Multiple Pre/Post Specifications for Heap-Manipulating Methods

Wei-Ngan Chin[1,2] Cristina David[1] Huu Hai Nguyen[2] Shengchao Qin[3]
[1] Department of Computer Science, National University of Singapore
[2] Computer Science Programme, Singapore-MIT Alliance
[3] Department of Computer Science, Durham University
{chinwn,davidcri,nguyenh2}@comp.nus.edu.sg shengchao.qin@durham.ac.uk

Abstract

Automated verification plays an important role for high assurance software. This approach typically uses a pair of pre/post conditions as a formal (but possibly partial) specification of each method before it is systematically verified. In this paper, we advocate for multiple pairs of pre/post conditions to be associated with each method which provides a way for such specification to be used in a wider range of contexts. Multiple pre/post specifications are important for heap-manipulating programs where they can be precisely expressed using separation logic. This work highlights the importance of multiple pre/post specifications, and a methodology to capture them via set of states during proof search. Their use can increase the success rate for automated verification.

1 Introduction

In recent years, separation logic formalism has been successfully applied to analysing and verifying heap-manipulating programs. This formalism supports succinct description of shapely data structures, such as near-balanced AVL-trees [12], and has been used to analyse and verify various program properties, including program termination [3]. One important feature of separation logic is its support for accurate references into memory states which can be captured with the help of inductive shape predicate and separating conjunction. For example, a list segment of length $\mathtt{n}$ can be specified by the following predicate definition:

$$\begin{array}{l}\mathtt{lseg(root,p,n)} \equiv \mathtt{root=p} \wedge \mathtt{n=0} \\ \quad \vee\ \exists \mathtt{q} \cdot \mathtt{root} {\mapsto} \mathtt{node}\langle \_, \mathtt{q}\rangle * \mathtt{lseg(q,p,n-1)} \\ \quad \mathbf{inv}\ \mathtt{n} \geq 0\end{array}$$

where $\mathtt{node}$ is an object type declared as

$$\mathtt{data\ node}\ \{\ \mathtt{int\ val;\ node\ next}\ \}$$

In separation logic notation, $\mathtt{x}{\mapsto}\mathtt{node}\langle \mathtt{a},\mathtt{b}\rangle$ captures a distinct memory cell referenced from $\mathtt{x}$, while separating conjunction $\Delta_1 * \Delta_2$ captures two *disjoint* heaps described by Δ_1 and Δ_2, respectively. In contrast, $\Delta_3 \wedge \Delta_4$ indicates two *overlapping* heaps described by Δ_3 and Δ_4, respectively. The formula $\mathtt{n} \geq 0$ after the **inv** keyword captures a heap-independent invariant for the predicate. This invariant is an approximation of the predicate, in the sense that whenever the predicate holds, the invariant also holds. By default, each shape predicate is expected to have a special (first) parameter, named $\mathtt{root}$, that can transitively reach all its memory cells. This "root" parameter will be instantiated with an actual argument for each instance of its predicate that appears in formulas. Furthermore, we shall use a uniform notation where object $\mathtt{x}{\mapsto}\mathtt{node}\langle \mathtt{a},..\rangle$ is written as $\mathtt{x}{::}\mathtt{node}\langle \mathtt{a},..\rangle$, and predicate $\mathtt{pred(x,a,..)}$ is written as $\mathtt{x}{::}\mathtt{pred}\langle \mathtt{a},..\rangle$. We refer to them collectively as *heap nodes*. With this new notation, we can omit the $\mathtt{root}$ parameter in the head of the predicate, and define a non-empty circular list, as follows:

$$\begin{array}{ll}\mathtt{clist}\langle \mathtt{n}\rangle \equiv & \exists \mathtt{p} \cdot \mathtt{root}{::}\mathtt{node}\langle \_, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{lseg}\langle \mathtt{root}, \mathtt{n}{-}1\rangle \\ & \mathbf{inv}\ \mathtt{n} \geq 1\end{array}$$

Shape predicates can also be used to describe more complex data structures with stronger properties. An example is the non-empty sorted list:

$$\begin{array}{l}\mathtt{sortl}\langle \mathtt{n,s,b}\rangle \equiv \mathtt{root}{::}\mathtt{node}\langle \mathtt{s},\mathtt{null}\rangle \wedge \mathtt{s=b} \wedge \mathtt{n=1} \\ \quad \vee\ \exists \mathtt{p,t} \cdot \mathtt{root}{::}\mathtt{node}\langle \mathtt{s},\mathtt{p}\rangle * \mathtt{p}{::}\mathtt{sortl}\langle \mathtt{n}{-}1,\mathtt{t},\mathtt{b}\rangle \wedge \mathtt{s}{\leq}\mathtt{t} \\ \quad \mathbf{inv}\ \mathtt{n}>0 \wedge \mathtt{s} \leq \mathtt{b}\end{array}$$

Predicate $\mathtt{sortl}\langle \mathtt{n,s,b}\rangle$ ensures that all values in the list are sorted in ascending order, with $\mathtt{s}$ and $\mathtt{b}$ to capture its min and max values, respectively. The sortedness property is ensured by the presence of $\mathtt{s} \leq \mathtt{t}$

in the above predicate. The parameters $\mathtt{n, s, b}$ actually capture some derived properties of the heap data structure, as stated in the predicate definition. These parameters play a role similar to "model fields" in some specification languages, such as Spec# [11, 1] and JML [4]. As a shorthand, we may omit the existential quantifiers without ambiguity.

Though separation logic with inductive predicates can be highly expressive, current automated theorem provers hardly provide any support for this form of substructural logic. Several recent works, such as [2, 5], have attempted to address this shortcoming by building specialised solvers that work for a fixed set of predicates (e.g. `lseg` without the size parameter). Our recent work [12] has lifted one crucial limitation by supporting the automated reasoning of user-defined predicates. To this end, we have designed a general prover that uses unfold/fold reasoning on predicates to support entailment between two heap states, and also for computing its residual outcome. This is essentially a prover with "frame inferring" capability. Given two heap states Δ_a and Δ_c, our prover can check for an entailment of the form: $\Delta_a \vdash \Delta_c * \Delta_r$, where Δ_r is a residual heap from Δ_a after fully accounting for the memory heap state of Δ_c. The residual is a consequence of the frame rule from separation logic that can be systematically determined by an incremental matching algorithm in our prover. Our entailment also handles disjunctive formulae and existential quantifiers. While our prover is sound, terminating and automated, it is incomplete.

In this paper, we propose one approach to partially overcome this incompleteness shortcoming, by providing a mechanism for specifying and handling multiple pre/post conditions in separation logic. Traditionally, each method is given a single pair of pre/post conditions which describes the expected pre-state prior to a method call and its subsequent post-state. Furthermore, even when multiple pre/post conditions are allowed in some specification languages, such as JML [4], the standard technique for handling them is to recombine into a single pre/post condition. As an example, consider a method with two (pairs of) pre/post conditions, $(\mathtt{pre}_1, \mathtt{post}_1)$ and $(\mathtt{pre}_2, \mathtt{post}_2)$, as shown below (in JML syntax):

$$\begin{array}{l}\mathtt{requires}\ \mathtt{pre}_1\ \mathtt{ensures}\ \mathtt{post}_1\\ \quad\textbf{also}\\ \mathtt{requires}\ \mathtt{pre}_2\ \mathtt{ensures}\ \mathtt{post}_2\end{array}$$

The proposed solution in JML [8] is to transform the two pre/post conditions into the following integrated pre/post condition:

$$\begin{array}{l}\mathtt{requires}\ \mathtt{pre}_1 \vee \mathtt{pre}_2\\ \mathtt{ensures}\ \mathtt{old}(\mathtt{pre}_1) \Longrightarrow \mathtt{post}_1 \wedge \mathtt{old}(\mathtt{pre}_2) \Longrightarrow \mathtt{post}_2\end{array}$$

While this re-combination may work with traditional Hoare logic that is based on pure formulae, it cannot be directly expressed for an arbitrary heap state Δ of separation logic, since $\mathtt{old}(\Delta)$ is not always determinable. As an example of multiple pre/post conditions in separation logic formula, consider the `append` method for joining two lists together:

```
void append(node x, node y)
{if x.next≠null then { append(x.next, y) }
 else {x.next:=y } }
```

A simple specification for this method is for joining two disjoint lists into a single longer list, as illustrated by the first pre/post condition below:

$$\begin{array}{l}\mathtt{requires}\ \mathtt{x{::}lseg}\langle \mathtt{null}, \mathtt{n}_1\rangle * \mathtt{y{::}lseg}\langle \mathtt{null}, \mathtt{n}_2\rangle \wedge \mathtt{n}_1 > 0\\ \mathtt{ensures}\ \mathtt{x{::}lseg}\langle \mathtt{null}, \mathtt{n}_1 + \mathtt{n}_2\rangle\}\end{array}$$

The $\mathtt{n}_1 > 0$ constraint ensures that the first input is non-empty. As a result, the pointer access operation $\mathtt{x.next} \neq \mathtt{null}$ can be proven to be safe. Another radically different view of this method is for the purpose of joining two sorted lists, whereby the largest value of the first sorted list is less than the smallest value of the second sorted list. This view is captured by a second pre/post condition below:

$$\begin{array}{l}\mathtt{requires}\ \mathtt{x{::}sortl}\langle \mathtt{n}_1, \mathtt{a}, \mathtt{b}\rangle * \mathtt{y{::}sortl}\langle \mathtt{n}_2, \mathtt{c}, \mathtt{d}\rangle \wedge \mathtt{b} \leq \mathtt{c}\\ \mathtt{ensures}\ \mathtt{x{::}sortl}\langle \mathtt{n}_1 + \mathtt{n}_2, \mathtt{a}, \mathtt{d}\rangle\}\end{array}$$

With a more specialised pre-state, our specification is able to conclude that the resulting list is a longer sorted list. Furthermore, the `sortl` predicate is non-empty which can prove that dereferencing by $\mathtt{x.next} \neq \mathtt{null}$ is safe. Such a specification captures a different view for the same `append` method, which would be needed towards the verification of the sortedness property for the quicksort algorithm. Our thesis is that it is often futile to combine these widely different pre/post conditions into a single pre/post condition, as the translations for $\mathtt{old}(\mathtt{x{::}lseg}\langle \mathtt{null}, \mathtt{n}_1\rangle * \mathtt{y{::}lseg}\langle \mathtt{null}, \mathtt{n}_2\rangle \wedge \mathtt{n}_1 > 0)$ and $\mathtt{old}(\mathtt{x{::}sortl}\langle \mathtt{n}_1, \mathtt{a}, \mathtt{b}\rangle * \mathtt{y{::}sortl}\langle \mathtt{n}_2, \mathtt{c}, \mathtt{d}\rangle \wedge \mathtt{b} \leq \mathtt{c})$ into heap (or pure) formulae cannot be systematically determined. For example, in JML, $\mathtt{old}(\mathtt{e})$ is guaranteed to be safe to use if `e` denotes a primitive value, or an immutable object [9] (sec 11.4.2). Otherwise, some ambiguity in specification is possible.

In this paper, we propose a new mechanism to support the direct handling of multiple pre/post conditions. Our mechanism is based on a set of states $\{\Delta_1, .., \Delta_n\}$ which represent abstract states that may arise during proof search used by automated verification. As a first step, we propose to generalise the entailment procedure into a non-deterministic version that explicitly returns a set of residual states, namely: $\Delta_a \vdash \Delta_c * \{\Delta_1, .., \Delta_n\}$, such that for all $1 \leq i \leq n$, $\Delta_a \vdash \Delta_c * \Delta_i$ holds. In the event of an entailment failure, the set of residual states is empty. Note that "states" here are the residual formulas, representing the abstract states of the program.

The set of residual states is meant to contain syntactically different states. This does not imply that all the states in the set are semantically different. Our approach may indeed cause some redundancy in the search for proofs, but it does not affect soundness of the entailment prover. Explicit set of states allows easier integration between the prover and other components of our verification system. In particular, its use is critical for handling multiple pre/post specifications which can now be viewed as part of proof search.

More specifically, this paper makes the following contributions:

- **Multiple pre/post:** This mechanism allows users to specify more properties of a procedure. Our system automatically verifies that each user-provided specification is correct, prior to using them for systematic proof search.
- **Set of states:** Provers for separation logic typically need to derive residues (or frames), which have to be communicated to program verifiers. Our direct support for non-deterministic proof search via set of states notation makes this task both explicit and exhaustive.
- **Implementation:** We have implemented a verification system that supports multiple pre/post specifications. Initial experiments show that the cost of conducting proof search via set of states remains low, but is crucial to verify some examples. Moreover, the smaller heap formulae from using multiple pre/post specifications may sometimes result in significant improvement to the performance of automated verification.

2 Separation Logic Formalism Used

The syntax for separation logic formulae that we will use is given in Figure 1. Each shape predicate *spred* has a body Φ, which can be recursive as it can mention the predicate being defined. Each predicate can also be equipped with a heap-independent invariant π_0 that is user-supplied but machine-checked for its validity. The invariant is marked with a prior keyword **inv**.

The separation constraints we use are in a disjunctive normal form Φ. Each disjunct consists of a $*$-separated heap constraint κ, referred to as *heap part*, and a heap-independent formula π, referred to as *pure part*. The pure part does not contain any heap nodes and is restricted to pointer equality/disequality (to facilitate precise aliasing and non-aliasing), Presburger arithmetic and set/bag constraints that can capture a collection of reachable values/addresses. Furthermore, Δ denotes a composite formula that could always be normalised into the Φ form (see Fig. 5 in Sec 4.1).

$$
\begin{array}{lll}
\mathit{spred} & ::= & c\langle v^* \rangle \equiv \Phi\ \mathtt{inv}\ \ \pi_0 \\
\Phi & ::= & \bigvee (\exists v^* \cdot \kappa \wedge \pi)^* \qquad \pi ::= \gamma \wedge \phi \\
\gamma & ::= & v_1{=}v_2 \mid v{=}\mathtt{null} \mid v_1{\neq}v_2 \mid v{\neq}\mathtt{null} \mid \gamma_1 \wedge \gamma_2 \\
\kappa & ::= & \mathtt{emp} \mid v{::}c\langle v^* \rangle \mid \kappa_1 * \kappa_2 \\
\Delta & ::= & \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists \mathtt{v} \cdot \Delta \\
\phi & ::= & \varphi \mid b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg \phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi \\
b & ::= & \mathtt{true} \mid \mathtt{false} \mid v \mid b_1 = b_2 \\
a & ::= & s_1{=}s_2 \mid s_1{\leq}s_2 \\
s & ::= & k^{\mathtt{int}} \mid v \mid k^{\mathtt{int}} {\times} s \mid s_1{+}s_2 \\
 & & \mid -s \mid \mathit{max}(s_1,s_2) \mid \mathit{min}(s_1,s_2) \mid |B| \\
\varphi & ::= & v{\in}B \mid B_1{=}B_2 \mid B_1{\sqsubset}B_2 \mid \forall v{\in}B \cdot \phi \mid \exists v{\in}B \cdot \phi \\
B & ::= & B_1{\sqcup}B_2 \mid B_1{\sqcap}B_2 \mid B_1{-}B_2 \mid \{\} \mid \{v\}
\end{array}
$$

Figure 1. Syntax for Formulas

Separation constraints are used in pre/post conditions and shape definitions. We will also use another special variable "`res`" to denote the returned value of an expression (including the method body). In order to handle them correctly without running into unmatched residual heap nodes, we require all constraints used in specifications be well-formed.

Definition 2.1 (Accessible) *A variable in a specification is said to be* accessible *if it is a method parameter or a special variable, i.e.* `root` *or* `res`.

Definition 2.2 (Reachable) *Given a heap constraint κ such that $\kappa = \mathtt{p}{::}\mathtt{c}\langle \mathtt{v}^* \rangle{*}\kappa_1$, node $\mathtt{p}{::}\mathtt{c}\langle \mathtt{v}^* \rangle$ is* reachable *from a variable* `q` *iff they satisfy the following relation:*

$$
\begin{array}{l}
\mathtt{reach}(\kappa_1, \mathtt{q}, \mathtt{p}{::}\mathtt{c}\langle \mathtt{v}^* \rangle) =_{df} (\mathtt{p} = \mathtt{q}) \\
\vee(\kappa_1 = \mathtt{q}{::}\mathtt{c_q}\langle .., \mathtt{r}, .. \rangle * \kappa_2 \wedge \mathtt{reach}(\kappa_2, \mathtt{r}, \mathtt{p}{::}\mathtt{c}\langle \mathtt{v}^* \rangle))
\end{array}
$$

Definition 2.3 (Well-Formed Constraint) *A separation constraint Φ is well-formed if (i) all objects and predicate instances in the constraint are reachable from accessible variables, (ii) Φ is in a disjunctive normal form $\bigvee(\exists v^* \cdot \kappa \wedge \pi)^*$ where κ denotes heap constraints, π pointer, arithmetic and set/bag constraints.*

The primary significance of the *well-formed* condition is that all heap nodes of a heap constraint are reachable from accessible variables. This allows the entailment proving procedure to correctly match nodes from the consequent with nodes from the antecedent of an entailment relation. Furthermore, arbitrary recursive shape relation can lead to non-termination in unfold/fold reasoning. We avoid this pitfall by using only *well-founded* shape predicates:

Definition 2.4 (Well-Founded Predicate) *A shape predicate is said to be* well-founded *if its body satisfies three conditions, namely: (i) it is a well-formed constraint, (ii) the parameter* `root` *may not be bound to a predicate instance, (iii) there is at most one object in each disjunct.*

3 Multiple Pre/Post Specifications

Our approach currently expects pre/post conditions to be specified for each method. With the rich variety of shapes that can be specified, there are often multiple ways of viewing a method's intended behaviour. In this section, we explore the contexts in which multiple pre/post specifications are useful for heap-manipulating methods. For simplicity, we shall illustrate via our running `append` method, that is reproduced below.

$$\begin{array}{l}\texttt{void append(node x, node y)}\\ \{\texttt{if x.next}\neq\texttt{null then }\{\texttt{ append(x.next, y) }\}\\ \;\texttt{else }\{\texttt{x.next:=y }\}\;\}\end{array}$$

Firstly, multiple pre/post specifications are helpful for capturing *strong aliasing* properties in separation logic. Our previous specification was for two disjoint input lists. We can also provide a pre/post specification in which `x` and `y` parameters are aliased, as follows:

$$\begin{array}{l}\texttt{requires x::lseg}\langle\texttt{null},\texttt{n}\rangle\wedge\texttt{n}>0\wedge\texttt{x}=\texttt{y}\\ \texttt{ensures x::clist}\langle\texttt{n}\rangle\end{array}$$

The aliasing by `x=y` causes the `append` function to return a circular list instead! To successfully verify the recursive call of `append`, we also require another pre/post specification that returns a list segment, as follows:

$$\begin{array}{l}\texttt{requires x::lseg}\langle\texttt{null},\texttt{n}\rangle\wedge\texttt{n}>0\\ \texttt{ensures x::lseg}\langle\texttt{n},\texttt{y}\rangle\end{array}$$

Secondly, we may require pre/post specification to handle *missing cases* that are absent in our predicates. Our earlier example made use of a non-empty `sortl` predicate which was essential for the first `x` parameter, but not the the second `y` parameter. To allow the `y` parameter to be possibly empty, we shall provide an extra pre/post specification, as follows:

$$\begin{array}{l}\texttt{requires x::sortl}\langle\texttt{n}_1,\texttt{i}_1,\texttt{j}_1\rangle\wedge\texttt{y}=\texttt{null}\\ \texttt{ensures x::sortl}\langle\texttt{n}_1,\texttt{i}_1,\texttt{j}_1\rangle\end{array}$$

Thirdly, we may use smaller predicates to specify additional properties of our method. This is helpful towards better functional correctness and also *scalable verification*. For example, we can further introduce another predicate that captures the reachability of a linked-list as a bag of its values, as follows:

$$\begin{array}{l}\texttt{llR}\langle\texttt{B}\rangle\equiv(\texttt{root}=\texttt{null}\wedge\texttt{B}=\{\})\vee\\ (\exists\texttt{v},\texttt{B}_1\cdot\texttt{root::node}\langle\texttt{v},\texttt{r}\rangle*\texttt{r::llR}\langle\texttt{B}_1\rangle\wedge\texttt{B}=\{\texttt{v}\}\cup\texttt{B}_1)\end{array}$$

This new predicate captures the reachable values in a linked-list but requires a bag constraint solver. It is useful as it can help specify the reachability property by the `append` method, as captured by the following extra pre/post specification:

$$\begin{array}{l}\texttt{requires x::llR}\langle\texttt{B}_1\rangle*\texttt{y::llR}\langle\texttt{B}_2\rangle\wedge\texttt{x}\neq\texttt{null}\\ \texttt{ensures x::llR}\langle\texttt{B}_1\cup\texttt{B}_2\rangle\}\end{array}$$

Multiple pre/post specifications allow us to freely use smaller predicates whereby different properties can be separately proven. As illustrated later by our experiments, this can help support scalable verification.

As stated before, the concept of *multiple pre/post* specification is not new. Conceptually, it is also related to the notion of intersection types [14], especially when it is compared with a dependent type system with effects on heap states. While the concept of multiple pre/post is not new, its use in separation logic is novel and provides a fresh challenge for automated verification.

In this paper, we propose the concept of *set of states* to handle multiple pre/post specification, and as a means towards systematic proof search. Our solution avoids the need to transform multiple pre/post specification into a single pre/post specification. We chose this path for the following reasons: (i) it is more concise, (ii) it allows pre/post conditions to be decomposed in a modular fashion, (iii) it integrates well with set of states, and (iv) its use results in smaller heap states. As smaller heap states are likely to be faster to verify, we expect multiple pre/post to give better support for scalable verification, whereby smaller predicates are strongly encouraged. This facilitates modular reuse of the verification processes, in addition to support for systematic proof search. The next two sections highlight the techniques we have formulated to support multiple pre/post specification in separation logic, and some experiments we conducted to validate its utility.

4 Our Approach

In our approach to verification, we expect users to supply two things that can assist in automatic verification of their code, namely shape predicate definitions and multiple pre/post specifications for each method. Our system has two major components, namely (i) Hoare-style verifier and (ii) entailment prover for separation logic, as shown in Figure 2.

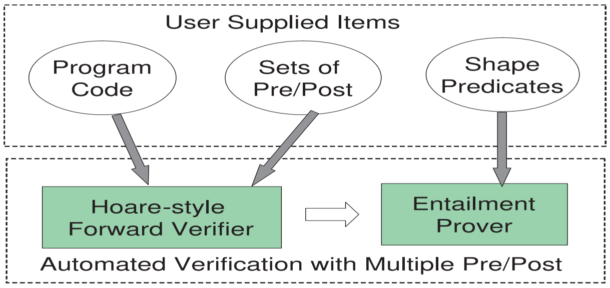

Figure 2. Our Approach to Verification

4.1 Forward Verifier

Our Hoare-style verifier is defined for an imperative object-based language with syntax in Figure 3. A program P consists of declarations *tdecl* and methods *meth*. Declarations can be shape predicates *spred* or object types *objt*. Each method is decorated with the specification $\{\Phi^i_{pr} \ast\!\!\!\longrightarrow \Phi^i_{po}\}^p_{i=1}$ that is made up of a collection of pre- and post-condition pairs. This more precise syntax for multiple pre/post pairs shall be used in our formalization. The intended meaning is that whenever the method is called in a program state satisfying precondition Φ^i_{pr} and if the method terminates, the resulting state will satisfy the corresponding postcondition Φ^i_{po}. We handle `while` loop in a similar way. Other constructs are standard.

Hoare-style code verifier is based on forward rules of the form $\{\Delta_1\}\mathtt{code}\{\Delta_2\}$, where Δ_1 is a prestate in separation formula that is given, while Δ_2 is a poststate that shall be computed by forward reasoning. To capture proof search, we generalize the forward rule to the form $\{\Delta\}\mathtt{code}\{\mathtt{S}\}$ where `S` is a set of heap states, discovered by a search-based verification process. When `S` is empty, the forward verification is said to have failed for Δ as prestate. For convenience, we also provide lifted variant of the forward verifier to take a set of prestates. Verification in such a case succeeds if any of the prestates gives rise to a successful verification, that is if at least one of the S_i is non-empty. This rule is useful when the forward verifier has processed at least one subexpression, potentially giving rise to a set of residual states.

$$\frac{\forall i \in 1..n \cdot \{\Delta_i\}\ \mathrm{code}\ \{S_i\}}{\vdash \{\{\Delta_1, .., \Delta_n\}\}\ \mathrm{code}\ \{\bigcup^n_{i=1} S_i\}}$$

Verification of a method starts with each precondition, and proves that the corresponding postcondition is guaranteed at the end of the method. The verification is formalized in the following rule:

$$\frac{\begin{array}{c}[\mathbf{FV{-}METH}] \\ V{=}\{v_1..v_n\} \quad W{=}prime(V) \\ \forall i = 1, .., p \ \cdot \ (\ \vdash \{\Phi^i_{pr} \wedge nochange(V)\}\, e\, \{S^i_1\} \\ (\exists W{\cdot}S^i_1) \vdash \Phi^i_{po} * S^i_2 \qquad S^i_2 {\neq} \{\})\end{array}}{\vdash t_0\ mn(t_1\, v_1, .., t_n\, v_n)\ \mathtt{where}\ \{\Phi^i_{pr} \ast\!\!\!\longrightarrow \Phi^i_{po}\}^p_{i=1}\ \{e\}}$$

The function *prime(V)* returns $\{v' \mid v \in V\}$. The predicate *nochange(V)* returns $\bigwedge_{v \in V}(v = v')$. If $V = \{\}$, *nochange(V)=true*. $\exists W \cdot S$ returns $\{\exists W \cdot S_i | S_i \in S\}$. The entailment $(\exists W{\cdot}S^i_1) \vdash \Phi^i_{po} * S^i_2$ is discharged by the entailment prover described in the next subsection.

At a method call, each of the method's precondition is checked. The combination of the residue S_i and the postcondition is added to the poststate. If a precondition is not entailed by the program state Δ, the corresponding residue is not added to the set of states. The test $\mathtt{S}{\neq}\{\}$ ensures that at least one precondition is satisfied.

$$\frac{\begin{array}{c}[\mathbf{FV{-}CALL}] \\ t_0\ m(t_1\ v_1, .., t_n\ v_n)\ \mathtt{where}\ \{\Phi^i_{pr} \ast\!\!\!\longrightarrow \Phi^i_{po}\}^p_{i=1}\ \{e\} \in P \\ \rho{=}[v'_j/v_j]^n_{j=1} \qquad \Delta \vdash \rho\Phi^i_{pr} * S_i \quad \forall i{=}1, .., p \\ S = \bigcup^p_{i=1} S_i * \Phi^i_{po} \qquad S \neq \{\}\end{array}}{\vdash \{\Delta\}\, m(v_1..v_n)\, \{S\}}$$

Note that the verification rule also invokes the entailment prover to discharge $\Delta \vdash \rho\Phi^i_{pr} * S_i$, where ρ represents a substitution of v_j by v'_j, for all $j = 1, .., n$.. The lifted separation conjunction $*$ over a set (i.e., $S_i * \Phi^i_{po}$) is defined in Fig. 5 in the next subsection.

Our verifier also ensures that each field access is safe from null dereferencing. This is shown in the field access rules in Fig. 4 which also includes other forward verification rules for the language. The verification rules attempt to track heap states, as accurately as possible, with path-sensitivity captured by [FV−IF]

P	$::= tdecl^*\ meth^*$	$tdecl ::= objt \mid spred$
$objt$	$::= \mathtt{data}\ c\ \{\ field^*\ \}$	$field ::= t\ v$
t	$::= c \mid \tau$	$\tau ::= \mathtt{int} \mid \mathtt{bool}$
$meth$	$::= (t \mid \mathtt{void})\ mn\ ((t\ v)^*)\ \mathtt{where}\ mspec\ \{e\}$	
$mspec$	$::= \{\Phi^i_{pr} \ast\!\!\!\longrightarrow \Phi^i_{po}\}^p_{i=1}$	
e	$::= \mathtt{null} \mid k^\tau \mid v \mid v.f \mid v{:=}e \mid v_1.f{:=}v_2$	
	$\mid \mathtt{new}\ c(v^*) \mid e_1; e_2 \mid t\ v;\ e \mid mn(v^*)$	
	$\mid \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$	
	$\mid \mathtt{while}\ v\ \mathtt{where}\ mspec\ \mathtt{do}\ e$	
c, v	$::= identifiers$	

Figure 3. A Core Imperative Language

$$\boxed{\textbf{FV-SPRED}}\quad \frac{XPure_0(\Phi) \Longrightarrow [0/\texttt{null}](\pi_{inv})}{\vdash\ c\langle v^*\rangle \equiv \Phi\ \texttt{inv}\ \pi_{inv}} \qquad \boxed{\textbf{FV-VAR}}\quad \frac{S{=}\{\Delta{\wedge}\texttt{res}{=}v'\}}{\vdash \{\Delta\}\, v\, \{S\}} \qquad \boxed{\textbf{FV-CONST}}\quad \frac{S = \{\Delta{\wedge}eq_\tau(\texttt{res}, k)\}}{\vdash \{\Delta\}\, k^\tau\, \{S\}} \qquad \boxed{\textbf{FV-LOCAL}}\quad \frac{\vdash \{\Delta\}\, e\, \{S\}}{\vdash \{\Delta\}\, \{t\ v;\ e\}\, \{\exists\, v, v'{\cdot}S\}}$$

$$\boxed{\textbf{FV-IF}}\quad \frac{\vdash \{\Delta{\wedge}v'\}\, e_1\, \{S_1\} \quad \vdash \{\Delta{\wedge}\neg v'\}\, e_2\, \{S_2\}}{\vdash \{\Delta\}\, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\, \{S_1{\vee}S_2\}} \qquad \boxed{\textbf{FV-NEW}}\quad \frac{S{=}\{\Delta * \texttt{res}{::}c\langle v_1', .., v_n'\rangle\}}{\vdash \{\Delta\}\, \texttt{new}\ c(v_1, .., v_n)\, \{S\}} \qquad \boxed{\textbf{FV-SEQ}}\quad \frac{\vdash \{\Delta\}\, e_1\, \{S_1\} \quad \vdash \{S_1\}\, e_2\, \{S_2\}}{\vdash \{\Delta\}\, e_1; e_2\, \{S_2\}}$$

$$\boxed{\textbf{FV-ASSIGN}}\quad \frac{\begin{array}{c}\vdash \{\Delta\}\, e\, \{S_1\} \\ S_2{=}\exists \texttt{res}{\cdot}(S_1 \wedge_{\{v\}} v'{=}\texttt{res})\end{array}}{\vdash \{\Delta\}\, v{:=}e\, \{S_2\}} \qquad \boxed{\textbf{FV-FIELD-READ}}\quad \frac{\begin{array}{c}\Delta \vdash v'{::}c\langle v_{1..n}\rangle * S_1 \quad S_1{\neq}\{\} \quad \mathit{fresh}\ v_1..v_n \\ S_2 = \exists v_1..v_n{\cdot}(S_1 * v'{::}c\langle v_{1..n}\rangle {\wedge} \texttt{res}{=}v_i)\end{array}}{\vdash \{\Delta\}\, v.f_i\, \{S_2\}} \qquad \boxed{\textbf{FV-FIELD-UPDATE}}\quad \frac{\begin{array}{c}\Delta \vdash v'{::}c\langle v_{1..n}\rangle * S_1 \quad S_1{\neq}\{\} \quad \mathit{fresh}\ v_1..v_n \\ S_2 = \exists v_1..v_n{\cdot}(S_1 * v'{::}[v_0'/v_i]c\langle v_{1..n}\rangle)\end{array}}{\vdash \{\Delta\}\, v.f_i{:=}v_0\, \{S_2\}}$$

$$\boxed{\textbf{FV-WHILE}}\quad \frac{\begin{array}{c} W{=}vars(v, e) \quad \forall i{\in}\{1..n\} \cdot \Delta \vdash \Phi^i_{pr} * S^i \quad S_1 = \bigcup_{i=1}^n (S^i *_W \Phi^i_{po}) \quad S_1{\neq}\{\} \\ \forall i \in \{1..n\} \cdot (\vdash \{\Phi^i_{pr} {\wedge} nochange(W) {\wedge} v'\}\, e\, \{T^i\} \quad \forall j{\in}\{1..n\} \cdot T^i {\vdash} \Phi^j_{pr} * R^j_i \\ R_i = \bigcup_{j=1}^n (R^j_i *_W \Phi^j_{po}) \quad \Phi^i_{pr} {\wedge} nochange(W) {\wedge} \neg v' \vee R_i {\vdash} \Phi^i_{po} * Z^i \quad Z^i{\neq}\{\}) \end{array}}{\vdash \{\Delta\}\, \texttt{while}\ v\ \texttt{where}\ \{\Phi^i_{pr} \ast\!\!\longrightarrow \Phi^i_{po}\}_{i=1}^n\ \texttt{do}\ e\, \{S_1\}}$$

Figure 4. Forward Verification Rules with Non-Determinism

rule, flow-sensitivity by [FV−SEQ] rule and context sensitivity by the [FV−CALL] rule. In a nutshell, verification is carried out at four places. For each call site, the [FV−CALL] rule (mentioned earlier) ensures that at least one of its method's preconditions is satisfied. At each method definition, the [FV−METH] rule checks that every postcondition holds for the method body assuming its respective precondition. At each `while` loop, [FV−WHILE] checks that each pair of pre/post annotations has been verified for the loop body in a modular fashion, and that at least one of its preconditions is satisfied. At each shape definition, [FV−SPRED] checks that its given invariant π_{inv} is sound w.r.t. (i.e. semantic consequence of) the well-formed heap formula Φ. The function $XPure_0(\Phi)$ generates a sound and heap-independent approximation of the heap constraint Φ. For instance,

$$XPure_0(\texttt{x::node}\langle\_,\_\rangle) \equiv \texttt{x} > \texttt{0}$$
$$XPure_0(\texttt{x::node}\langle\_,\_\rangle * \texttt{y::node}\langle\_,\_\rangle) \equiv \texttt{x>0}{\wedge}\texttt{y>0}{\wedge}\texttt{x}{\neq}\texttt{y}$$
$$XPure_0(\texttt{x::lseg}\langle\texttt{p},\texttt{n}\rangle) \equiv \texttt{n} \geq \texttt{0}$$

For the shape predicate case above, we can get a more precise approximation by unrolling the predicate definition once, for example:

$$XPure_1(\texttt{x::lseg}\langle\texttt{p},\texttt{n}\rangle) \equiv \texttt{x=p}{\wedge}\texttt{n=0} \vee \texttt{x>0}{\wedge}\texttt{n>0}$$

The definition for the general approximation procedure $XPure_n(\Phi)$ (also used in the entailment prover) can be found in [12] where `n` denotes the number of unrollings done on the shape predicates.

The operators $\wedge_{\{v\}}$ (in assignment rule) and $*_W$ (in while rule) are *composition with update* operators. Given a state Δ_1, a state change Δ_2, and a set of variables to be updated $X{=}\{x_1, \ldots, x_n\}$, the composition operator $\oplus_X$ is defined as :

$$\begin{array}{l} \Delta_1 \oplus_X \Delta_2 =_{df} \exists\ r_1..r_n \cdot \rho_1\, \Delta_1 \oplus \rho_2\, \Delta_2 \\ \text{where}\quad r_1, \ldots, r_n \text{ are fresh variables;} \\ \qquad\qquad \rho_1 = [r_i/x_i']_{i=1}^n\ ;\ \rho_2 = [r_i/x_i]_{i=1}^n \end{array}$$

Note that ρ_1 and ρ_2 are substitutions that link each latest value of x_i' in Δ_1 with the corresponding initial value x_i in Δ_2 via a fresh variable r_i. The binary operator $\oplus$ is either $\wedge$ or $*$.

Normalization rules for separation constraints and lifted operators over sets of states are given in Fig. 5.

4.2 Entailment Prover

The other major component of our system is the entailment prover. This prover directly uses the definitions of shape predicates to reason about entailment between two heap formulae. A key novelty is the use of a set of heap states to support non-deterministic entailment. By non-determinism, we mean a search process that returns multiple answers, any one of which indicates a successful verification. More formally, our entailment prover has the form $\Delta_a \vdash \Delta_c * S$ where S is a set of possible residual poststates. The entailment succeeds when S is non-empty, otherwise it is deemed to have failed. When S captures multiple residual states, they signify different search outcomes during proving.

During entailment, each pair of aliased objects/predicates from Δ_a and Δ_c are matched up, whenever they can be made identical. Also, unfolding may

$(\Delta_1 \vee \Delta_2) \wedge \pi$	$\leadsto (\Delta_1 \wedge \pi) \vee (\Delta_2 \wedge \pi)$	$(\exists x \cdot \Delta_1) * \Delta_2$	$\leadsto \exists y \cdot ([y/x]\Delta_1 * \Delta_2)$
$(\Delta_1 \vee \Delta_2) * \Delta$	$\leadsto (\Delta_1 * \Delta) \vee (\Delta_2 * \Delta)$	$(S_1 \vee S_2)$	$\leadsto \{\Delta_1 \vee \Delta_2 \mid \Delta_1 \in S_1,\ \Delta_2 \in S_2\}$
$(\kappa_1 \wedge \pi_1) * (\kappa_2 \wedge \pi_2)$	$\leadsto (\kappa_1 * \kappa_2) \wedge (\pi_1 \wedge \pi_2)$	$F(S)$	$\leadsto \{F(\Delta) \mid \Delta \in S\}$
$(\kappa_1 \wedge \pi_1) \wedge (\pi_2)$	$\leadsto \kappa_1 \wedge (\pi_1 \wedge \pi_2)$	where	
$(\gamma_1 \wedge \phi_1) \wedge (\gamma_2 \wedge \phi_2)$	$\leadsto (\gamma_1 \wedge \gamma_2) \wedge (\phi_1 \wedge \phi_2)$	$F(\mathcal{A}) ::= \mathcal{A} \wedge \pi \mid \mathcal{A} \wedge_W \pi \mid \mathcal{A} * \Delta \mid \mathcal{A} *_W \Delta \mid \exists x \cdot \mathcal{A}$	
$(\exists x \cdot \Delta) \wedge \pi$	$\leadsto \exists y \cdot ([y/x]\Delta \wedge \pi)$	y denotes fresh variable	

Figure 5. Normalization Rules for Separation Constraints and with operators lifted to a Set

be applied to a predicate in the antecedent Δ_a that matches an object in the consequent Δ_c. Alternatively, a predicate in the consequent that is aliased with an object in the antecedent is handled by folding. The folding process is effectively a recursive invocation of entailment to find a set of heap nodes that matches with the predicate's definition. It is different from unfolding a predicate in the antecedent as we allow bindings on free variables to transfer into the antecedent at the end of folding. This is critical for capturing free variables from preconditions that are used in postconditions.

We give the formal rules for the entailment prover in Figure 6. Note that the complete form for the entailment relation is $\Delta_a \vdash^{\kappa}_{V} \Delta_c * S$ which denotes $\kappa * \Delta_a \vdash \exists V \cdot (\kappa * \Delta_c) * S$. As mentioned earlier, the purpose of the entailment prover is to check that heap nodes in the antecedent Δ_a are sufficiently precise to cover all nodes from the consequent Δ_c, and to compute a set of possible residual heap states S (which is empty if the entailment search fails). κ is the history of nodes from the antecedent that have been used to match nodes from the consequent, V is the list of existentially quantified variables from the consequent. Note that κ and V are derived. The entailment prover is invoked with $\kappa = \texttt{emp}$ and $V = \emptyset$.

During the entailment search, whenever a pair of aliased and identical objects/predicates from Δ_a and Δ_c are identified, the [ENT−MATCH] rule is invoked to match them up. If no immediate match can be identified, an unfold/fold operation may be invoked. The rule [ENT−UNFOLD] is to unfold a predicate in the antecedent Δ_a so as to match up with an object in the consequent Δ_c. The rule [ENT−FOLD] is a recursive invocation of the entailment procedure for a predicate from the consequent, in order to identify a set of heap nodes that match with that predicate's definition. When the heap part of the consequent becomes emp, the rule [ENT−EMP] is invoked where the heap part of the antecedent is approximated by a pure constraints via $XPure_n$ and an entailment relation between pure terms are checked.

In [ENT−MATCH] rule, when a match occurs, the bindings between free variables from the matched node in the consequent and the corresponding variables from the antecedent are generated and kept in the antecedent via $freeEqn(\rho, V)$ which is defined as follows:

$$
\begin{array}{l}
freeEqn([u_i/v_i]_{i=1}^{n}, V) =_{df} \\
\quad \texttt{let } \pi_i = (\texttt{if } v_i \in V \texttt{ then } true \texttt{ else } v_i{=}u_i) \\
\quad \texttt{in } \bigwedge_{i=1}^{n} \pi_i
\end{array}
$$

In our system, free variables in consequent are variables from method preconditions. Hence these bindings act as substitutions that have to be kept in antecedent to allow subsequent program state (from residual heap) to be aware of their values.

The test $IsPred(c)$ (resp. $IsObj(c)$) used in the folding and unfolding entailment rules returns true if c is defined as a shape predicate (resp. an object). When a fold to a predicate $p_2{::}c_2\langle v_2^*\rangle$ is performed, the constraints related to variables v_2^* are important. The *split* function projects these constraints out and differentiates those constraints based on free variables.

$$
\begin{array}{l}
split_V^{\{v_2^*\}}(\bigwedge_{i=1}^{n} \pi_i^r) = \\
\quad \texttt{let } (\pi_i^a, \pi_i^c) = \texttt{if } FV(\pi_i^r) \cap v_2^* = \emptyset \texttt{ then } (\texttt{true}, \texttt{true}) \\
\qquad\qquad\qquad\quad \texttt{elseif } FV(\pi_i^r) \cap V = \emptyset \texttt{ then } (\pi_i^r, \texttt{true}) \\
\qquad\qquad\qquad\quad \texttt{else } (\texttt{true}, \pi_i^r) \\
\quad \texttt{in } (\bigwedge_{i=1}^{n} \pi_i^a, \bigwedge_{i=1}^{n} \pi_i^c)
\end{array}
$$

The folding operation [FOLDING] requires a special version of entailment that returns three extra things: (i) consumed heap nodes, (ii) existential variables used, and (iii) final consequent. The final consequent is used to return a constraint for $\{v^*\}$ via $\exists W_i \cdot \pi_i$. A set of answers is returned by the fold step as we allow it to explore multiple ways of matching up with its disjunctive definition.

We also have a rule for handling a set of heap states in the antecedent, as follows:

$$
\frac{\forall i{\in}1..n \quad \Delta_i \vdash^{\kappa}_{V} \Delta * S_i}{\{\Delta_1, .., \Delta_n\} \vdash^{\kappa}_{V} \Delta * (\bigcup_{i=1}^{n} S_i)}
$$

This entailment relation has been used in the forward verification rules [FV−METH] and [FV−WHILE].

5 Implementation

We have built a prototype system using Objective Caml. The proof obligations generated by our verifi-

$$\frac{\rho{=}[0/\texttt{null}] \quad b{=}(\mathit{XPure}_n(\kappa_1{*}\kappa{\wedge}\pi_1) \Longrightarrow \exists V{\cdot}\rho\pi_2)}{\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}(\pi_2) * \{\kappa_1{\wedge}\pi_1 \mid b\}}\;[\underline{\textbf{ENT-EMP}}]$$

$$\frac{\mathit{XPure}_n(p_1{::}c\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1) \Longrightarrow p_1{=}p_2 \quad \rho{=}[v_1^*/v_2^*] \quad \kappa_1{\wedge}\pi_1{\wedge}\mathit{freeEqn}(\rho,V){\vdash}^{\kappa * p_1::c\langle v_1^*\rangle}_{V-\{v_2^*\}}\rho(\kappa_2{\wedge}\pi_2) * S}{p_1{::}c\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}(p_2{::}c\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2) * S}\;[\underline{\textbf{ENT-MATCH}}]$$

$$\frac{\begin{array}{c}\mathit{IsPred}(\mathsf{c}_2){\wedge}\mathit{IsObj}(\mathsf{c}_1) \quad \{(\Delta_i,\kappa_i^r,\pi_i^r)\}_{i=1}^n{=}\mathit{fold}^{\kappa}(p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1, p_2{::}c_2\langle v_2^*\rangle) \\ \mathit{XPure}_n(p_1{::}c\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1) \Longrightarrow p_1{=}p_2 \quad (\pi_i^a,\pi_i^c){=}\mathit{split}_V^{\{v_2^*\}}(\pi_i^r) \;\; \Delta_i{\wedge}\pi_i^a{\vdash}^{\kappa_i^r}_{V}\kappa_2{\wedge}(\pi_2{\wedge}\pi_i^c) * S_i\end{array}}{p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}(p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2) * \bigcup_{i=1}^n S_i}\;[\underline{\textbf{ENT-FOLD}}]$$

$$\frac{\begin{array}{c}\mathit{XPure}_n(p_1{::}c\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1) \Longrightarrow p_1{=}p_2 \quad \mathit{IsPred}(\mathsf{c}_1){\wedge}\mathit{IsObj}(\mathsf{c}_2) \\ \mathit{unfold}(p_1{::}c_1\langle v_1^*\rangle){*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}(p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2) * S\end{array}}{p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}(p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2) * S}\;[\underline{\textbf{ENT-UNFOLD}}]$$

$$\frac{\Delta_1{\vdash}^{\kappa}_{V}\Delta_2 * S_1 \quad \Delta_1{\vdash}^{\kappa}_{V}\Delta_3 * S_2 \quad S{=}S_1{\cup}S_2}{\Delta_1{\vdash}^{\kappa}_{V}(\Delta_2{\vee}\Delta_3) * S}\;[\underline{\textbf{ENT-RHS-OR}}]$$

$$\frac{\Delta_1{\vdash}^{\kappa}_{V}\Delta_3 * S_1 \quad \Delta_2{\vdash}^{\kappa}_{V}\Delta_3 * S_2 \quad S_3{=}\{\Delta_5{\vee}\Delta_6 \mid \Delta_5{\in}S_1, \Delta_6{\in}S_2\}}{\Delta_1{\vee}\Delta_2{\vdash}^{\kappa}_{V}\Delta_3 * S_3}\;[\underline{\textbf{ENT-LHS-OR}}]$$

$$\frac{\Delta_1{\vdash}^{\kappa}_{V\cup\{w\}}([w/v]\Delta_2) * S_1 \quad \mathit{fresh}\ w \quad S{=}\{\exists\ w\cdot\Delta \mid \Delta{\in}S_1\}}{\Delta_1{\vdash}^{\kappa}_{V}(\exists\ v\cdot\Delta_2) * S}\;[\underline{\textbf{ENT-RHS-EX}}]$$

$$\frac{[w/v]\Delta_1{\vdash}^{\kappa}_{V}\Delta_2 * S \quad \mathit{fresh}\ w}{\exists v\cdot\Delta_1{\vdash}^{\kappa}_{V}\Delta_2 * S}\;[\underline{\textbf{ENT-LHS-EX}}]$$

$$\frac{c\langle v^*\rangle{\equiv}\Phi\ \texttt{inv}\ \pi\ \in P}{\mathit{unfold}(p{::}c\langle v^*\rangle)\ =_{df}\ [p/\texttt{root}]\Phi}\;[\underline{\textbf{UNFOLDING}}]$$

$$\frac{c\langle v^*\rangle{\equiv}\Phi\ \texttt{inv}\ \pi\ \in P \quad W_i{=}V_i{-}\{v^*,p\} \quad \kappa{\wedge}\pi{\vdash}^{\kappa'}_{\{p,v^*\}}[p/\texttt{root}]\Phi * \{(\Delta_i,\kappa_i,V_i,\pi_i)\}_{i=1}^n}{\mathit{fold}^{\kappa'}(\kappa{\wedge}\pi, p{::}c\langle v^*\rangle) =_{df} \{(\Delta_i,\kappa_i,\exists W_i{\cdot}\pi_i)\}_{i=1}^n}\;[\underline{\textbf{FOLDING}}]$$

Figure 6. Non-Deterministic Separation Constraint Entailment

cation are discharged by our entailment proving procedure with the help of Omega Calculator [15] for arithmetic constraints of Presburger form and MONA [7] for set constraints.

Our system uses the set-of-states technique to implement multiple pre/post specification, but this technique is orthogonal to multiple pre/post. This is because set-of-states may also arise from disjunction in the consequent from entailment proving, even when single pre/post specifications are used. Our first set of experiments, summarized in Figure 7, was designed to evaluate the cost of supporting set-of-states. We provide statistics on the program size as lines of code and programmer effort as lines of annotation required. We also give the time taken (in seconds) to verify each of those programs. Most of our examples can be verified within 5 seconds despite the use of Presburger solver. Verification time of a function includes time to verify all functions that it calls.

The examples show that proof search via set of states does not incur much overhead, since most of the time there is only one state. The average overhead introduced by the set of states for our examples is around 0.05 seconds. However, set of states is crucial to verify the `count` or `delete` methods of circular list, which is based on the `lseg` predicate, and in those examples that have been marked with a *failed* (to verify). Multiple pre/post specifications are needed in the examples marked with a $\surd$. From the experiments presented in Figure 7, we may conclude that the cost of supporting set of states is quite low. However, set of states may be crucial for some examples, as it may be required when handling disjunctive consequent or multiple pre/post.

Even if multiple pre/post conditions are not critically needed, there may be occasions when they are desirable due to the possibility of relying on smaller and simpler constraints during verification. In order to highlight the performance gains due to the use of multiple pre/post, Figure 8 compares the timings obtained for some programs with single pre/post (capturing the reachability property) that have been translated to equivalent programs with multiple pre/post. This translation exploits the use of smaller and simpler predicates, where possible, which resulted in faster overall verification. As the proof obligations generated for the examples in Figure 8 contain set constraints, we used MONA to discharge them when needed. However, for some examples with single pre/post, the complexity of the constraints caused an out-of-memory error.

As an example of breaking a single specification into multiple pre/post, let's consider the following predicate which describes a non-empty list sorted in ascending

Programs	LOC	LOA	Timing (with Omega solver)		Uses Multi Pre/Post
			(set-of-states)	(one-state)	
Linked List			verifies size/length		
delete	14	4	0.08	0.06	√
reverse	12	2	0.06	0.05	
Circular Linked List			verifies size + cyclic structure		
delete	38	6	0.2	*failed*	√
count	36	4	0.24	*failed*	
Doubly Linked List			verifies size + double links		
append	26	2	0.16	0.12	
flatten (from tree)	4	34	0.35	0.33	
Sorted List			verifies size + min + max + sortedness		
delete	21	4	0.16	0.15	√
insertion_sort	45	6	0.37	0.32	√
selection_sort	52	6	0.34	0.31	√
merge_sort	107	10	0.61	0.56	√
quick_sort	85	10	0.75	0.71	√
AVL Tree			verifies size + height + height-balanced		
insert	169	10	5.06	5.00	
Red-Black Tree			verifies size + black-height + height-balanced		
insert	167	12	1.53	1.39	
delete	352	22	17.44	14.72	
2-3 Tree			verifies height-balanced		
insert	196	16	24.41	*failed*	√
Perfect Tree			verifies perfectness		
insert	70	6	0.28	0.24	
Complete Tree			verifies completeness		
insert	83	10	1.62	1.49	√

Figure 7. Comparison between the verification times (in seconds) with and without set of states

order. The predicate tracks the length of the list, $\mathtt{n}$, the minimum value, $\mathtt{m}$, and also the entire set/bag of values stored in the list, $\mathtt{B}$. The sorting property is ensured by the presence of $\mathtt{m} \leq \mathtt{m}_1$.

$$\begin{array}{l} \mathtt{sllM}\langle\mathtt{n},\mathtt{m},\mathtt{B}\rangle \;\equiv\; \mathtt{root}::\mathtt{node}\langle\mathtt{m},\mathtt{null}\rangle \wedge \mathtt{B}=\{\mathtt{m}\} \wedge \mathtt{n}=1 \\ \quad \vee\ \exists\mathtt{p}\cdot\mathtt{root}::\mathtt{node}\langle\mathtt{m},\mathtt{p}\rangle * \mathtt{p}::\mathtt{sllM}\langle\mathtt{n}-1,\mathtt{m}_1,\mathtt{B}_1\rangle \\ \qquad \wedge\ \mathtt{m}\leq\mathtt{m}_1 \wedge\ \mathtt{B}=\mathtt{B}_1\cup\{\mathtt{m}\} \qquad \mathbf{inv}\ \mathtt{n}>0 \end{array}$$

In order to transform an example using the above predicate into multiple pre/post, `sllM` can be split into three smaller predicates, each of them specialized on a certain property: `llR` which captures the set/bag of values and was already defined in Section 3, `sllm` which captures the sortedness and `ll` which captures the size.

$$\begin{array}{l} \mathtt{sllm}\langle\mathtt{m}\rangle \;\equiv\; \mathtt{root}::\mathtt{node}\langle\mathtt{m},\mathtt{null}\rangle \\ \quad \vee\ \ \exists\mathtt{p}\cdot\mathtt{root}::\mathtt{node}\langle\mathtt{m},\mathtt{p}\rangle * \mathtt{p}::\mathtt{sllm}\langle\mathtt{m}_1\rangle \wedge \mathtt{m}\leq\mathtt{m}_1 \\ \mathtt{ll}\langle\mathtt{n}\rangle \;\equiv\; \mathtt{root}=\mathtt{null} \wedge \mathtt{n}=0 \\ \quad \vee\ \ \exists\mathtt{p}\cdot\mathtt{root}::\mathtt{node}\langle\_,\mathtt{p}\rangle * \mathtt{p}::\mathtt{ll}\langle\mathtt{n}-1\rangle \qquad \mathbf{inv}\ \mathtt{n}\geq 0 \end{array}$$

By applying the above splitting on a single pre/post of insertion sort algorithm (into three smaller pre/post) we have managed to verify it in 3.96 seconds, while with a single pre/post we obtained an out-of-memory error from MONA.

6 Related Work

Recently, separation logic has been used to reason about heap-manipulating programs in various contexts, e.g. locality [5], termination [3], concurrency [13]. Similar to [2], most of these works only support a limited set of predicates. Our recent work [12] allowed size properties to be defined in user-supplied recursive predicates in separation logic, which are then automatically verified via a sound, terminating but incomplete proof system. Building on this work, the current paper advocates the use of multiple pre/post for better expressivity in method specifications using separation logic. We also propose a non-deterministic proof search procedure by carrying over sets of abstract states, which proves to be crucial in verifying certain complicated properties for heap manipulating programs.

On the inference front, Lee et al. [10] has conducted an intraprocedural analysis for loop invariants using grammar approximation under separation logic. Their analysis can handle a wide range of shape predicates with local sharing but is restricted to predicates with two parameters and without size/bag properties. A recent work [6] has also formulated interprocedural shape inference but is restricted to just the list segment shape

Programs	Method	Timing (with Omega and MONA) (single pre/post)	(multiple pre/post)
Linked List	append	2.68	0.42
	insert	0.95	0.4
	reverse	2.7	0.41
Circular Linked List	insert	0.47	0.22
	delete	2.51	0.67
Doubly Linked List	insert	1.51	0.55
	append	29.75	1.8
	flatten (from tree)	*out-of-memory*	5.49
Sorted List	insertion_sort	*out-of-memory*	3.96
	selection_sort	*out-of-memory*	2.6
Binary Search Tree	insert	*out-of-memory*	2.04
	delete	*out-of-memory*	4.7

Figure 8. Comparison between the verification times (in seconds) with and without multiple pre/post

predicate. While our system does not perform inference of pre/post specification, we provide better support for automated verification via an expressive specification mechanism. For example, data structures with strong invariants, such as balanced heights and sortedness, are easily captured by our specification mechanism (with the help of multiple pre/post) prior to automatic verification.

7 Conclusion

We have introduced set of states as a way to support multiple pre/post specifications in separation logic. This approach allows proof search to be captured explicitly, permitting a simple automated search strategy to be developed. Our current search strategy is exhaustive but *directed* and is guaranteed to *terminate*.

Multiple pre/post method specifications enhances proof search in separation logic. This feature puts creative control back into users' hands. Nevertheless, we provide machine support for automatically checking and then applying each high-level specification from the user. We believe that multiple pre/post specifications can greatly enhance both the expressivity and performance of automated verification via separation logic. As shown by our initial experiments, it allows more verifications to be carried out successfully and with potential improvement to performance.

Acknowledgement

This work is supported by the Singapore-MIT Alliance and A*STAR research grant R-252-000-233-205. Shengchao Qin is supported in part by EPSRC grant EP/E021948/1.

References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Int'l Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer-Verlag, LNCS, 2004.

[2] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *APLAS*. Springer-Verlag, November 2005.

[3] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, LNCS, 2006.

[4] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 2005.

[5] D. Distefano, P. W. O'Hearn, and H. Yang. A Local Shape Analysis based on Separation Logic. In *TACAS*. Springer-Verlag, March 2006.

[6] A. Gotsman, J. Berdine, and B. Cook. Interprocedural Shape Analysis with Separated Heap Abstractions. In *SAS*, Springer LNCS, Seoul, Korea, August 2006.

[7] Nils Klarlund and Anders Mller. Mona version 1.4 - user manual.

[8] G. T. Leavens. JML's Rich, Inherited Specifications for Behavioral Subtypes. In *ICFEM*, Macao, China, November 2006. Springer-Verlag.

[9] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Mller, and J. Kiniry. JML Reference Manual (DRAFT), February 2007.

[10] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*. Springer Verlag, April 2005.

[11] K.R.M. Leino and P. Muller. A verification methodology for model fields. In *15th ESOP*, March 2006.

[12] H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. In *VMCAI*, Nice, France, January 2007.

[13] M Parkinson, R Bornat, and P O'Hearn. Modular verification of a non-blocking stack. In *ACM POPL*, Nice, France, January 2007.

[14] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, 1991.

[15] W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.