These are all successful examples of verification/analysis of heap-manipulating programs, esp. those processing pointer-based shared mutable data structures.

However, a recent prevalent trend of component-based software engineering [7] poses great challenge for quality assurance and verification of programs. This methodology involves the integration of software components from both native development and third-parties, and thus the source code of some components/procedures might be unknown for verification. For example, some pro-

[⋆] Now with Citigroup Inc.

grams may have calls to third-party library procedures whose code is not accessible (e.g. in binary form). Some components may be invoked by remote procedure calls only with a native interface such as COM/DCOM [14]. Still, some components could be used for dynamic upgrading of running systems whose cost of being stopped/restarted is too expensive to bear [15]. Other scenarios include function pointers (e.g. in C), interface method invocation (e.g. in OO) and mobile code, which all contain procedures not available for static verification.

To verify such programs, existing approaches generally do not provide elegant solutions. For example, black-box testing [2] regards the unknown procedures as black-boxes to test their functionality, which cannot formally prove the absence of program bugs, therefore may not be enough for safety-critical systems. Likewise, specification mining [1] discovers possible specifications for the (unknown part of the) program by observing its execution and traces, which is also dynamically performed and bears the same problem. For static verifiers/analysers, SpaceInvader [3] simply assumes the program and the unknown procedure have disjoint memory footprints so that the unknown call can be safely ignored due to the hypothetical frame rule [12], whereas this assumption does not hold in many cases. Some methods [4, 6] try to take into account all possible implementations for the unknown procedure; however there can be too many such candidates in general, and hence the verification might be infeasible for large-scaled programs. Finally, some verifiers will just stop at the first unknown procedure call and provide an incomplete verification [11], which is obviously undesirable.

Approach and contributions. We propose a novel approach in this paper to verifying heap-manipulating programs calling unknown procedures. Given a specification $\mathsf{S} = \{\Phi_{pr}\}$ prog $\{\Phi_{po}\}$ where prog contains calls to an unknown procedure $\mathsf{unknown}$, we try to infer a specification S_u for $\mathsf{unknown}$ based on the calling context(s) of prog. The verification of prog against S can now be safely reduced to the verification of $\mathsf{unknown}$ against the inferred specification S_u, provided that the verification of the known fragments does not cause any problems. The inferred specification is subject to a later verification when an implementation or a specification for the unknown procedure becomes available. This is essentially an improvement of our previous work [8] by extending the program properties to be verified from simple pointer safety to full functional correctness of linked data structures. Such properties include structural numerical ones like size and height, relational numerical ones like sortedness, and multi-set ones like symbolic content. Our paper makes the following technical contributions:

- We propose a novel framework in a combined abstract domain (involving both shape and pure properties) for the verification of full functional correctness of programs with unknown calls.
- Our approach is essentially *top-down*, as it can be used to infer the specification for callee procedures based on the specification for the caller procedure. Hence it may benefit the general software development process as a complement for current *bottom-up* approaches [3, 11].
- We have invented an abduction mechanism which can be applied in this combined domain. It not only can infer shape-based anti-frames for an entailment, but also can discover corresponding pure information (numerical

and/or multi-set) as well. We also defined a partial order as a guidance for the quality of abduction results.

– We have conducted some initial experimental studies to test the viability and performance of our approach. Preliminary results show that our approach can derive expressive specifications which fully capture the behaviours of the unknown code in many cases.

In the following we will first illustrate our approach with an illustrative example and then describe its formal settings. Any technical details not described due to space limit can be found in our technical report [9].

2 The Approach

We first introduce our specification mechanism, followed by an illustrative example for the verification.

2.1 User-defined Predicates

Separation logic [13] extends Hoare logic to support reasoning about shared mutable data structures. It provides separation conjunction ($*$) to form formulae like $\mathtt{p_1} * \mathtt{p_2}$ to assert that two heaps described by $\mathtt{p_1}$ and $\mathtt{p_2}$ are domain-disjoint. Our abstract domain is founded on a hybrid logic of both separation logic and classical first-order logic to specify both separation and pure properties. Over this domain we allow user-defined inductive predicates. For example, with a data structure definition for a node in a list `data node { int val; node next; }`, we can define a predicate for a list with the content stored in its nodes as

$$\mathtt{root}{::}\mathtt{llB}\langle\mathtt{S}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset) \vee (\exists \mathtt{v},\mathtt{q},\mathtt{S_1}{\cdot}\mathtt{root}{::}\mathtt{node}\langle\mathtt{v},\mathtt{q}\rangle{*}\mathtt{q}{::}\mathtt{llB}\langle\mathtt{S_1}\rangle \wedge \mathtt{S}{=}\mathtt{S_1}{\sqcup}\{\mathtt{v}\})$$

The parameter `root` for the predicate `llB` is the root pointer referring to the list. Its content is denoted by the multi-set `S`. A uniform notation $\mathtt{p}{::}\mathtt{c}\langle\mathtt{v}^*\rangle$ is used for either a singleton heap or a predicate. If `c` is a data node, the notation represents a singleton heap, $\mathtt{p}{\mapsto}\mathtt{c}[\mathtt{v}^*]$, e.g. the $\mathtt{root}{::}\mathtt{node}\langle\mathtt{v},\mathtt{q}\rangle$ above. If `c` is a predicate name, then the data structure pointed to by `p` has the shape `c` with parameters $\mathtt{v}^*$, e.g., the $\mathtt{q}{::}\mathtt{llB}\langle\mathtt{S_1}\rangle$ above.

If users want to verify a sorting algorithm, they can incorporate sortedness property into the above predicate as follows:

$$\begin{aligned}&\mathtt{sllB}\langle\mathtt{S}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset) \vee \\ &\qquad (\mathtt{root}{::}\mathtt{node}\langle\mathtt{v},\mathtt{q}\rangle * \mathtt{q}{::}\mathtt{sllB}\langle\mathtt{S_1}\rangle \wedge \mathtt{S}{=}\{\mathtt{v}\}{\sqcup}\mathtt{S_1} \wedge (\forall \mathtt{u}{\in}\mathtt{S_1} \cdot \mathtt{v}{\leq}\mathtt{u}))\end{aligned}$$

where we use the following shortened notation: (i) default `root` parameter in LHS may be omitted, (ii) unbound variables, such as `q` and $\mathtt{S_1}$, are implicitly existentially quantified. Meanwhile, later we may still use underscore _ to denote an implicitly quantified variable. Such user-supplied predicates can be used to specify method specifications.

2.2 Illustrative Example

In this section, we illustrate informally, via an example, how our approach verifies a program by inferring the specification for the unknown procedure it invokes.

Example 1 (Motivating example). Our goal is to verify the procedure `sort` against the given specification shown in Figure 1. According to the specification, the procedure takes in a non-empty linked list `x` and returns a sorted list referenced as

```
0 node sort(node x) requires x::llB⟨S⟩ ensures res::sllB⟨S⟩
1 {  // res is the value returned by the procedure
1a    // Forward analysis begins with current state $\sigma$ : x::llB⟨S⟩
2   if (x == null) return null;
2a    // $\sigma$ : x::llB⟨S⟩ $\wedge$ x=null $\wedge$ res=null
2b    // Check whether current state meets the postcondition: $\sigma \vdash$ res::sllB⟨S⟩
2b    //  which succeeds; the verification on this branch terminates
3   else {
3a    // $\sigma$ : x::llB⟨S⟩ $\wedge$ x$\neq$null
3b    // Unknown call is now encountered (line 4); extract its precondition from $\sigma$:
3c    // $\Phi^{u}_{pr}$ := Local($\sigma$, {x}) := x::llB⟨S⟩ $\wedge$ x$\neq$null
3d    // Also distinguish the frame part not touched by unknown call:
3e    // $R_0$ := Frame($\sigma$, {x}) := emp $\wedge$ x$\neq$null
4   node y = unknown(x);
4a    // Immediately after the unknown call we know nothing about its effect, so
4b    // we begin to discover its post-effect starting from emp (saved in $\sigma'$):
4c    // $\sigma'_0$ : emp $\wedge$ x=a $\wedge$ y=$res_u$      $\sigma := R_0 * \sigma'_0$      $\sigma' := \sigma'_0$
4d    // Next instruction (y.next) requires y be a node
4e    // But the entailment checking $\sigma \vdash$ y::node⟨v, p⟩ fails
4f    // This requirement might be part of the unknown call's post-effect; we use
4g    // abduction to find it and add it to current state and unknown call's post:
4h    // $\sigma * [\sigma'_1] \triangleright$ y::node⟨v, p⟩ (s.t. $\sigma * \sigma'_1 \vdash$ y::node⟨v, p⟩ $*$ true)
4i    // $\sigma'_1$ : y::node⟨v, p⟩                  $\sigma := \sigma * \sigma'_1$      $\sigma' := \sigma' * \sigma'_1$
5   node z = y.next;
5a    // Current state $\sigma$ : y::node⟨v, z⟩
5b    // Next instruction invokes this procedure recursively and requires its pre, but
5c    //  $\sigma \vdash$ z::llB⟨$S_1$⟩ fails possibly due to lack of knowledge about unknown call
5d    // Again we use abduction to find the missing part of unknown call's post-effect
5e    // $\sigma * [\sigma'_2] \triangleright$ z::llB⟨$S_1$⟩ (s.t. $\sigma * \sigma'_2 \vdash$ z::llB⟨$S_1$⟩ $*$ true)
5f    // $\sigma'_2$ : z::llB⟨$S_1$⟩                  $\sigma := \sigma * \sigma'_2$      $\sigma' := \sigma' * \sigma'_2$
6   node w = sort(z);
6a    // Current state $\sigma$ : y::node⟨v, z⟩ $*$ w::sllB⟨$S_1$⟩ (w already refers to a sorted list)
7   y.next = w;
7a    // Current state $\sigma$ : y::node⟨v, w⟩ $*$ w::sllB⟨$S_1$⟩
8   return y;
8a    // $\sigma$ : y::node⟨v, w⟩ $*$ w::sllB⟨$S_1$⟩ $\wedge$ res=y; it should imply sort's postcondition
8b    // But $\sigma \vdash$ res::sllB⟨S⟩ still fails, suggesting more post-effect of unknown call
8c    // A final abduction is conducted to find it: $\sigma * [\sigma'_3] \triangleright$ res::sllB⟨S⟩
8d    // $\sigma'_3$ : S={v}$\sqcup S_1 \wedge \forall u \in S_1 \cdot v \leq u$   $\sigma := \sigma * \sigma'_3$      $\sigma' := \sigma' * \sigma'_3$
8e    // All abduction results will be combined at last to form unknown call's post
9   } }
9a // $\Phi^{u}_{pr}$ : a::llB⟨S⟩ $\wedge$ a$\neq$null (a is the unknown procedure's formal parameter)
9b // $\Phi^{u}_{po}$ : $res_u$::node⟨v, b⟩ $*$ b::llB⟨$S_1$⟩ $\wedge$ S={v}$\sqcup S_1 \wedge \forall u \in S_1 \cdot v \leq u$
```

Fig. 1. Verification of sort which invokes an unknown procedure unknown.

res. The (symbolic) content of these two lists are identical (S). Note that sort calls an unknown procedure unknown at line 4. As we do not have available knowledge about it, the discovery of its specifications is essential for both the verification and our understanding of the program (such that we may find out what sorting algorithm this procedure implements).

We conduct a forward analysis on the program body starting with the precondition $\mathtt{x}{::}\mathtt{llB}\langle\mathtt{S}\rangle$ (line 0). The results of our analysis (e.g. the abstract states) are marked as comments in the code. The analysis carries on until it reaches the unknown procedure call at line 4.

As afore-shown, the current state before line 4 is $\mathtt{x}{::}\mathtt{llB}\langle\mathtt{S}\rangle \wedge \mathtt{x}{\neq}\mathtt{null}$ (σ at line 3a). Then we want to discover the precondition for the unknown call from it. To do that, we split σ into two disjoint parts: the local part $\Phi^{\mathtt{u}}_{pr}$ (line 3c) that is depended on, and possibly mutated by, the unknown procedure; and the frame part $\mathtt{R_0}$ (line 3e) that is not accessed by the unknown procedure. Intuitively, the local part of a state w.r.t. a set of variables X is the part of the heap reachable from variables in X; while the frame part denotes the unreachable heap part. Thus we take $\Phi^{\mathtt{u}}_{pr}$ (line 3c) as a crude precondition for the unknown procedure. The frame part $\mathtt{R_0}$ is not touched by the unknown call and will remain in the post-state, as shown in line 4c.

At line 4c, the abstract state after the unknown call (σ) consists of two parts: one is the aforesaid frame $\mathtt{R_0}$ not accessed by the call, and the other is the procedure's postcondition which is unfortunately not available. Our next step is to discover the postcondition by examining the code fragment after the unknown call (lines 4a to 8e). For this task, a traditional approach is a backward reasoning from the caller's postcondition towards the unknown call's postcondition. However, this is proven infeasible for separation logic based shape domain by previous works [3], and hence we employ another approach with a forward reasoning from the unknown call towards the caller's postcondition, using *abduction* to discover the unknown call's postcondition.

Initially, we assume the unknown procedure having an empty heap σ'_0 as its postcondition[1], and gradually discover the missing parts of the postcondition during the symbolic execution of the code fragment after the unknown call. To do that, our analysis keeps track of a pair (σ, σ') at each program point, where σ refers to the current heap state, and σ' denotes the expected postcondition discovered so far for the unknown procedure. The notations $\sigma'_{\mathtt{i}}$ are used to represent parts of the discovered postcondition.

At line 5, y.next is dereferenced, whose value is then assigned to z. Such dereference causes a problem, as we have an empty heap beforehand (σ in line 4c). However, this is not necessarily due to a program error; it might be attributed to the fact that the unknown call's postcondition is still unknown. Therefore, our analysis performs an abduction (line 4h) to infer the missing part σ'_1 for σ such that $\sigma * \sigma'_1$ implies that y points to a node. As shown in line 4i, σ'_1 is inferred to be $\mathtt{y}{::}\mathtt{node}\langle\mathtt{v},\mathtt{p}\rangle$, which is accumulated into σ' as part of the expected

[1] Note that we introduce fresh logical variables a and $\mathtt{res_u}$ to record the value of x and y when unknown returns.

postcondition of the unknown procedure. (We will explain the details for abduction in Section 4.) Now the heap state combined with the inferred σ'_1 meets the requirement of the dereference, and thus the forward analysis continues.

At line 6, the procedure `sort` is called recursively. Here the current heap state still does not satisfy the precondition of `sort` (as shown in line 5c). Blaming the lack of knowledge about the unknown call's postcondition, we conduct another abduction (line 5e) to infer the missing part σ'_2 for σ such that $\sigma * \sigma'_2$ entails the precondition of `sort` w.r.t. some substitution $[\mathtt{z}/\mathtt{x}]$. Updated with the abduction result $\mathtt{z}{::}\mathtt{llB}\langle \mathtt{S}_1\rangle$, the program state now meets the precondition of `sort`, which is later transformed to $\mathtt{w}{::}\mathtt{sllB}\langle \mathtt{S}_1\rangle$ as the effect of sorting over `z`.

After that, line 7 links `y` and the sorted list `w` together. Then `y` is returned as the procedure's result at last. The corresponding state σ at line 8a is expected to establish the postcondition of `sort` for the overall verification to succeed. However, it does not (as shown in line 8b). Again this might be because part of the unknown call's postcondition is still missing. Therefore, we perform a final abduction (line 8c) to infer the missing σ'_3 as follows:

$$(\mathtt{y}{::}\mathtt{node}\langle \mathtt{v},\mathtt{w}\rangle * \mathtt{w}{::}\mathtt{sllB}\langle \mathtt{S}_1\rangle \wedge \mathtt{res}{=}\mathtt{y}) * [\sigma'_3] \rhd \mathtt{res}{::}\mathtt{sllB}\langle \mathtt{S}\rangle$$

such that $\sigma * \sigma'_3$ implies the postcondition. In this case, our abductor returns σ'_3 as a sophisticated pure constraint $\mathtt{S}{=}\{\mathtt{v}\}{\sqcup}\mathtt{S}_1 \wedge \forall \mathtt{u}{\in}\mathtt{S}_1{\cdot}\mathtt{v}{\leq}\mathtt{u}$ as the result which is then added into σ', as shown in line 8d.

Finally, we generate the expected pre/post-specification for the unknown procedure (lines 9a and 9b). The precondition is obtained from the local prestate of the unknown call, $\Phi^{\mathtt{u}}_{pr}$ at line 3c, by replacing all variables that are aliases of `a` with the formal parameter `a`. The postcondition is obtained from the accumulated abduction result, σ', after performing a similar substitution (which also involves formal parameter $\mathtt{res}_\mathtt{u}$). Our discovered specification for the unknown procedure `node unknown(node a)` is:

$$\Phi^{\mathtt{u}}_{pr} : \mathtt{a}{::}\mathtt{llB}\langle \mathtt{S}\rangle \wedge \mathtt{a}{\neq}\mathtt{null}$$
$$\Phi^{\mathtt{u}}_{po} : \exists \mathtt{b} \cdot \mathtt{res}_\mathtt{u}{::}\mathtt{node}\langle \mathtt{v},\mathtt{b}\rangle * \mathtt{b}{::}\mathtt{llB}\langle \mathtt{S}_1\rangle \wedge \mathtt{S}{=}\{\mathtt{v}\}{\sqcup}\mathtt{S}_1 \wedge \forall \mathtt{u}{\in}\mathtt{S}_1{\cdot}\mathtt{v}{\leq}\mathtt{u}$$

This derived specification has two implications. The first is that the entire program is verified on the condition that `unknown` meets such specification. The second is an improvement of our understanding on the behaviours of both the caller (`sort`) and the callee (`unknown`): the callee should choose the smallest element from its input list, and its way of choice decides the type of sorting for the caller (selection or bubble sort).

3 Language and Abstract Domain

To simplify presentation, we focus on a strongly-typed C-like imperative language in Figure 2. A program *Prog* consists of two parts: type declarations and method declarations. The type declarations *tdecl* can define either data type *datat* (e.g. `node`) or predicate *spred* (e.g. `llB`). The method declarations include *meth* and *munk*, of which the second contains invocations to unknown procedures while the first does not. The *spred* and *mspec* are defined in Figure 3.

Note that the language is expression-oriented, so the body of a method is an expression composed of standard instructions and constructors of an imperative

$$\begin{array}{lll}
\mathit{Prog} & ::= & \mathit{tdecl}\ \mathit{meth}\ \mathit{munk} \qquad\qquad \mathit{tdecl} ::= \mathit{datat} \mid \mathit{spred} \\
\mathit{datat} & ::= & \texttt{data}\ c\ \{\ \mathit{field}\ \} \qquad \mathit{field} ::= t\ x \qquad t ::= c \mid \tau \\
\mathit{meth} & ::= & t\ \mathit{mn}\ ((\boldsymbol{t\ x}); (\boldsymbol{t\ y}))\ \mathit{mspec}\ \{e\} \qquad \tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{void} \\
\mathit{munk} & ::= & t\ \mathit{mn}\ ((\boldsymbol{t\ x}); (\boldsymbol{t\ y}))\ \mathit{mspec}\ \{v\} \\
e & ::= & d \mid d[x] \mid x{=}e \mid e_1; e_2 \mid t\ x;\ e \mid \texttt{if}\ (x)\ e_1\ \texttt{else}\ e_2 \mid \texttt{while}\ x\ \{e\}\ \texttt{inv}\ \Delta \\
u & ::= & \mathit{unk}(\boldsymbol{x}; \boldsymbol{y}) \mid \mathit{unk}(\boldsymbol{x_0}; \boldsymbol{y_0}); e_1; \mathit{unk}_1(\boldsymbol{x_1}; \boldsymbol{y_1}); e_2; ...; e_{n-1}; \mathit{unk}_n(\boldsymbol{x_n}; \boldsymbol{y_n}) \mid \\
 & & \texttt{if}\ (x)\ v\ \texttt{else}\ e \mid \texttt{if}\ (x)\ e\ \texttt{else}\ v \mid \texttt{if}\ (x)\ v_1\ \texttt{else}\ v_2 \mid \texttt{while}\ x\ \{v\}\ \texttt{inv}\ \Delta \\
v & ::= & e_1; u; e_2 \\
d & ::= & \texttt{null} \mid k^{\tau} \mid x \mid \texttt{skip} \mid \texttt{new}\ c(\boldsymbol{x}) \mid \mathit{mn}(\boldsymbol{x}; \boldsymbol{y}) \\
d[x] & ::= & x.f \mid x.f{:=}z \mid \texttt{free}(x)
\end{array}$$

Fig. 2. A core (C-like) imperative language.

language. e is the (recursively defined) program constructor and d and $d[x]$ are atom instructions. Note also that the language allows both call-by-value and call-by-reference method parameters (which are separated with a semicolon ; where the ones before ; are call-by-value and the ones after are call-by-reference).

To address the unknown calls, we employ *unknown constructors* u and v to denote expressions that involve invocations to the unknown procedures ($\mathit{unk}(\boldsymbol{x}, \boldsymbol{y})$). An *unknown block* v is defined as a sequence of normal expressions sandwiching an *unknown expression* u, which can be a single unknown call, or a sequence of unknown calls, or an if-conditional statement/while loop containing an unknown block. Our aim is to discover the specifications for the unknown procedures in u and v to verify the whole program.

$$\begin{array}{lll}
\mathit{mspec} & ::= & \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \qquad \mathit{spred} ::= \texttt{root}{::}c\langle \boldsymbol{v} \rangle \equiv \Phi \\
\Delta & ::= & \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta \\
\Phi & ::= & \bigvee \boldsymbol{\sigma} \qquad\qquad\qquad\qquad \sigma ::= \exists \boldsymbol{v} \cdot \kappa \wedge \pi \\
\kappa & ::= & \texttt{emp} \mid v{::}c\langle \boldsymbol{v} \rangle \mid \kappa_1 * \kappa_2 \qquad \pi ::= \gamma \wedge \phi \\
\gamma & ::= & v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \texttt{true} \mid \gamma_1 \wedge \gamma_2 \\
\phi & ::= & \varphi \mid b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg \phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi \\
b & ::= & \texttt{true} \mid \texttt{false} \mid v \mid b_1 = b_2 \qquad a ::= s_1{=}s_2 \mid s_1{\leq}s_2 \\
s & ::= & k^{\texttt{int}} \mid v \mid k^{\texttt{int}} \times s \mid s_1 + s_2 \mid -s \mid \mathit{max}(s_1,s_2) \mid \mathit{min}(s_1,s_2) \ \mid\ |\mathtt{B}| \\
\varphi & ::= & v \in \mathtt{B} \mid \mathtt{B}_1{=}\mathtt{B}_2 \mid \mathtt{B}_1 \sqsubset \mathtt{B}_2 \mid \mathtt{B}_1 \sqsubseteq \mathtt{B}_2 \mid \forall v \in \mathtt{B} \cdot \phi \mid \exists v \in \mathtt{B} \cdot \phi \\
\mathtt{B} & ::= & \mathtt{B}_1 \sqcup \mathtt{B}_2 \mid \mathtt{B}_1 \sqcap \mathtt{B}_2 \mid \mathtt{B}_1 - \mathtt{B}_2 \mid \emptyset \mid \{v\}
\end{array}$$

Fig. 3. The specification language.

Our specification language (in Figure 3) allows (user-defined) shape predicates to specify both separation and pure properties. The shape predicates *spred* are constructed with disjunctive constraints Φ. We require that the predicates be well-formed [11].

A conjunctive abstract program state σ is composed of a heap (shape) part κ and a pure part π, where π consists of γ, ϕ and φ as aliasing, numerical and bag information, respectively. We use SH to denote a set of such conjunctive states. During our verification, the abstract program state at each program point will be a disjunction of σ's, denoted by Δ (and the set of such disjunctions $\mathcal{P}_{\mathsf{SH}}$). An abstract state Δ can be normalised to the Φ form [11].

The memory model of our specification formulae is adapted from the model given for "early versions" of separation logic [13], except that we have extensions to handle user-defined shape predicates and related pure properties. Meanwhile, for program variables in abstract states, we use unprimed ones to denote their initial values and primed ones for current values [9, 11].

4 Abduction

As shown in Section 2, when analysing the code after an unknown call, it is possible that the current state cannot meet the required precondition for the next instruction due to the lack of information about the unknown procedure. Therefore we need to infer the unknown procedure's specification with *abduction* (or abductive reasoning) [3, 5]. It works as follows: for a failed entailment checking $\sigma_1 \vdash \sigma_2 * \texttt{true}$, it attempts to compute an anti-frame σ', such that $\sigma_1 * \sigma' \vdash \sigma_2 * \texttt{true}$ succeeds. For instance, the entailment checking $\texttt{emp} \vdash \texttt{x}{::}\texttt{llB}\langle\texttt{S}\rangle$ fails as the antecedent contains an empty heap. Then $\texttt{x}{::}\texttt{llB}\langle\texttt{S}\rangle$ will be found to strengthen the antecedent and validate the entailment $\texttt{emp} * \texttt{x}{::}\texttt{llB}\langle\texttt{S}\rangle \vdash \texttt{x}{::}\texttt{llB}\langle\texttt{S}\rangle$.

An abduction $\sigma_1 * [\sigma'] \rhd \sigma_2$ can also be written as $\sigma_1 * [\sigma'] \rhd \sigma_2 * \sigma_3$, where σ_1 and σ_2 are inputs, σ' is the abduction result (the anti-frame), and σ_3 is the frame part resulted from the entailment checking $\sigma_1 * \sigma' \vdash \sigma_2$.

$$\frac{\sigma \nvdash \sigma_1 * \texttt{true} \quad \sigma_1 \vdash \sigma * \sigma' \quad \sigma * \sigma' \vdash \sigma_1 * \sigma_2}{\sigma * [\sigma'] \rhd \sigma_1 * \sigma_2}$$

$$\frac{\begin{array}{c}\sigma \nvdash \sigma_1 * \texttt{true} \quad \sigma_1 \nvdash \sigma * \texttt{true} \quad \sigma_0 \in \mathsf{unroll}(\sigma) \quad \mathsf{data\_no}(\sigma_0) \leq \mathsf{data\_no}(\sigma_1) \\ \sigma_0 \vdash \sigma_1 * \sigma' \text{ or } \sigma_0 * [\sigma'_0] \rhd \sigma_1 * \sigma' \quad \sigma''{=}\mathit{XPure}_1(\sigma') \quad \sigma \wedge \sigma'' \vdash \sigma_1 * \sigma_2\end{array}}{\sigma \wedge [\sigma''] \rhd \sigma_1 * \sigma_2}$$

$$\frac{\sigma \nvdash \sigma_1 * \texttt{true} \quad \sigma_1 \nvdash \sigma * \texttt{true} \quad \sigma_1 * [\sigma'_1] \rhd \sigma * \sigma' \quad \sigma''{=}\mathit{XPure}_1(\sigma') \quad \sigma \wedge \sigma'' \vdash \sigma_1 * \sigma_2}{\sigma \wedge [\sigma''] \rhd \sigma_1 * \sigma_2}$$

$$\frac{\sigma \nvdash \sigma_1 * \texttt{true} \quad \sigma_1 \nvdash \sigma * \texttt{true} \quad \sigma * \sigma_1 \nvdash \texttt{false}}{\sigma * [\sigma_1] \rhd \sigma_1 * \sigma_2}$$

Fig. 4. Abduction rules.

Our abduction rules given in Figure 4 deal with four different cases. The first rule triggers when the LHS (σ) does not imply the RHS (σ_1) but the RHS implies the LHS with some formula (σ') as the frame. This rule is quite general and applies in many cases, such as the state immediately after an unknown call where we start with $\texttt{emp}$ as the heap state. For the example above $\texttt{emp} \nvdash \texttt{x}{::}\texttt{llB}\langle\texttt{S}\rangle$, the RHS can entail the LHS with frame $\texttt{x}{::}\texttt{llB}\langle\texttt{S}\rangle$. The abduction then checks whether σ plus the frame information σ' entails σ_1 with some frame formula σ_2 ($\texttt{emp}$ in this example), and returns the result $\texttt{x}{::}\texttt{llB}\langle\texttt{S}\rangle$.

In the case described by the second rule, neither side implies the other, e.g. for $\texttt{x}{::}\texttt{sllB}\langle\texttt{S}\rangle$ as LHS (σ) and $\exists \texttt{p}, \texttt{u}, \texttt{v} \cdot \texttt{x}{::}\texttt{node}\langle\texttt{u}, \texttt{p}\rangle * \texttt{p}{::}\texttt{node}\langle\texttt{v}, \texttt{null}\rangle$ as RHS (σ_1). As the shape predicates in the antecedent σ are formed by disjunctions according to their definitions (like $\texttt{sllB}$), its certain disjunctive branches may imply σ_1. As the rule suggests, to accomplish abduction $\sigma * [\sigma''] \rhd \sigma_1 * \sigma_2$, we first unfold σ ($\sigma_0 \in \mathsf{unroll}(\sigma)$) and try entailment or further abduction with the results (σ_0) against σ_1. If it succeeds with a frame σ', then we first obtain a pure approximation of σ' with *XPure* [11], and confirm the abduction by ensuring $\sigma \wedge \sigma'' \vdash \sigma_1 * \sigma_2$, for some σ_2. For the example above, the abduction returns $|\texttt{S}|{=}2$ as the anti-frame σ' and discovers the nontrivial frame $\texttt{S}{=}\{\texttt{u}, \texttt{v}\} \wedge \texttt{u}{\leq}\texttt{v}$

(σ_2). Note the function $\mathsf{data\_no}$ returns the number of data nodes in a state, e.g. it returns one for $\mathtt{x}{::}\mathtt{node}\langle\mathtt{v},\mathtt{p}\rangle * \mathtt{p}{::}\mathtt{llB}\langle\mathtt{T}\rangle$. This syntactic check is important for the termination of the abduction. The unroll unfolds all shape predicates once in σ, normalises the result to a disjunctive form ($\bigvee_{i=1}^{n}\sigma_i$), and returns the result as a set of formulae ($\{\sigma_1, ..., \sigma_n\}$). The *XPure* is a strengthened version of that in [11], as it also keeps the pure part of σ' in the result.

In the third rule, neither side entails the other, and the second rule does not apply, for example $\exists\mathtt{p},\mathtt{u},\mathtt{v}\cdot\mathtt{x}{::}\mathtt{node}\langle\mathtt{u},\mathtt{p}\rangle * \mathtt{p}{::}\mathtt{node}\langle\mathtt{v},\mathtt{null}\rangle$ as LHS (σ) and $\exists\mathtt{S}\cdot\mathtt{x}{::}\mathtt{sllB}\langle\mathtt{S}\rangle$ as RHS (σ_1). In this case the antecedent cannot be unfolded as they are already data nodes. As the rule suggests, it reverses two sides of the entailment and applies the second rule to uncover the constraints σ_1' and σ'. Then it checks that the LHS (σ), with σ' added, does imply the RHS (σ_1) before it returns σ'. For the example above, the abduction returns $\mathtt{u}{\leq}\mathtt{v}$ which is essential for the two nodes to form a sorted list (σ_1).

When an abduction is conducted, the first three rules should be attempted first; if they do not succeed in finding a solution, the last rule is invoked to simply add the consequent to the antecedent, provided that they are consistent. It is effective for situations like $\mathtt{x}{::}\mathtt{node}\langle\_,\_\rangle \nvdash \mathtt{y}{::}\mathtt{node}\langle\_,\_\rangle$, where we should add $\mathtt{y}{::}\mathtt{node}\langle\_,\_\rangle$ to the LHS directly (as the other three rules do not apply here).

One observation on abduction is that there can be many solutions of the anti-frame σ' for the entailment $\sigma_1 * \sigma' \vdash \sigma_2 * \mathtt{true}$ to succeed. For instance, $\mathtt{false}$ is always a solution but should be avoided where possible. For all possible solutions to an abduction, we can compare their "quality" with a partial order $\preceq$ over SH defined by the entailment relationship ($\vdash$):

$$\sigma_1 \preceq \sigma_2 =_{df} \sigma_2 \vdash \sigma_1 * \mathtt{true}$$

and the smaller (weaker) one in two abduction solutions is regarded as better. We prefer to find solutions that are (potentially locally) minimal with respect to $\preceq$ and consistent. However, such solutions are generally not easy to compute and could incur excess cost (with additional disjunction in the analysis). Therefore, our abductive inference is designed more from a practical perspective to discover anti-frames that should be suitable as specifications for unknown procedures, and the partial order $\preceq$ is more a guidance of the decision choices of our abduction implementation, rather than a guarantee to find the theoretically best solution.

5 Verification

This section presents our algorithms to verify programs with unknown calls.

1. Main verification algorithm. Our main verification algorithm is given in Figure 5. It verifies an unknown block v (the third parameter) against given specifications $mspec_v$ (the second parameter). The first parameter includes the specifications of already available procedures which might be invoked as well as the unknown ones in the program to be verified. Upon successful verification, this algorithm returns specifications that should be met by the unknown procedures in v. If the verification fails, it suggests that the current program cannot meet one or more given specifications due to a potential program bug. The specifications for unknown procedures will be expressed in terms of special variables $\boldsymbol{a}, \boldsymbol{b}$, etc. as in the earlier example.

```
Algorithm Verify($\mathcal{T}$, $mspec_v$, $v$)
 1 Denote $v$ as $\{ e_1; u; e_2 \}$ ;  $mspec_u := \emptyset$
 2 $(\boldsymbol{x_0}, \boldsymbol{y_0}) :=$ prog_var($v$) ;  $(\boldsymbol{x}, \boldsymbol{y}) :=$ prog_var($u$)
 3 foreach ($requires\ \Phi_{pr}\ ensures\ \Phi_{po}$) $\in mspec_v$ do
 4    $\mathsf{S}_0 := [\![e_1]\!]_{\mathcal{T}}\{ \Phi_{pr} \wedge \boldsymbol{y_0'}{=}\boldsymbol{y_0} \}$
 5    if false $\in \mathsf{S}_0$ then return fail endif
 6    foreach $\sigma \in \mathsf{S}_0$ do
 7      $\Phi^u_{pr} :=$ Local($\sigma, \{\boldsymbol{x}, \boldsymbol{y}\}$)
 8      $\boldsymbol{z} :=$ fv($\Phi^u_{pr}$) $\setminus \{\boldsymbol{x}, \boldsymbol{y}\}$
 9      $\mathsf{S} := [\![e_2]\!]^{\mathsf{A}}_{\mathcal{T}}\{([\boldsymbol{b}/\boldsymbol{y}]$ Frame($\sigma, \{\boldsymbol{x}, \boldsymbol{y}\}$) $\wedge\, \boldsymbol{x}{=}\boldsymbol{a} \,\wedge$
              $\boldsymbol{y}{=}\boldsymbol{b} \wedge \boldsymbol{z}{=}\boldsymbol{c}$, emp $\wedge\, \boldsymbol{x}{=}\boldsymbol{a} \wedge \boldsymbol{y}{=}\boldsymbol{b} \wedge \boldsymbol{z}{=}\boldsymbol{c})\}$
10      $\mathsf{S}' := \{\ (\sigma, \sigma') \mid (\sigma, \sigma'){\in}\mathsf{S} \wedge \sigma \vdash \Phi_{po} * \mathtt{true}\ \} \ \cup$
            $\{\ (\sigma * \sigma'', \sigma' * \sigma'') \mid (\sigma, \sigma'){\in}\mathsf{S}\ \wedge$
              $\sigma \nvdash \Phi_{po}{*}\mathtt{true} \wedge \sigma{*}[\sigma''] \rhd \Phi_{po}{*}\mathtt{true}\ \}$
11      if $\exists(\sigma, \sigma'){\in}\mathsf{S}'$ . fv($\sigma'$) $\nsubseteq$ ReachVar($\sigma, \{\boldsymbol{a}, \boldsymbol{b}\}$)
          then return (fail, $\sigma'$) endif
12      foreach $(\sigma, \sigma') \in \mathsf{S}'$ do
13        $\Phi^u_{pr} := [\boldsymbol{a}/\boldsymbol{x}, \boldsymbol{b}/\boldsymbol{y}, \boldsymbol{c}/\boldsymbol{z}]\, \Phi^u_{pr}$
14        $\Phi^u_{po} :=$ sub_alias($\sigma', \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\}$)
15        $g :=$ (fv($\Phi^u_{pr}$) $\cap$ fv($\Phi^u_{po}$)) $\cup \{\boldsymbol{a}, \boldsymbol{b}\}$
16        $mspec_u := mspec_u \cup \{(requires\ \exists(\mathsf{fv}(\Phi^u_{pr})\backslash g) \cdot \Phi^u_{pr}$
                                     $ensures\ \Phi^u_{po})\}$
17      end foreach
18    end foreach
19 end foreach
20 $\mathcal{T}_u :=$ CaseAnalysis($\mathcal{T}$, $mspec_u$, $u$)
21 return $\mathcal{T} \uplus \mathcal{T}_u$
end Algorithm
```

Fig. 5. The main verification algorithm.

The algorithm initialises in the first two lines. It distinguishes the body of the unknown block v (as an unknown expression u in between two normal expressions e_1 and e_2), sets up the set to store discovered specifications (line 1), and finds the program variables that are potentially accessed by v and u, respectively (prog_var in line 2). Note that $\boldsymbol{x_0}$ and $\boldsymbol{x}$ are the variables read by v and u, and $\boldsymbol{y_0}$ and $\boldsymbol{y}$ are those mutated. For example, if v contains an assignment `y = x` then `x` will be in $\boldsymbol{x_0}$ and `y` in $\boldsymbol{y_0}$.

After the initialisation, for each specification ($requires\ \Phi_{pr}\ ensures\ \Phi_{po}$) to verify against (line 3), the algorithm works in three steps. The first step is to compute the preconditions of u (lines 4–7). It first conducts a symbolic execution from Φ_{pr} over e_1 (the program segment before u) to obtain its post-states, from which the preconditions for u will be extracted (line 4). The symbolic execution is essentially a forward analysis whose details are presented later. If the post-states include `false`, then it means the given Φ_{pr} cannot guarantee e_1's memory safety, and thus fail is returned (line 5). Otherwise, each post-state of e_1 is processed by function Local as a candidate precondition for u (line 7). Intuitively, it extracts the part of each σ reachable from the variables that may be accessed by u, namely, $\boldsymbol{x}$ and $\boldsymbol{y}$. The function Local is defined as follows:

$$\begin{aligned}\mathsf{Local}(\exists \boldsymbol{z} \cdot \kappa \wedge \pi, \{\boldsymbol{x}\}) =_{df} & \exists \mathtt{fv}(\sigma) \cup \{\boldsymbol{z}\} \setminus \mathtt{ReachVar}(\kappa \wedge \pi, \{\boldsymbol{x}\}) \cdot \\ & \mathtt{ReachHeap}(\kappa \wedge \pi, \{\boldsymbol{x}\}) \wedge \pi\end{aligned}$$

where $\mathsf{fv}(\sigma)$ stands for all free (program and logical) variables occurring in σ, and $\mathtt{ReachVar}(\kappa \wedge \pi, \{\boldsymbol{x}\})$ is the minimal set of variables reachable from $\{\boldsymbol{x}\}$:

$$\begin{aligned}&\{\boldsymbol{x}\} \cup \{z_2 \mid \exists z_1, \pi_1 \cdot z_1{\in}\mathtt{ReachVar}(\kappa{\wedge}\pi, \{\boldsymbol{x}\}) \wedge \pi{=}(z_1{=}z_2 \wedge \pi_1)\} \cup \{z_2 \mid \\ &\ \exists z_1, \kappa_1 \cdot z_1{\in}\mathtt{ReachVar}(\kappa{\wedge}\pi, v) \wedge \kappa{=}(z_1{::}c\langle .., z_2, ..\rangle * \kappa_1)\} \subseteq \mathtt{ReachVar}(\kappa{\wedge}\pi, \{\boldsymbol{x}\})\end{aligned}$$

That is, it is composed of aliases of $\boldsymbol{x}$ as well as variables reachable from $\boldsymbol{x}$. And the formula $\mathtt{ReachHeap}(\kappa{\wedge}\pi, \{\boldsymbol{x}\})$ denotes the part of κ reachable from $\{\boldsymbol{x}\}$ and is formally defined as the $*$-conjunction of the following set of formulae:

$$\{\kappa_1 \mid \exists z_1, z_2, \kappa_2 \cdot z_1{\in}\mathtt{ReachVar}(\kappa{\wedge}\pi, \{\boldsymbol{x}\}) \wedge \kappa{=}\kappa_1{*}\kappa_2 \wedge \kappa_1{=}z_1{::}c\langle .., z_2, ..\rangle\}$$

The second step is to discover the postconditions for u (lines 9–11). This is mainly completed with another symbolic execution with abduction over e_2 (line 9), whose details are also introduced later. Here we denote u's post-state as $\mathtt{emp}$, since its knowledge is not available yet. Therefore, the initial state for the symbolic execution of e_2 is simply the frame part of state not touched by u. The function Frame is formally defined as

$$\mathsf{Frame}(\exists \boldsymbol{z} \cdot \kappa \wedge \pi, \{\boldsymbol{x}\}) =_{df} \exists \boldsymbol{z} \cdot \mathtt{UnreachHeap}(\kappa \wedge \pi, \{\boldsymbol{x}\}) \wedge \pi$$

where $\mathtt{UnreachHeap}(\exists \boldsymbol{z} \cdot \kappa \wedge \pi, \{\boldsymbol{x}\})$ is the formula consisting of all $*$-conjuncts from κ which are not in $\mathtt{ReachHeap}(\exists \boldsymbol{z} \cdot \kappa \wedge \pi, \{\boldsymbol{x}\})$.

The conjunctions $\boldsymbol{x}{=}\boldsymbol{a} \wedge \boldsymbol{y}{=}\boldsymbol{b} \wedge \boldsymbol{z}{=}\boldsymbol{c}$ in line 9 are to keep track of variable snapshot accessed by u using the special variables $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$. Then the symbolic execution returns a set S of pairs (σ, σ') where σ is a possible post-state of e_2 and σ' records the discovered effect of u. However, maybe u still has some effect that is only exposed in the expected postcondition Φ_{po} for the whole program; therefore we need to check whether or not σ can establish Φ_{po}. If not, another abduction $\sigma{*}[\sigma''] \triangleright \Phi_{po}$ is invoked to discover further effect σ'' which is then added into σ'.

There can still be some complication here. Note that the effect discovered during e_2's symbolic execution may not be attributed all over to u; it is also possible that there is a bug in the program, or the given specification is not sufficient. As a consequence of that, the result σ' returned by our abduction may contain more information than what can be expected from u, in which case we cannot simply regard the whole σ' as the postcondition of u. To detect such a situation, we introduce the check in line 11. It tests whether the whole abduction result is reachable from variables accessed by u. If not, then the unreachable part cannot be expected from u, which indicates a possible bug in the program or some inconsistency between the program and its specification. In such cases, the algorithm returns an additional formula that can be used by a further analysis to either identify the bug or strengthen the specification.

The third step (lines 12–17) is to form the derived specifications for u in terms of variables $\boldsymbol{a}, \boldsymbol{b}$ and g. Here g denotes logical variables not explicitly accessed by

u, but occurring in both pre- and postconditions (ghost variables). The formula $\mathsf{sub\_alias}(\sigma', \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\})$ is obtained from σ' by replacing all variables with their aliases in $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\}$. Finally, at line 20, the obtained specifications mspec_u for u are passed to the case analysis algorithm (given in Figure 6) to derive the specifications of unknown procedures invoked in u.

2. Case analysis algorithm. In order to discover specifications for unknown procedures invoked in u, the algorithm in Figure 6 conducts a case analysis according to the structure of u. In the first case (line 2), u is simply a single unknown call. In this situation, the algorithm returns all the pre-/postcondition pairs from mspec_u as the unknown procedure's specifications.

```
Algorithm CaseAnalysis($\mathcal{T}$, $mspec_u$, $u$)
 1 switch $u$
 2    case $unk(\boldsymbol{x};\boldsymbol{y})$
 3       return $\{\,(unk(\boldsymbol{x};\boldsymbol{y}), mspec_u)\}$
 4    case if $(x)$ $v_1$ else $v_2$
 5       $mspec_T := \{(requires\ \Phi_{pr}\wedge x\ ensures\ \Phi_{po})\ |$
                     $(requires\ \Phi_{pr}\ ensures\ \Phi_{po}) \in mspec_u\}$
 6       $mspec_F := \{(requires\ \Phi_{pr}\wedge\neg x\ ensures\ \Phi_{po})\ |$
                     $(requires\ \Phi_{pr}\ ensures\ \Phi_{po}) \in mspec_u\}$
 7       $R_1 := \mathsf{Verify}(\mathcal{T}, mspec_T, v_1)$
 8       $R_2 := \mathsf{Verify}(\mathcal{T}, mspec_F, v_2)$
 9       return $R_1 \uplus R_2$
10    case if $(x)$ $v$ else $e$
11       $mspec_T := \{(requires\ \Phi_{pr}\wedge x\ ensures\ \Phi_{po})\ |$
                     $(requires\ \Phi_{pr}\ ensures\ \Phi_{po}) \in mspec_u\}$
12       $R := \mathsf{Verify}(\mathcal{T}, mspec_T, v)$
13       if $\exists(requires\ \Phi_{pr}\ ensures\ \Phi_{po}) \in mspec_u, \sigma \in [\![e]\!]_{\mathcal{T}}\{\Phi_{pr} \wedge \neg x\} \cdot$
             $\sigma{=}\mathtt{false} \vee \sigma \nvdash \Phi_{po}{*}\mathtt{true}$ then return fail
14       else return $R$ endif
15    case if $(x)$ $e$ else $v$    (Similar to the previous case)
16    case while $x$ $\{v\}$ inv $\Delta$
17       return $\mathsf{Verify}(\mathcal{T}, requires\ \Delta\wedge x\ ensures\ \Delta, v)$
18    case $unk_0(\boldsymbol{x_0};\boldsymbol{y_0})$ $\{\ ;e_i; unk_i(\boldsymbol{x_i};\boldsymbol{y_i})\}_{i=1}^{n}$
19       return $\{\,(unk_i(\boldsymbol{x_i};\boldsymbol{y_i}), \mathsf{SeqUnkCalls}(\mathcal{T}, mspec_u, u))\}_{i=0}^{n}$
end Algorithm
```

Fig. 6. The case analysis algorithm.

In the second case (line 4), u is an `if`-conditional and both branches contain an unknown block. The algorithm uses the main algorithm to verify the two branches separately with preconditions $\Phi_{pr}\wedge x$ and $\Phi_{pr}\wedge\neg x$ respectively, where Φ_{pr} is one of the preconditions of the whole `if`. The results obtained from the two branches are then combined using the $\uplus$ operator:

$$R_1 \uplus R_2 =_{df} \{(\mathsf{f}, \mathsf{Refine}(mspec_{\mathsf{f}}^1 \cup mspec_{\mathsf{f}}^2)) \mid (\mathsf{f}, mspec_{\mathsf{f}}^1){\in}R_1 \wedge (\mathsf{f}, mspec_{\mathsf{f}}^2){\in}R_2\}$$

where Refine is used to eliminate any specification $(requires\ \Phi_{pr}\ ensures\ \Phi_{po})$ from a set if there exists a "stronger" one $(requires\ \Phi'_{pr}\ ensures\ \Phi'_{po})$ such that $\Phi'_{pr}{\preceq}\Phi_{pr}$ and $\Phi_{po}{\preceq}\Phi'_{po}$. It is defined as

```
$\mathsf{Refine}(\emptyset) =_{df} \emptyset$
$\mathsf{Refine}(\{(requires\ \Phi_{pr}\ ensures\ \Phi_{po})\} \cup \mathsf{Spec}) =_{df}$
    if $\exists(requires\ \Phi'_{pr}\ ensures\ \Phi'_{po}){\in}\mathsf{Spec} \cdot \Phi'_{pr}{\preceq}\Phi_{pr} \wedge \Phi_{po}{\preceq}\Phi'_{po}$
    then $\mathsf{Refine}(\mathsf{Spec})$ else $\{(requires\ \Phi_{pr}\ ensures\ \Phi_{po})\} \cup \mathsf{Refine}(\mathsf{Spec})$
```

and $\uplus$ is to refine the union of two specification sets.

The third and fourth cases (lines 10 and 15) are for `if`-conditionals which contain only one unknown block in one of the two branches. This is handled in a similar way as in the second case. The only difference is, for the branch without unknown blocks, we need to verify it with the underlying semantics (line 13).

The fifth case is the `while` loop. As we assume its invariant is already given for the verification, we simply verify its body with the main algorithm, regarding the invariant as both pre- and postconditions (line 17).

In the last case (line 21), where u consists of multiple unknown procedure calls in sequence, another algorithm $\mathsf{SeqUnkCalls}$ is invoked to deal with it. We informally introduce its idea here due to space limit; its algorithm and subsequent discussions about our solution can be found in the report [9].

Suppose we have $\{\Phi_{pr}\}$ $\{unk_0(\boldsymbol{x_0};\boldsymbol{y_0}); e; unk_1(\boldsymbol{x_1};\boldsymbol{y_1})\}$ $\{\Phi_{po}\}$ to be verified, where e is the only known code fragment within the block. Our current solution finds a common specification to capture both unknown procedures' behaviours.

The algorithm works in three steps. In the first step, it extracts the precondition for the first procedure, say Φ^u_{pr}, from the given precondition Φ_{pr} by extracting the part of heap that may be accessed by the call via $\boldsymbol{x_0}$ and $\boldsymbol{y_0}$, which is similar to the first step of the main algorithm Verify. Aiming at a general specification for both unknown calls, it then assumes that the second procedure has a similar precondition Φ^u_{pr}. In the second step, it symbolically executes the code fragment e with the help of the abductor, to discover a crude postcondition, say Φ^u, expected from the first unknown call. This is similar to the second step of the main algorithm Verify, except that the postcondition for e is now assumed to be Φ^u_{pr}. In the third step, the algorithm takes Φ^u (with appropriate variable substitutions) as the postcondition of the second unknown call, and checks whether or not the derived post (Φ^u) satisfies Φ_{po}. If not, it invokes another abduction to strengthen Φ^u to obtain the final postcondition Φ^u_{po} for the unknown procedures. Note that this strengthening does not affect soundness: the strengthened Φ^u_{po} can still be used as a general postcondition for both unknown procedures.

3. Abstract semantics. Our verification algorithms utilise two semantics: an underlying semantics and an abstract semantics with abduction. They are used to conduct the forward analysis over program body. The type of our underlying semantics is defined as

$$[\![e]\!]\ :\ \mathsf{AllSpec} \rightarrow \mathcal{P}_{\mathsf{SH}} \rightarrow \mathcal{P}_{\mathsf{SH}}$$

where $\mathsf{AllSpec}$ contains procedure specifications (extracted from the program *Prog*). For some expression e, given its precondition, the semantics will calculate the postcondition.

The abstract semantics with abduction is of the type:

$$[\![e]\!]^{\mathsf{A}}\ :\ \mathsf{AllSpec} \rightarrow \mathcal{P}(\mathsf{SH} \times \mathsf{SH}) \rightarrow \mathcal{P}(\mathsf{SH} \times \mathsf{SH})$$

It takes a piece of program and a specification table, to map a (disjunctive) set of pair of symbolic heaps to another such set (where the first in the pair is the current state and the second is the accumulated postcondition for unknown call).

Formal definition of both semantics can be found in the technical report [9].

4. Soundness and termination. For soundness of our verification, we have the following theorem:

Theorem 1 (Soundness). *Our analysis is sound due to the soundness of entailment checking, abduction and abstract semantics.*

The proof for entailment checking is by structural induction [11]. For abduction, as its result is always checked with entailment, its soundness follows that of entailment checking's. Finally, the soundness of abstract semantics is proven by induction over program constructors.

We have also confirmed that our verification terminates:

Theorem 2 (Termination). *Our verification will terminate in finite steps for finite input of programs and specifications.*

This is because our algorithms perform structural reasoning over finite input. More details of soundness and termination can be found in our report [9].

6 Experimental Results

We have implemented the verification algorithms and the abstract semantics with Objective Caml and evaluated them over some heap-manipulating programs. The results are in Tables 1 and 2. In each table, the first and second columns denote the programs used for evaluation and their time consumption, respectively. During the experiments, we manually hide some instructions in the original programs as calls to unknown procedures, whose specifications we try to discover during the verification process. Accordingly, the third column in the first table contain both the specifications of the programs to be verified (upper line), and the derived specifications for the unknown procedure (lower line). For the second table, as we used the same specification $\mathtt{x}{::}\mathtt{llB}\langle\mathtt{S}\rangle \ast\!\!\rightarrow \mathtt{res}{::}\mathtt{sllB}\langle\mathtt{S}\rangle$ to verify all the sorting algorithms, the third column (from the second line on) states the discovered specification for the unknown call only. Due to space limit, more experimental results are available in our report [9].

It can be seen that all programs are successfully verified, with some obligations on the unknown calls discovered. We note down two observations on the experimental results. The first is that the discovered specifications for the unknown procedures are usually more general than what we expect. Bear in mind that we have replaced some instructions from those programs with unknown calls. We have compared the inferred specifications for those unknown calls with the original instructions. The results show that the specifications derived by our algorithm not only fully capture the behaviours of those instructions, but also suggest other possible implementations. A case in point is list's `travrs`. Its "unknown call" was originally an assignment `x = x.next` which traverses the list towards its end by one node. We are able to infer that the unknown call may actually traverse the list for arbitrary number of nodes, provided it does not go

Prog.	Time	**Main spec.** ($\Phi_{pr} \ast\!\!\!\rightarrow \Phi_{po}$) and **Derived unknown spec.** ($\Phi^u_{pr} \ast\!\!\!\rightarrow \Phi^u_{po}$)
List processing programs		
create	0.405	$\mathtt{emp \wedge n\geq 0 \ast\!\!\!\rightarrow res::llB\langle S\rangle \wedge n=\lvert S\rvert \wedge \forall v\in S\cdot 1\leq v\leq n}$
		$\mathtt{emp \wedge a\geq 1 \ast\!\!\!\rightarrow res::node\langle c, b\rangle \wedge 1\leq c\leq n}$
	1.020	$\mathtt{emp \wedge n\geq 0 \ast\!\!\!\rightarrow res::sllB2\langle S\rangle \wedge n=\lvert S\rvert \wedge \forall v\in S\cdot 1\leq v\leq n}$
		$\mathtt{emp \wedge a\geq 1 \ast\!\!\!\rightarrow res::node\langle c, b\rangle \wedge a-1\leq c\leq a}$
sort_ insert	0.667	$\mathtt{x::ll\langle n\rangle \wedge n\geq 1 \ast\!\!\!\rightarrow x::ll\langle m\rangle \wedge m=n+1}$
		$\mathtt{a::node\langle b, c\rangle \ast c::ll\langle d\rangle \ast\!\!\!\rightarrow a::node\langle b, e\rangle \ast e::ll\langle d+1\rangle}$
	0.764	$\mathtt{x::sll\langle n,xs,xl\rangle \wedge v\geq xs \ast\!\!\!\rightarrow x::sll\langle n+1,mn,mx\rangle \wedge mn=xs \wedge mx=max(xl,v)}$
		$\mathtt{a::node\langle b,c\rangle \ast c::sll\langle d,g,h\rangle \wedge b\leq f\leq g \ast\!\!\!\rightarrow a::node\langle b,e\rangle \ast e::sll\langle d+1,f,h\rangle}$
delete	0.646	$\mathtt{x::llB\langle S\rangle \wedge \lvert S\rvert\geq 2 \ast\!\!\!\rightarrow x::llB\langle T\rangle \wedge \exists a\cdot S=T\sqcup\{a\}}$
		$\mathtt{a::node\langle b, c\rangle \ast c::node\langle d, e\rangle \ast e::llB\langle E\rangle \ast\!\!\!\rightarrow a::node\langle b, e\rangle \ast e::llB\langle E\rangle}$
	0.916	$\mathtt{x::sllB\langle S\rangle \wedge \lvert S\rvert\geq 2 \ast\!\!\!\rightarrow x::sllB\langle T\rangle \wedge \exists a\cdot S=T\sqcup\{a\}}$
		$\mathtt{a::node\langle b, c\rangle \ast c::node\langle d, e\rangle \ast e::sllB\langle E\rangle \wedge \forall f\in E\cdot b\leq d\leq f \ast\!\!\!\rightarrow}$ $\mathtt{a::node\langle b, e\rangle \ast e::sllB\langle E\rangle \wedge \forall f\in E\cdot b\leq f}$
travrs	0.272	$\mathtt{x::ll\langle m\rangle \wedge n\geq 0\wedge m\geq n \ast\!\!\!\rightarrow x::ls\langle p,k\rangle \ast res::ll\langle r\rangle \wedge p=res \wedge k=n \wedge m=n+r}$
		$\mathtt{a::ll\langle b\rangle \ast\!\!\!\rightarrow a::ls\langle c\rangle \ast res::ll\langle d\rangle \wedge b=c+d \wedge c\leq n}$
	2.322	$\mathtt{x::sllB\langle S\rangle \wedge n\geq 0\wedge\lvert S\rvert\geq n \ast\!\!\!\rightarrow}$ $\mathtt{x::slsB\langle p, T\rangle \ast res::sllB\langle S_2\rangle \wedge p=res \wedge}$ $\mathtt{\lvert T\rvert=n \wedge S=T\sqcup S_2 \wedge \forall u\in T,v\in S_2 \cdot u\leq v}$
		$\mathtt{a::sllB\langle A\rangle \ast\!\!\!\rightarrow}$ $\mathtt{a::slsB\langle A_1\rangle \ast res::sllB\langle R\rangle \wedge}$ $\mathtt{A=A_1\sqcup R \wedge \lvert A_1\rvert\leq n \wedge \forall b\in A_1,c\in R\cdot b\leq c}$
Binary tree, binary search tree, AVL tree and red-black tree processing programs		
height	0.821	$\mathtt{x::bt\langle S, h\rangle \ast\!\!\!\rightarrow x::bt\langle T, k\rangle \wedge res=h=k \wedge S=T}$
		$\mathtt{a::bt\langle A, b\rangle \wedge a\neq null \ast\!\!\!\rightarrow a::node2\langle c, d, e\rangle \ast d::bt\langle D, f\rangle \ast e::bt\langle E, g\rangle \wedge}$ $\mathtt{A=\{c\}\sqcup D\sqcup E \wedge b=max(f,g)+1 \wedge (res=d \vee res=e)}$
search	1.851	$\mathtt{x::bst\langle sm, lg\rangle \ast\!\!\!\rightarrow x::bst\langle mn, mx\rangle \wedge sm=mn \wedge lg=mx \wedge 0\leq res\leq 1}$
		$\mathtt{a::bst\langle b, c\rangle\wedge a\neq null \ast\!\!\!\rightarrow a::node2\langle d, e, f\rangle \ast e::bst\langle b, g\rangle \ast f::h\langle c\rangle\wedge g\leq d\leq h}$
avl_ins	5.202	$\mathtt{x::avl\langle S, h\rangle \ast\!\!\!\rightarrow res::avl\langle T, k\rangle \wedge T=S\sqcup\{v\} \wedge h\leq k\leq h+1}$
		$\mathtt{a::avl\langle A, b\rangle \ast\!\!\!\rightarrow a::avl\langle A, b\rangle \wedge res=b}$
rbt_ins	9.093	$\mathtt{x::rbt\langle S, cl, bh\rangle \ast\!\!\!\rightarrow res::rbt\langle T, cl_1, bh_1\rangle \wedge T=S\sqcup\{v\}}$
		$\mathtt{a::rbt\langle A, b, c\rangle \ast\!\!\!\rightarrow a::rbt\langle A, b, c\rangle \wedge res=b}$

Table 1. Selected experimental results (lists and trees).

beyond the list's tail or where the user has specified as input, which allows more implementations for the unknown procedure to be verified.

The second observation is that the precision of unknown calls' discovered specifications depends on its caller's given specification. As can be seen we have verified several list-processing programs where each one has various specifications. Within these programs we want to point out that the ones with specifications of both normal lists and sorted lists share the same code (but just with two different specifications). Such examples include `create`, `sort_insert`, `delete`, and so on. For `create` which creates a list containing numbers from 1 to `n` in descending order, we can see once incorporated with `llB` as specification predicates, the unknown call is expected to return a node whose value `c` is within 1 to `n`. Comparatively, when verified for sortedness, `c` is inferred to be between `a−1` and `a`, as for sortedness to hold. For `delete`'s sorted version, we also have the extra information that the list with one node removed is still a sorted list (with the multi-set value constraints), whose result is stronger than the normal list version.

Prog.	**Time**	**Main spec.** ($\Phi_{pr} \ast\!\!\!\rightarrow \Phi_{po}$) or **Derived unknown spec.** ($\Phi^{u}_{pr} \ast\!\!\!\rightarrow \Phi^{u}_{po}$)
Sorting (main)		$\mathtt{x}::\mathtt{llB}\langle\mathtt{S}\rangle \ast\!\!\!\rightarrow \mathtt{res}::\mathtt{sllB}\langle\mathtt{T}\rangle \wedge \mathtt{T}{=}\mathtt{S}$
`merge`	4.099	$\mathtt{a}::\mathtt{sllB}\langle\mathtt{A}\rangle \ast \mathtt{b}::\mathtt{sllB}\langle\mathtt{B}\rangle \ast\!\!\!\rightarrow \mathtt{res}::\mathtt{sllB}\langle\mathtt{R}\rangle \wedge \mathtt{R}{=}\mathtt{A}{\sqcup}\mathtt{B}$
`quick`	2.064	$\mathtt{a}::\mathtt{lbd}\langle\mathtt{A}\rangle \ast\!\!\!\rightarrow \mathtt{a}::\mathtt{lbd}\langle\mathtt{A}_1\rangle \ast \mathtt{res}::\mathtt{lbd}\langle\mathtt{R}\rangle \wedge \mathtt{A}{=}\mathtt{A}_1{\sqcup}\mathtt{R} \wedge \forall\mathtt{c}{\in}\mathtt{A}_1, \mathtt{d}{\in}\mathtt{R} \cdot \mathtt{c}{\leq}\mathtt{b}{\leq}\mathtt{d}$
`unknown`	1.824	$\mathtt{a}::\mathtt{llB}\langle\mathtt{A}\rangle{\wedge}\mathtt{a}{\neq}\mathtt{null} \ast\!\!\!\rightarrow \mathtt{res}::\mathtt{node}\langle\mathtt{c},\mathtt{b}\rangle{\ast}\mathtt{b}::\mathtt{llB}\langle\mathtt{B}\rangle{\wedge}\mathtt{A}{=}\{\mathtt{c}\}{\sqcup}\mathtt{B}{\wedge}\forall\mathtt{d}{\in}\mathtt{B}{\cdot}\mathtt{c}{\leq}\mathtt{d}$

Table 2. Selected experimental results (sorting).

7 Conclusion

It is a practical and challenging problem to verify the full functional correctness of heap-manipulating imperative programs with unknown procedure calls. Our proposed solution infers expected specifications for unknown procedures from their calling contexts. The program is verified correct on condition that the invoked unknown procedures meet the inferred specifications. We employ a forward program analysis over a combined domain and invent a novel abduction for it to synthesise the specifications of the unknown procedure. As a proof of concept, we have also implemented a prototype system to test the viability of the proposed approach. Our main future work is to explore more general solution for unknown calls in sequence to achieve more reasonable specifications for them.

Acknowledgement. This work was supported in part by the EPSRC projects EP/G042322/1 and EP/E021948/1.

References

1. G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *POPL*, 2002.
2. B. Beizer and J. Wiley. Black-box testing: techniques for functional testing of software and systems. *IEEE Software*, 13(5), September 1996.
3. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *36th POPL*, January 2009.
4. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, 1994.
5. R. Giacobazzi. Abductive analysis of modular logic programs. In *ILPS*, 1994.
6. D. Gopan and T. Reps. Low-level library analysis and summarization. In *19th CAV*, 2007.
7. W. Kozaczynski and G. Booch. Component-based software engineering. *IEEE Software*, 15(5):34–36, September 1998.
8. C. Luo, F. Craciun, S. Qin, G. He, and W.-N. Chin. Verifying pointer safety for programs with unknown calls. *Journal of Symbolic Computation*, To appear.
9. S. Qin, C. Luo, G. He, F. Craciun, and W.-N. Chin. Verifying heap-manipulating programs with unknown calls. Research report, Teesside University, 2010. `http://www.scm.tees.ac.uk/s.qin/papers/unknown.pdf`.
10. S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Thor: A tool for reasoning about shape and arithmetic. In *CAV*, 2008.
11. H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *8th VMCAI*, 2007.
12. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *31st POPL*, January 2004.
13. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17th LICS*, 2002.
14. R. Sessions. *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley & Sons, Inc., New York, NY, USA, 1998.
15. C. Szyperski. Component technology: what, where, and how? In *ICSE*, 2003.
16. J. Woodcock. Verified software grand challenge. In *14th FM*, 2006.