Enhancing Program Verification with Lemmas

Huu Hai Nguyen and Wei-Ngan Chin

Department of Computer Science, National University of Singapore

Abstract. One promising approach to verifying heap-manipulating programs is based on *user-defined* inductive predicates in separation logic. This approach can describe data structures with complex invariants and sound reasoning based on unfold/fold. However, an important component towards more expressive program verification is the use of *lemmas* that can soundly relate predicates beyond their original definitions. This paper outlines a new *automatic* mechanism for proving and applying *user-specified lemmas* under separation logic.

Keywords : Lemma Proving, Lemma Application, Program Verification, Separation Logic, Entailment.

1 Introduction

Inductive predicates based on separation logic [22, 16] offer an important approach to the specification of data structures that make extensive use of pointers and require sophisticated invariants. The technique brings the conveniences of algebraic data structures to the imperative settings, including precise yet simple and intuitive data structure definitions. It also enables effective and automatic reasoning based on the folding and unfolding of predicate definitions, and can verify programs over a wide range of interesting data structures. However, there are some crucial limitations in existing automated verification systems that rely solely on the unfold/fold mechanism. Firstly, it constrains traversals of a data structure to links explicitly allowed by the recursively defined predicates. These are typically top-down unravelling of the data structures, in that a program first accesses the "root" of a data structure, then any of its (non-dangling) fields that can be shown pointing to other objects or data structures. Secondly, the unfold/fold reasoner cannot discover auxiliary relations between predicates that may require inductive proofs.

In this work, we propose a new mechanism that aims to address the aforementioned shortcomings. The main idea is to explicitly state any auxiliary relations between predicate definitions, so that a deductive mechanism based on unfold/fold can prove and use them. This information is presented to the system in the form of *lemmas* that can be viewed as auxiliary relations for the predicates, apart from their definitions. These auxiliary properties can capture different linkage patterns in the data structure. They can also reveal complex relations between distinct but related predicates. Currently, user effort is required in stating the lemmas. Nevertheless, once stated, each of these lemmas is automatically *proven* once and *applied* many times, without further user assistance.

As the need for lemmas in theorem proving is well-known, our contribution is not on the lemmas per se, but rather on the mechanisms to prove and apply them for automated verification via separation logic. These mechanisms are non-trivial, especially for handling more complex lemmas. We shall show that our procedure is sound, terminates and is directed. Our specific contributions are:

- **Alternative Traversals.** Lemmas provide different ways to reason about inductive predicates, which allows *alternative traversals* of data structures that are not captured by the original predicate definitions.
- **Complex Subsumption.** Predicates are often related to one another in complex ways (possibly involving multiple predicates from a heap state with side conditions) that may require inductive proofs. Lemmas provide an explicit way to capture such *complex subsumptions* between heap states for use through the deductive mechanism based on unfold/fold reasoning.
- **Lemma Proving.** To prove lemmas automatically, we use the same deductive mechanism as our entailment checker, after an initial unfold on the *base* predicate in the antecedent. The lemma itself can be applied during proving, when needed, which corresponds to a cyclic proof by infinite descent [5, 4]. Our proposal can be viewed as a special case of [4] since it is based on a fragment of separation logic. However, we have succeeded in providing an automated procedure for cyclic proving under this fragment which is highly suited for program verification via forward reasoning.
- **Lemma Application.** Our program verifier can also apply the lemmas describing auxiliary relationships between predicates by automatically coercing one predicate to another, as needed. Coercion provides suitable transformations on formulas that facilitate *proof search* to enhance the capability of automated verification. Our coercion mechanism is goal-directed and terminates.

2 Examples

We now illustrate the usefulness of lemmas in program verification with an example which shows the ability of lemmas to provide alternative unfoldings and foldings of predicates, thereby providing different ways to reveal points-to facts not apparent in the original definitions of predicates. Let us consider the following class and predicate definitions.

$$\texttt{class node} \; \{ \; \texttt{int val}; \; \texttt{node next} \}$$

$$\texttt{class node2} \; \{ \; \texttt{int val}; \; \texttt{node2 prev}; \; \texttt{node2 next} \}$$

$$\texttt{root}{::}\texttt{ll}\langle \texttt{s} \rangle \equiv \texttt{root}{=}\texttt{null} \wedge \texttt{s}{=}0 \;\vee\; \exists \texttt{r}{\cdot}\texttt{root}{::}\texttt{node}\langle \_, \texttt{r} \rangle * \texttt{r}{::}\texttt{ll}\langle \texttt{s}{-}1 \rangle \; \textbf{inv} \; \texttt{s}{\geq}0;$$

$$\begin{array}{l}\texttt{root}{::}\texttt{dsegN}\langle \texttt{s}, \texttt{p}, \texttt{n}, \texttt{t} \rangle \equiv \texttt{root}{=}\texttt{n} \wedge \texttt{p}{=}\texttt{t} \wedge \texttt{s}{=}0 \;\vee \\ \qquad \exists \texttt{r}{\cdot}\texttt{root}{::}\texttt{node2}\langle \_, \texttt{p}, \texttt{r} \rangle * \texttt{r}{::}\texttt{dsegN}\langle \texttt{s}{-}1, \texttt{root}, \texttt{n}, \texttt{t} \rangle \; \textbf{inv} \; \texttt{s}{\geq}0;\end{array}$$

$$\begin{array}{l}\texttt{root}{::}\texttt{dcl}\langle \texttt{s} \rangle \equiv \texttt{root}{=}\texttt{null} \wedge \texttt{s}{=}0 \;\vee \\ \qquad \exists \texttt{r}_1, \texttt{r}_2{\cdot}\texttt{root}{::}\texttt{node2}\langle \_, \texttt{r}_1, \texttt{r}_2 \rangle * \texttt{r}_2{::}\texttt{dsegN}\langle \texttt{s}{-}1, \texttt{root}, \texttt{root}, \texttt{r}_1 \rangle \; \textbf{inv} \; \texttt{s}{\geq}0;\end{array}$$

Predicate `ll` defines a linear-linked list of length `s`. Predicate `dsegN`, adopted from [11], defines a doubly-linked list segment. Parameter `s` denotes its length, while `p` is the dangling `prev` field of the first element, `n` is the dangling `next` field of the last element which is also pointed to by `t`. The `dcl` predicate defines a circular list by making the dangling pointer of the `dsegN` predicate point to the same distinguished `root` node, thereby making a cycle.

Details of our specification language is given in Sec 3. Briefly, each predicate describes a data structure, which is a collection of objects reachable from a base pointer denoted by `root` in the predicate definition. `root` also serves as an implicit argument of the predicate. The expression after **inv** keyword captures a *pure*, i.e. *heap-independent*, formula that always holds for the given predicate. Formula $\mathtt{p}::\mathtt{c}\langle \mathtt{v}^*\rangle$ denotes either a points-to fact if `c` is a class name, or an instance of predicate `c` with $\mathtt{p}, \mathtt{v}^*$ as its arguments, where `p` is the actual argument for `root` and $\mathtt{v}^*$ are arguments for the explicit parameters.

The `dsegN` predicate, by its definition, favors one direction of linkage. Traversing the list in a forward manner by following the `next` field is naturally supported by the definition with unfold/fold reasoning. However, traversing the list in a backward manner using the `prev` field is not as easily done. The problem manifests itself in, for example, the following `delete` procedure for a circular doubly-linked list. The procedure deletes the element pointed by `x` and updates `x`. The precondition requires the circular list to be non-empty, and the postcondition asserts that the updated `x` points to a circular list with one fewer element.

1 **void** delete(**ref** node2 x)
2   **requires** $\mathrm{x}::\mathrm{dcl}\langle \mathrm{s}\rangle \wedge \mathrm{s} \geq 1$
3   **ensures** $\mathrm{x'}::\mathrm{dcl}\langle \mathrm{s}-1\rangle$;
4 {
5   **if** (x.next == x) x = null;
6   **else** {
7   // $\mathtt{x}::\mathtt{node2}\langle \_, \mathtt{r}_1, \mathtt{r}_2\rangle * \mathtt{r}_2::\mathtt{dsegN}\langle \mathtt{s}-1, \mathtt{x}, \mathtt{x}, \mathtt{r}_1\rangle \wedge \mathtt{s} \geq 2$
8   node tmp = x.prev;
9   // $\mathtt{x}::\mathtt{node2}\langle \_, \mathtt{r}_1, \mathtt{r}_2\rangle * \mathtt{r}_2::\mathtt{dsegN}\langle \mathtt{s}-1, \mathtt{x}, \mathtt{x}, \mathtt{r}_1\rangle \wedge \mathtt{tmp} = \mathtt{r}_1 \wedge \mathtt{s} \geq 2$
10   tmp.next = x.next;
11   // $\mathtt{x}::\mathtt{node2}\langle \_, \mathtt{r}_1, \mathtt{r}_2\rangle * \mathtt{r}_2::\mathtt{dsegN}\langle \mathtt{s}-2, \mathtt{x}, \mathtt{r}_1, \mathtt{r}_3\rangle * \mathtt{r}_1::\mathtt{node2}\langle \_, \mathtt{r}_3, \mathtt{r}_2\rangle$
12   // $\quad \wedge\, \mathtt{tmp} = \mathtt{r}_1 \wedge \mathtt{s} \geq 2$
13   x.next.prev = x.prev;
14   // $\quad \mathtt{x}::\mathtt{node2}\langle \_, \mathtt{r}_1, \mathtt{r}_2\rangle * \mathtt{r}_1::\mathtt{node2}\langle \_, \mathtt{r}_1, \mathtt{r}_2\rangle \wedge \mathtt{r}_2 = \mathtt{r}_1 \wedge \mathtt{x} = \mathtt{r}_3 \wedge \mathtt{s} = 2 \wedge \mathtt{tmp} = \mathtt{r}_1$
15   // $\vee\; \mathtt{x}::\mathtt{node2}\langle \_, \mathtt{r}_1, \mathtt{r}_2\rangle * \mathtt{r}_2::\mathtt{node2}\langle \_, \mathtt{r}_3, \mathtt{r}_4\rangle * \mathtt{r}_4::\mathtt{dsegN}\langle \mathtt{s}-3, \mathtt{r}_2, \mathtt{r}_1, \mathtt{r}_3\rangle$
16   // $\quad\quad * \mathtt{r}_1::\mathtt{node2}\langle \_, \mathtt{r}_3, \mathtt{r}_2\rangle \wedge \mathtt{s} \geq 3$
17   x = x.next ; }}

Fig. 1. Delete from circular list

For exposition purpose, intermediate program states are given as comments (after $//$) in the code, though they are automatically derived from the initial precondition. To verify the assignment to `tmp.next` at line `10`, the program verifier requires an explicit points-to fact $\mathtt{tmp}::\mathtt{node2}\langle \_, \_, \_\rangle$. This is enforced by the following entailment where

Φ_R is inferred.

$$\texttt{x::node2}\langle\_,\texttt{r}_1,\texttt{r}_2\rangle * \texttt{r}_2\texttt{::dsegN}\langle\texttt{s}-1,\texttt{x},\texttt{x},\texttt{r}_1\rangle \wedge \texttt{tmp}{=}\texttt{r}_1 \wedge \texttt{s}{\geq}2$$
$$\vdash \texttt{tmp::node2}\langle\_,\_,\_\rangle * \Phi_R$$

This proof obligation is challenging for reasoning based on unfolding and folding of inductive definitions [16], since the dsegN predicate does not explicitly state that the parameter t points to an object when the data structure is non-empty. Fortunately, the problem can be solved by adopting the following two-way equivalence lemma.

$$\texttt{root::dsegN}\langle\texttt{s},\texttt{p},\texttt{n},\texttt{t}\rangle \wedge \texttt{s}{>}0 \leftrightarrow \exists\texttt{r}\cdot\texttt{root::dsegN}\langle\texttt{s}-1,\texttt{p},\texttt{t},\texttt{r}\rangle * \texttt{t::node}\langle\_,\texttt{r},\texttt{n}\rangle \quad (1)$$

3 Specification Language

Figure 2 shows the grammar for our specification language that has been mostly adopted from [16] except for lemma specifications. Shape predicate spred is the main specification construct that provides data structure descriptions. Formulas are compiled to an internal representation in which arguments for heap formulas are distinct and fresh. Additional existentially quantified variables are introduced if necessary to obtain this normal form.

$$\begin{array}{rcl}
\textit{Predicate}\ \mathsf{spred} & ::= & [\texttt{root::}]c\langle v^*\rangle \equiv \Phi\ [\mathbf{inv}\ \pi] \\
\textit{Formula}\ \ \Phi & ::= & \bigvee \exists v^* \cdot (\kappa \wedge \pi) \\
\textit{Pure form.}\ \ \pi & ::= & \gamma \wedge \phi \\
\textit{Pointer form.}\ \ \gamma & ::= & v_1 = v_2 \mid v = \texttt{null} \mid v_1 \neq v_2 \mid v \neq \texttt{null} \mid \gamma_1 \wedge \gamma_2 \\
\textit{Heap form.}\ \ \kappa & ::= & \mathbf{emp} \mid v\texttt{::}c\langle v^*\rangle \mid \kappa_1 * \kappa_2 \\
\textit{Presburger arith.}\ \ \phi & ::= & \mathsf{arith} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi \\
\mathsf{arith} & ::= & \mathsf{a}_1 = \mathsf{a}_2 \mid \mathsf{a}_1 \neq \mathsf{a}_2 \mid \mathsf{a}_1 < \mathsf{a}_2 \mid \mathsf{a}_1 \leq \mathsf{a}_2 \\
\mathsf{a} & ::= & k \mid v \mid k \times \mathsf{a} \mid \mathsf{a}_1 + \mathsf{a}_2 \mid -\mathsf{a} \mid \mathbf{max}(\mathsf{a}_1,\mathsf{a}_2) \mid \mathbf{min}(\mathsf{a}_1,\mathsf{a}_2) \\
\textit{Lemma}\ \ L & ::= & H \wedge G \bowtie B \\
\textit{Complex Lemma}\ \ L' & ::= & \forall v^* \cdot ((H * E) \wedge G \rightarrow B) \\
\textit{Head}\ \ H & ::= & [\texttt{root::}]c\langle v^*\rangle \\
\textit{ExtraHeap}\ \ E & ::= & \kappa \\
\textit{Body}\ \ B & ::= & \Phi \\
\textit{Guard}\ \ G & ::= & \pi \\
\bowtie & ::= & \rightarrow \mid \leftarrow \mid \leftrightarrow \\
k & \in & \text{Integer constants} \\
v,c & \in & \text{Identifiers}
\end{array}$$

Fig. 2. Grammar for Shape Predicates and Lemmas

Recursive shape predicate definitions need to satisfy certain syntactic restrictions, namely *well-formed* and *well-founded* conditions, to ensure soundness and termination of static reasoning. *Well-formed* conditions ensure that shape predicates and formulas do not admit garbage. They thus disallow predicates such as $\texttt{root::p}\langle\rangle \equiv \exists\texttt{x}\cdot\texttt{root::node}\langle\_,\_\rangle * \texttt{x::node}\langle\_,\_\rangle$ where $\texttt{x::node}\langle\_,\_\rangle$ is garbage as it is inaccessible from the free variables. *Well-founded* conditions disallow root to be passed as argument to a recursive predicate invocation. That means root either is null, dangles, or points to an object which ensures a decreasing heap with each recursive predicate instance.

We now describe a special class of lemmas L allowed by our new specification language. Each L lemma consists of a *head* H and a *body* B. The head H is a single predicate. The *guard* G is a pure formula whose variables are solely from H, which can be omitted if it is true. The body B is a formula in separation logic. The direction $\bowtie$ of a lemma constrains its applicability. The lemmas are divided into three groups, namely : (i) *weakening* lemmas using $\rightarrow$, (ii) *strengthening* lemmas using $\leftarrow$, and (iii) *equivalence* lemmas using $\leftrightarrow$. We expect lemmas to be *well-formed* and *well-founded*, but allows the root parameter to reference a predicate. These lemmas have a similar format as user-defined predicates and can therefore be handled by the same unfold/fold mechanism of our prover, except that it can be goal-directed.

However, we are also interested to support lemmas with more general LHS and with universally quantified variables in the guard. These more complex lemmas are captured by L' in Fig 2 as a weakening lemma. There is no need to consider a strengthening version of complex lemma since it can be converted to L'-form by swapping the two sides. To illustrate the utility of complex lemma, consider a list segment predicate below:

$$\mathtt{root{::}lseg}\langle \mathtt{p}, \mathtt{s}\rangle \equiv \mathtt{root{=}p} \wedge \mathtt{s{=}0} \ \vee\ \exists \mathtt{r}\cdot\mathtt{root{::}node}\langle \_, \mathtt{r}\rangle * \mathtt{r{::}lseg}\langle \mathtt{p}, \mathtt{s}{-}1\rangle \ \mathbf{inv}\ \mathtt{s}{\geq}0;$$

One simple L-form lemma to support list segment breaking and joining is:

$$\mathtt{root{::}lseg}\langle \mathtt{p}, \mathtt{n}\rangle \leftrightarrow \exists \mathtt{a}, \mathtt{b}, \mathtt{r}\cdot \mathtt{root{::}lseg}\langle \mathtt{r}, \mathtt{a}\rangle * \mathtt{r{::}lseg}\langle \mathtt{p}, \mathtt{b}\rangle \wedge \mathtt{n{=}a{+}b} \wedge \mathtt{a{,}b}{\geq}0$$

However, this lemma cannot support entailment proving that requires the capture of size properties for broken segments, such as the following:

$$\mathtt{x{::}lseg}\langle \mathtt{p}, \mathtt{n}\rangle \wedge \mathtt{n{=}8} \vdash \exists \mathtt{r}\cdot \mathtt{x{::}lseg}\langle \mathtt{r}, \mathtt{a}\rangle * \mathtt{r{::}lseg}\langle \mathtt{p}, \mathtt{b}\rangle \wedge \mathtt{a{=}2} \wedge \mathtt{b{=}6}\ *\ \varPhi_R$$

To support the above entailment, we require a more general L'-form lemma where some variables in the guard, such as a and b, are universally quantified, as follows:

$$\forall \mathtt{a}, \mathtt{b}\cdot (\mathtt{root{::}lseg}\langle \mathtt{p}, \mathtt{n}\rangle \wedge \mathtt{n{=}a{+}b} \wedge \mathtt{a{,}b}{\geq}0 \rightarrow \exists \mathtt{r}\cdot \mathtt{root{::}lseg}\langle \mathtt{r}, \mathtt{a}\rangle * \mathtt{r{::}lseg}\langle \mathtt{p}, \mathtt{b}\rangle)$$

Such lemmas allow universally quantified variables to be instantiated which can crucially increase the expressive power of our entailment prover. They can be provided for the list segment with length property, but not for the list segment with bag of values property. Furthermore, there are also lemmas with multiple predicates on the LHS. An example of this was used in [1] for a decidable fragment of separation logic to safely break a class of non-touching list segments. (Our thanks to Peter O'Hearn for highlighting the importance of complex lemmas to us.)

4 Entailment

Given formulas $\varPhi_1$ and $\varPhi_2$, our entailment prover checks if $\varPhi_1$ entails $\varPhi_2$, that is if in any heap satisfying $\varPhi_1$, we can find a subheap satisfying $\varPhi_2$. Moreover, we determine a formula $\varPhi_R$ for the residue heap state which captures the frame condition. Formally, our entailment relation is defined as follows:

Definition 4.1 (Entailment) *A formula Φ_1 entails a formula Φ_2 with residue Φ_R iff*

$$\forall s, h_1 \cdot s, h_1 \models \Phi_1 \Rightarrow \exists h_2, h_R \cdot h_1 = h_2 * h_R \wedge s, h_2 \models \Phi_2 \wedge s, h_R \models \Phi_R$$

The main features of our entailment prover are that, besides determining if the above relation holds, it also *infers* the *residual heap* of the entailment, that is a formula Φ_R such that $s, h_R \models \Phi_R$ and *derives* the predicate parameters. These two features are important for program verification tasks using forward analysis. The relation is formalized using judgment of the form where κ denotes the consumed heap and V is the set of existential variables encountered :

$$\Phi_1 \vdash^{\kappa}_{V} \Phi_2 * \Phi_R$$

A sound and terminating proof system for the above entailment relation is presented in [16]. That system relies on unfolding and folding of the predicate definitions to compute the subheap of Φ_1 that matches Φ_2 and the residue Φ_R. In the current paper, additional proof rules that handle user-supplied lemmas shall be presented which greatly enhances our entailment prover. We provide a re-cap on the unfold/fold mechanisms.

We apply an *unfold* operation on a predicate in the antecedent that matches with an object in the consequent. For instance, when checking:

$$\mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle \wedge \mathtt{n}{>}3 \vdash (\exists \mathtt{r} \cdot \mathtt{x}{::}\mathtt{node}\langle\_, \mathtt{r}\rangle * \mathtt{r}{::}\mathtt{node}\langle\_, \mathtt{y}\rangle \wedge \mathtt{y}{\neq}\mathtt{null}) * \Phi_\mathtt{R}$$

where $\Phi_\mathtt{R}$ is the residue, we unfold the $\mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle$ heap formula in the antecedent twice to match two objects in the consequent. This results in the following reductions towards a residual state:

$$\begin{array}{ll}
\exists \mathtt{q_1}\cdot\mathtt{x}{::}\mathtt{node}\langle_, \mathtt{q_1}\rangle{*}\mathtt{q_1}{::}\mathtt{ll}\langle\mathtt{n}{-}1\rangle{\wedge}\mathtt{n}{>}3 & \vdash (\exists\mathtt{r}\cdot\mathtt{x}{::}\mathtt{node}\langle_, \mathtt{r}\rangle{*}\mathtt{r}{::}\mathtt{node}\langle_, \mathtt{y}\rangle{\wedge}\mathtt{y}{\neq}\mathtt{null}) * \Phi_\mathtt{R} \\
\mathtt{q_1}{::}\mathtt{ll}\langle\mathtt{n}{-}1\rangle{\wedge}\mathtt{n}{>}3 & \vdash (\mathtt{q_1}{::}\mathtt{node}\langle_, \mathtt{y}\rangle \wedge \mathtt{y}{\neq}\mathtt{null}) * \Phi_\mathtt{R} \\
\exists \mathtt{q_2}\cdot\mathtt{q_1}{::}\mathtt{node}\langle_, \mathtt{q_2}\rangle{*}\mathtt{q_2}{::}\mathtt{ll}\langle\mathtt{n}{-}2\rangle{\wedge}\mathtt{n}{>}3 & \vdash \mathtt{q_1}{::}\mathtt{node}\langle_, \mathtt{y}\rangle{\wedge}\mathtt{y}{\neq}\mathtt{null} * \Phi_\mathtt{R} \\
\mathtt{q_2}{::}\mathtt{ll}\langle\mathtt{n}{-}2\rangle{\wedge}\mathtt{n}{>}3{\wedge}\mathtt{q_2}{=}\mathtt{y} & \vdash \mathtt{y}{\neq}\mathtt{null} * \Phi_\mathtt{R}
\end{array}$$

We apply a *fold* operation when an object in the antecedent is aliased with a predicate in the consequent. An example is:

$$\mathtt{x}{::}\mathtt{node}\langle 1, \mathtt{q_1}\rangle{*}\mathtt{q_1}{::}\mathtt{node}\langle 2, \mathtt{null}\rangle{*}\mathtt{y}{::}\mathtt{node}\langle 3, \mathtt{null}\rangle \ \vdash\ \mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle{\wedge}\mathtt{n}{>}1 * \Phi_\mathtt{R}$$

The fold step may be recursively applied but is guaranteed to terminate for well-founded predicates. Furthermore, the fold operation may introduce bindings for free parameters of the folded predicate. In the above, we obtain $\mathtt{n}{=}2$ which may be transferred to the antecedent since $\mathtt{n}$ is free. This allows our folding step to finally derive $\mathtt{y}{::}\mathtt{node}\langle 3, \mathtt{null}\rangle \wedge \mathtt{n}{=}2 \vdash \mathtt{n}{>}1 * \Phi_\mathtt{R}$ from which we will obtain $\Phi_\mathtt{R} = \mathtt{y}{::}\mathtt{node}\langle 3, \mathtt{null}\rangle \wedge \mathtt{n}{=}2$.

5 Lemma Application

User-supplied lemmas are *proved* and *applied* to support sound proof search by the entailment prover. Since the proof of a lemma may apply the lemma itself inductively, we first present the proof rules that apply lemmas. Depending on whether the lemma is applied to the antecedent or the consequent of the entailment, our entailment prover treats it as an unfolding or folding, respectively. $\leftarrow$ lemmas can be applied to only the consequent of an entailment, $\rightarrow$ to only the antecedent, and $\leftrightarrow$ to both.

5.1 Weakening the Antecedent by Lemma Unfolding

A lemma $H \wedge G \bowtie B$ where $\bowtie$ is $\rightarrow$ or $\leftrightarrow$ can be seen as an alternative way to unfold a predicate. Its application is formalized below which says that the lemma is applied if we can find a substitution ρ that matches H to $p_1{::}c_1\langle v_1^*\rangle$ and satisfies the guard.

$$\frac{\begin{array}{c}\boxed{\mathbf{L{-}LEFT}}\\ \mathit{IsPred}(c_1) \qquad p_1{::}c_1\langle v_1^*\rangle * \kappa_1 \wedge \pi_1 \vdash \rho G \\ \rho = \mathit{match}(H, p_1{::}c_1\langle v_1^*\rangle) \quad (\rho B) * \kappa_1 \wedge \pi_1 \vdash_V^{\kappa * p_1::c_1\langle v_1^*\rangle} (\kappa_2 \wedge \pi_2) * \Phi\end{array}}{p_1{::}c_1\langle v_1^*\rangle * \kappa_1 \wedge \pi_1 \vdash_V^{\kappa} (\kappa_2 \wedge \pi_2) * \Phi}$$

where $\Phi \vdash \pi$ checks if guard π holds under Φ, and *match* is defined as:

$$\mathit{match}(p_1{::}c\langle v_1^*\rangle, p_0{::}c\langle v_0^*\rangle) \stackrel{\text{def}}{=} [p_1 \mapsto p_0, v_1^* \mapsto v_0^*]$$

For a goal-directed lemma application, we shall only apply this rule when there exists a predicate $\mathtt{p}_2{::}\mathtt{c}_2\langle \mathtt{v}_2^*\rangle \in \kappa_2$ in the consequent that would (subsequently) match up via aliasing with a $\mathtt{p}_2{::}\mathtt{c}_2\langle \mathtt{v}_3^*\rangle$ in the RHS of lemma $\rho\mathtt{B}$ where $\mathtt{p}_2 \in \{\mathtt{p}_1, \mathtt{v}_1^*\}$.

We now show how a lemma can help verify the `delete` procedure, in particular during an assignment to the `prev` field of the `tmp` object at line `10`. As part of the verification, the following entailment needs to be checked, where the antecedent denotes program state at that program point.

$$\begin{array}{l} \quad \mathtt{x}{::}\mathtt{node2}\langle \_, \mathtt{r}_1, \mathtt{r}_2\rangle * \mathtt{r}_1{::}\mathtt{dsegN}\langle \mathtt{r}_3, \_, \_, \mathtt{r}_2\rangle \wedge \mathtt{tmp} = \mathtt{r}_2 \\ \quad \wedge\, \mathtt{r}_3 = \mathtt{s} - 1 \wedge \mathtt{s} > 1 \vdash \mathtt{tmp}{::}\mathtt{node2}\langle \_, \_, \_\rangle * \Phi_R \\ \rightsquigarrow (\boxed{\mathbf{L{-}LEFT}}) \\ \quad \mathtt{x}{::}\mathtt{node2}\langle \_, \mathtt{r}_1, \mathtt{r}_2\rangle * \mathtt{r}_1{::}\mathtt{dsegN}\langle \mathtt{r}_4, \_, \mathtt{r}_2, \_\rangle * \mathtt{r}_2{::}\mathtt{node2}\langle \_, \_, \_\rangle \\ \quad \wedge\, \mathtt{tmp} = \mathtt{r}_2 \wedge \mathtt{r}_4 = \mathtt{r}_3 - 1 \wedge \mathtt{r}_3 = \mathtt{s} - 1 \wedge \mathtt{s} > 1 \vdash \mathtt{tmp}{::}\mathtt{node2}\langle \_, \_, \_\rangle * \Phi_R \\ \rightsquigarrow (\boxed{\mathbf{ENT{-}MATCH}}) \\ \quad \mathit{Success} \end{array}$$

After the above goal-directed lemma application, we can reveal a match up between $\mathtt{r}_2{::}\mathtt{node2}\langle \_, \_, \_\rangle$ (from the lemma) and $\mathtt{tmp}{::}\mathtt{node2}\langle \_, \_, \_\rangle$ (from the consequent), before successfully proving the entailment.

Our proposal also handles the more complex lemma form: $\forall v^* \cdot (H * E \wedge G \rightarrow B)$. We have designed and implemented it as follows:

$$\frac{\begin{array}{c}\boxed{\mathbf{L{-}LEFT{-}COMPLEX}}\\ \mathit{IsPred}(c_1) \qquad \rho = \mathit{match}(H, p_1{::}c_1\langle v_1^*\rangle) \\ \kappa_1 \wedge \pi_1 \vdash_V^{\kappa * p_1::c_1\langle v_1^*\rangle} \rho E * \Phi_1 \\ \rho B * ([(v \mapsto ?)^*]{\vdash}\rho G) * \Phi_1 \wedge \pi_1 \vdash_V^{\kappa * p_1::c_1\langle v_1^*\rangle} (\kappa_2 \wedge \pi_2) * \Phi\end{array}}{p_1{::}c_1\langle v_1^*\rangle * \kappa_1 \wedge \pi_1 \vdash_V^{\kappa} (\kappa_2 \wedge \pi_2) * \Phi}$$

To support the above proof rule, we provide a new delayed guard $([(v \mapsto ?)^*]{\vdash}\rho G)$ that is used to support the instantiations of v^* when the body ρB is being matched by our entailment procedure. Once v^* have been instantiated, we test the guard G before its instantiations are added to the antecedent. The use of lemmas with universal variables, where possible, allows stronger proofs to be asserted than what is possible using corresponding lemmas with existentially quantified variables. In our approach, this is realised by a novel instantiation mechanism from the delayed guard construct.

5.2 Strengthening the Consequent by Lemma Folding

A lemma $H \wedge G \bowtie B$ where $\bowtie$ is $\leftarrow$ or $\leftrightarrow$ provides an alternative way to fold a predicate. Its application is formalized as follows:

$$\frac{\begin{array}{c}\boxed{\textbf{L-RIGHT}}\\ \mathit{IsPred}(c_2) \qquad \rho = \mathit{match}(H, p_2{::}c_2\langle v_2^*\rangle) \qquad \kappa_1 \wedge \pi_1 \vdash \rho G \\ (\Phi^r, \kappa^r, \pi^r) \in \mathit{foldL}^{\kappa}(\kappa_1 \wedge \pi_1, p_2{::}c_2\langle v_2^*\rangle, \rho B) \\ (\pi^a, \pi^c) = \mathit{split}_V^{\{v_2^*\}}(\pi^r) \qquad \Phi^r \wedge \pi^a \vdash_V^{\kappa^r} (\kappa_2 \wedge \pi_2 \wedge \pi^c) * \Phi \end{array}}{\kappa_1 \wedge \pi_1 \vdash_V^{\kappa} (p_2{::}c_2\langle v_2^*\rangle * \kappa_2 \wedge \pi_2) * \Phi}$$

foldL performs folding using a lemma instead of the body of a predicate.

$$\frac{\begin{array}{c}\boxed{\textbf{L-FOLD}}\\ W_i = V_i - \{v^*, p\} \qquad \kappa \wedge \pi \vdash_{\{p,v^*\}}^{\kappa'} \rho B * \{(\Phi_i, \kappa_i, V_i, \pi_i)\}_{i=1}^{n}\end{array}}{\mathit{foldL}^{\kappa'}(\kappa \wedge \pi, p{::}c\langle v^*\rangle, \rho B) \stackrel{\text{def}}{=} \{(\Phi_i, \kappa_i, \exists W_i \cdot \pi_i)\}_{i=1}^{n}}$$

Note that the folding function *foldL* uses a specialized entailment checking procedure. The checker returns a set of quadruples $(\Phi^r, \kappa^r, V, \pi^r)$, each being the result of a successful folding against a disjunct of the predicate definition or the lemma body. The meaning of each component of a triple is as following:

- Φ^r is the residue (frame) not consumed by the folded disjunct.
- κ^r is the part of the heap consumed by the folded disjunct. By definition, $\kappa^r * \Phi^r$ equals the heap in the first argument of $foldL$.
- V is the set of existential variables generated from unfoldings of the predicate definition.
- π^r is the pure constraint of the folded disjunct. It is used to obtain information, such as bindings to values, for predicate parameters. This information is especially useful for forward verification.

This use of a *set of states* can be generalized to the entire system which results in entailment proving of the form $\Phi_A \vdash \Phi_C * S$ that has been implemented in our tool. Here, S denotes a set of residual heap states that arise from proof search for successful entailment. Failure of entailment is denoted by S=$\{\}$, while multiple answers denote alternative successful outcomes of entailment with the respective residual heaps. Proof search (with the help of lemmas) increases the expressivity of our verifier.

5.3 An Example of Entailment with Lemma Capability

An interesting application of lemma involves the list-with-tail predicate which is defined as follows:

$$\begin{array}{l}\texttt{root::ll\_tail}\langle\texttt{tx},\texttt{n}\rangle \equiv \texttt{root::node}\langle\_,\texttt{null}\rangle \wedge \texttt{n} = 1 \wedge \texttt{tx} = \texttt{root} \\ \quad \vee\ \texttt{root::node}\langle\_,\texttt{r}\rangle * \texttt{r::ll\_tail}\langle\texttt{tx},\texttt{n}-1\rangle \quad \textbf{inv}\ \texttt{n} \geq 1\end{array}$$

The predicate captures a list of n objects, with tx pointing to the last one. It can be coerced to a list segment, and vice versa, via the lemma:

$$\texttt{root::ll\_tail}\langle\texttt{tx},\texttt{n}\rangle \leftrightarrow \texttt{root::lseg}\langle\texttt{tx},\texttt{n}-1\rangle * \texttt{tx::node}\langle\_,\texttt{null}\rangle \qquad (2)$$

By applying this lemma, our verifier can easily prove the following specification for the concatenation of two lists with tail pointers:

$$
\begin{array}{l}
\{\mathtt{x}::\mathtt{ll\_tail}\langle \mathtt{tx}, \mathtt{n}\rangle * \mathtt{y}::\mathtt{ll\_tail}\langle \mathtt{ty}, \mathtt{m}\rangle\} \\
\mathtt{tx.next} = \mathtt{y}; \\
\{\mathtt{x}::\mathtt{ll\_tail}\langle \mathtt{ty}, \mathtt{m}+\mathtt{n}\rangle\}
\end{array}
$$

Separation logic semantics requires $\mathtt{tx}::\mathtt{node}\langle \_, \_\rangle$ to be present in the program state in order to safely perform the dereference operation via `tx.next`. Such an object can be exposed via an unfolding of the `ll_tail` predicate using the lemma, resulting in the following program state prior to the assignment:

$\{\mathtt{x}::\mathtt{lseg}\langle \mathtt{tx}, \mathtt{n}-1\rangle * \mathtt{tx}::\mathtt{node}\langle \_, \mathtt{null}\rangle * \mathtt{y}::\mathtt{ll\_tail}\langle \mathtt{ty}, \mathtt{m}\rangle\}$

which is then updated by the assignment to:

$\{\mathtt{x}::\mathtt{lseg}\langle \mathtt{tx}, \mathtt{n}-1\rangle * \mathtt{tx}::\mathtt{node}\langle \_, \mathtt{y}\rangle * \mathtt{y}::\mathtt{ll\_tail}\langle \mathtt{ty}, \mathtt{m}\rangle\}$

The weakening on the postcondition is done via an entailment, whose proof is sketched below. This proof is performed automatically by our system.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{\left(\begin{array}{l}\text{recursive entailment}\\ \text{described below}\end{array}\right)}{\begin{array}{l}\mathtt{tx}::\mathtt{node}\langle \_, \mathtt{y}\rangle * \mathtt{y}::\mathtt{ll\_tail}\langle \mathtt{ty}, \mathtt{m}\rangle \\ \vdash\ \mathtt{tx}::\mathtt{lseg}\langle \mathtt{ty}, \mathtt{m}\rangle \\ \quad *\{\mathtt{ty}::\mathtt{node}\langle \_, \mathtt{null}\rangle\}\end{array}}\,(\underline{\mathbf{FOLD}})
\qquad
\dfrac{\left(\begin{array}{l}\text{match ty with}\\ \text{residue from fold}\end{array}\right)}{\begin{array}{l}\mathtt{ty}::\mathtt{node}\langle \_, \mathtt{null}\rangle \\ \vdash\ \mathtt{ty}::\mathtt{node}\langle \_, \mathtt{null}\rangle \\ \quad *\{\mathtt{emp}\}\end{array}}
}{\begin{array}{l}\mathtt{tx}::\mathtt{node}\langle \_, \mathtt{y}\rangle * \mathtt{y}::\mathtt{ll\_tail}\langle \mathtt{ty}, \mathtt{m}\rangle \\ \vdash\ \mathtt{tx}::\mathtt{lseg}\langle \mathtt{ty}, \mathtt{m}\rangle * \mathtt{ty}::\mathtt{node}\langle \_, \mathtt{null}\rangle * \{\mathtt{emp}\}\end{array}}\,(\underline{\mathbf{FOLD}})
}{\begin{array}{l}\mathtt{x}::\mathtt{lseg}\langle \mathtt{tx}, \mathtt{n}-1\rangle * \mathtt{tx}::\mathtt{node}\langle \_, \mathtt{y}\rangle * \mathtt{y}::\mathtt{ll\_tail}\langle \mathtt{ty}, \mathtt{m}\rangle \\ \vdash\ \mathtt{x}::\mathtt{lseg}\langle \mathtt{ty}, \mathtt{m}+\mathtt{n}-1\rangle * \mathtt{ty}::\mathtt{node}\langle \_, \mathtt{null}\rangle * \{\mathtt{emp}\}\end{array}}\,(\underline{\mathbf{L{-}RIGHT}})
}{\begin{array}{l}\mathtt{x}::\mathtt{lseg}\langle \mathtt{tx}, \mathtt{n}-1\rangle * \mathtt{tx}::\mathtt{node}\langle \_, \mathtt{y}\rangle * \mathtt{y}::\mathtt{ll\_tail}\langle \mathtt{ty}, \mathtt{m}\rangle \\ \vdash\ \mathtt{x}::\mathtt{ll\_tail}\langle \mathtt{ty}, \mathtt{m}+\mathtt{n}\rangle * \{\mathtt{emp}\}\end{array}}\,(\underline{\mathbf{L{-}RIGHT}})
$$

Our entailment prover first converts the list with tail pointer in the consequent to a list segment and a node. It then breaks the list segment into two and match the first segment with the aliased segment in the antecedent. Subsequently, it performs a fold on a $\mathtt{tx}::\mathtt{lseg}\langle \mathtt{ty}, \mathtt{m}\rangle$ predicate which invokes a recursive entailment, as follows:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{(\text{ derive residue})}{\mathtt{ty}::\mathtt{node}\langle \_, \mathtt{null}\rangle \vdash \mathtt{emp} * \{\mathtt{ty}::\mathtt{node}\langle \_, \mathtt{null}\rangle\}}
}{\begin{array}{l}\mathtt{y}::\mathtt{lseg}\langle \mathtt{ty}, \mathtt{m}-1\rangle * \mathtt{ty}::\mathtt{node}\langle \_, \mathtt{null}\rangle \\ \vdash\ \mathtt{y}::\mathtt{lseg}\langle \mathtt{ty}, \mathtt{m}-1\rangle * \{\mathtt{ty}::\mathtt{node}\langle \_, \mathtt{null}\rangle\}\end{array}}\,(\underline{\mathbf{MATCH}})
}{\mathtt{y}::\mathtt{ll\_tail}\langle \mathtt{ty}, \mathtt{m}\rangle \vdash \mathtt{y}::\mathtt{lseg}\langle \mathtt{ty}, \mathtt{m}-1\rangle * \{\mathtt{ty}::\mathtt{node}\langle \_, \mathtt{null}\rangle\}}\,(\underline{\mathbf{L{-}LEFT}})
}{\begin{array}{l}\mathtt{tx}::\mathtt{node}\langle \_, \mathtt{y}\rangle * \mathtt{y}::\mathtt{ll\_tail}\langle \mathtt{ty}, \mathtt{m}\rangle \\ \vdash\ (\exists \mathtt{r} \cdot \mathtt{tx}::\mathtt{node}\langle \_, \mathtt{r}\rangle * \mathtt{r}::\mathtt{lseg}\langle \mathtt{ty}, \mathtt{m}-1\rangle) * \{\mathtt{ty}::\mathtt{node}\langle \_, \mathtt{null}\rangle\}\end{array}}\,(\underline{\mathbf{MATCH}})
}{\begin{array}{l}\mathtt{tx}::\mathtt{node}\langle \_, \mathtt{y}\rangle * \mathtt{y}::\mathtt{ll\_tail}\langle \mathtt{ty}, \mathtt{m}\rangle \\ \vdash\ \mathtt{tx}::\mathtt{lseg}\langle \mathtt{ty}, \mathtt{m}\rangle * \{\mathtt{ty}::\mathtt{node}\langle \_, \mathtt{null}\rangle\}\end{array}}\,(\underline{\mathbf{FOLD}})
$$

Such applications of lemmas are critical for automatically deriving non-trivial proofs to support program verification.

5.4 Termination

To prevent non-termination during lemma applications, we assign a history to each heap constraint $p{::}c\langle v^*\rangle$ where c is a predicate name. The history is a set of predicate names which are transitively rewritten to $p{::}c\langle v^*\rangle$. Lemma application is possible only if it does not rewrite a predicate to some predicate already in the former's history. Initially the history is empty. After each predicate application, the predicate name in the head H is added to the history of each and every predicate $p{::}c\langle v^*\rangle$ in the body ρB, in addition to the history of the matching predicate instance $p{::}c\langle v^*\rangle$. Folding and unfolding predicate instances pass the predicate history on to the predicate instances in the body.

Theorem 5.1 (Termination) *Entailment proving is terminating, even in the presence of lemma applications.*

Proof Sketch. Termination is guaranteed by the fact that only a finite number of lemma applications can occur when proving an entailment. This is the case since there is a finite number of lemmas, and each predicate instance maintains a history of predicates that are rewritten by lemma applications to the current predicate instance. Therefore lemma applications cannot occur after a finite number of steps in the entailment checking process. Termination is then guaranteed by the entailment checking as in [16].

6 Lemma Proving

Correctness of lemmas is *automatically proved* by our system via the entailment prover. A weakening lemma is proved by showing that the predicate in the head of the lemma entails the body. A strengthening lemma needs an entailment in the reverse direction. An equivalence lemma needs both. During this entailment proving, the lemma being proved can be soundly used in the proof itself as an instance of cyclic proof. Formally, proving $\rightarrow$ and $\leftrightarrow$ lemmas amount to discharging the following proof obligation:

$$\textit{unfold}(H * E \wedge G, \texttt{root}) \vdash B * \texttt{emp} \tag{3}$$

whereas $\leftarrow$ and $\leftrightarrow$ generate the following obligation:

$$\textit{unfold}(B, \texttt{root}) \vdash (H * E \wedge G) * \texttt{emp} \tag{4}$$

At the start of lemma proving, we always unfold the head predicate in the antecedent. This ensures that *infinite descent* occurs for the resulting cyclic proof which guarantees a progress condition needed for sound induction. During lemma proving, the lemma being proved may be applied to the unfolded formulas as an instance of cyclic proving. Furthermore, we also check that the entailment derives an **empty** residual heap. This ensures that both sides of the lemma cover the same heap region.

7 Implementation

We have built a prototype system using Objective Caml. The proof obligations generated by this verification system are discharged by our entailment proving procedure with the help of Omega Calculator [21] and CVC [23]. These two arithmetic solvers have complementary strengths. In many cases, CVC Lite is faster; but Omega is more complete. We therefore run both of them and get the timing of the first returning prover; or use Omega's when CVC Lite fails.

We tested our system on a suite of examples summarized in Figure 3. These examples are small but handle data structures with sophisticated shape and size properties such as sorted lists, balanced trees, etc. in a uniform way. Verification time for each function includes time to verify all functions that it calls. We compare the timings obtained with and without lemmas. Lemma proving time is not included, since they are proven once and applied many times. Preliminary results indicate that proof search with lemmas does not incur much overhead due to the directed nature of search. On the other hand, lemmas are important to verify a number of examples that would fail otherwise. For example, the bubble-sort algorithm requires sorted list to be coerced into an unsorted list expected for its precondition, whenever a swap has occurred for the bubble procedure. Also, the file manager traverses its doubly-linked lists in two directions. while circular lists are built using list segments that may require breaking and joining.

8 Related Work and Concluding Remarks

The general framework of separation logic is highly expressive but undecidable. Thus, in the search for a decidable fragment of separation logic, Berdine *et al.* [1] supports only a limited set of lemmas and predicates *without* size properties, disjunctions and existential quantifiers. This fragment forms the basis of a program verifier called Smallfoot [2]. Jia and Walker [13] also identified a decidable logic but without recursive predicates for automated reasoning of pointer programs. Preoteasa [20] showed that separation logic rules such as the frame rule are correct with respect to the predicate transformer semantics for a language with recursive procedures, local variables, value and value-result parameters via the PVS theorem prover [18]. Marti et. al. [15] verifed the heap manager of a small embedded operating system, while Feng et. al. [9] showed how the effects of interrupts and thread preemptions can be soundly modelled through ownership transfers. These approaches are based on separation logic but currently require hand-written Coq proofs. Separation logic has also been used to automatically reason about heap-manipulating programs in various contexts, e.g. locality [8], termination [3], concurrency [19]. Similar to [1], most of these works only support a limited predefined set of predicates and lemmas. Our recent work [16] allowed user-specified inductive predicates in separation logic, which are then automatically verified via a sound, terminating but incomplete verification system. Building on this prior work, the current paper proposes a new mechanism based on *user-specified lemmas* that can be *automatically proven* and *applied* by our program verifier. This feature can greatly enhance the capability of our automated program verification system, and is an important step towards building a more complete program verifier. Compared to traditional theorem provers, like Isabelle [17], our approach attained the following improvements: 1) it

Programs	LOC	Timing	
		with lemmas	without lemmas
List with Tail		verifies size/length	
append	1	0.18	*failed*
Circular Linked List		verifies size + circularity	
delete_first	15	0.07	0.04
count	15	0.13	*failed*
Doubly Linked Circular List		verifies size + double links + circularity	
delete	12	0.26	*failed*
Doubly Linked List		verifies size + double links	
append	26	0.16	0.12
flatten (from tree)	34	0.35	0.33
Sorted List		verifies size + min + max + sortedness	
delete	21	0.16	0.15
insertion_sort	36	0.37	0.32
selection_sort	52	0.34	0.31
bubble_sort	42	0.64	*failed*
merge_sort	105	0.61	0.56
quick_sort	85	0.67	0.65
File Manager		verifies directory structure	
search_name	18	1.71	1.49
mkdir	43	3.02	*failed*
remove	50	4.66	*failed*
copy_folder	67	7.50	*failed*
AVL Tree		verifies size + height + height-balanced	
insert	169	5.06	5.00
Red-Black Tree		verifies size + black-height + height-balanced	
insert	167	1.53	1.39

Fig. 3. Verification Times (in seconds) for Data Structures with Arithmetic Constraints

is based on separation logic (not classical logic), 2) it is automatically proven (via cyclic proof), 3) it is automatically applied (during entailment), and 4) it always terminates. In contrast, traditional theorem prover handles lemmas (for classical logic) using either user-specifiable tactics/heuristics or requires manual proofs, and is not guaranteed to terminate.

On the inference front, Lee et al. [14] has formalized 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 properties. Another work [10] has also formulated interprocedural shape inference but is restricted to just the list segment shape predicate. More recently, Guo et al [12] showed how fairly complex shapes can be inferred with the help of a technique based on *truncation point* which can be viewed as a lemma for cutting (or grafting) a subheap of the same predicate from (or into) a given shape. However, the presence of numerical properties makes the truncation point technique difficult for more general user-defined predicates. The reason is that, after cutting a subheap and then grafting back a piece of heap of the same shape, the shape of the original heap is restored, but not necessarily its content or other quantitative properties. Another recent work by Chang and Rival [6] proposes a backward unfolding technique that requires an in-built (but generic) lemma for splitting inductive segments. This hardwired use of a lemma can be viewed as a special case to our user-defined approach. While our system does not focus on the inference aspect, we provide better support for automated verification via an expressive data structures and lemmas specification mechanism. For example, data structures with strong invariants, such as balanced heights, sortedness and graph-like pointer links, are easily captured by our specification mechanism prior to automatic verification.

To the best of the authors' knowledge, most past works in automated program verification have not made systematic provision for *user-specified lemmas* that can be automatically proven and applied, so as to widen the class of programs that can be automatically verified. However, the use of user-specified lemmas can be found in works based on dependent type systems and proof checkers. An example of this is the Applied Type System (ATS) [7] that was proposed for combining programs with proofs. In ATS, dependent types for capturing program invariants and lemmas are highly expressive, but users must supply all expected properties, associated proofs, and precisely state where they are to be applied, with ATS playing the role of a proof-checker. On the contrary, our proposed technique performs lemma proving and program verification automatically, without the need for such detailed guidance.

To summarise, we have introduced a new mechanism to support user-supplied lemmas for automated program verification via separation logic. This approach is *directed* and is guaranteed to *terminate*. It is *directed* because the lemmas are applied selectively, as guided by the need for the eventual matching up of heap predicates during entailment proving. It terminates since we use well-formed and well-founded heap formulae for both shape predicates and lemmas, together with a cycle detection technique. One strength of our approach is that users are allowed to add relevant lemmas to further enhance the capability of the automated program verification system. This puts creative control back into users' hands. Nevertheless, we provide machine support for automatically proving and then applying these given lemmas. With the appropriate use of universal quantifiers, these lemmas can be quite expressive. We believe that lemmas can greatly enhance the capability of automated program verification in general, and separation logic in particular; as they play the role of cut rules in proof systems.

Acknowledgments : Our thanks to Shengchao Qin for providing insights on how lemmas should be represented, and to Cristina David for bravely carrying the mantle for our SLEEK prover sub-system. Anonymous reviewers of CAV 2008 provided critical but fair comments that have helped us improve the presentation of this paper. This work is supported by an A*STAR-funded research project R-252-000-233-305 on "A Constructive Framework for Dependable Software".

References

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

2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, Springer LNCS, 2006.
3. J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, Springer-Verlag LNCS, 2006.
4. James Brotherston. Formalised inductive reasoning in the logic of bunched implications. In *SAS*, Springer-Verlag LNCS, 2007.
5. James Brotherston and Alex Simpson. Complete sequent calculi for induction and infinite descent. In *LICS*, pages 51–62, 2007.
6. Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.
7. C. Chen and H. Xi. Combining Programming with Theorem Proving. In *ICFP*, Tallinn, Estonia, September 2005.
8. D. Distefano, P. W. O'Hearn, and H. Yang. A Local Shape Analysis based on Separation Logic. In *TACAS*, LNCS. Springer-Verlag, March 2006.
9. X. Feng, Z. Shao, Y. Dong, and Y Guo. Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads. In *PLDI*, Tucson, Arizona, June 2008. ACM Press.
10. A. Gotsman, J. Berdine, and B. Cook. Interprocedural Shape Analysis with Separated Heap Abstractions. In *SAS*, Springer-Verlag LNCS, Seoul, Korea, August 2006.
11. Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *PLDI*, pages 266–277, 2007.
12. Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *PLDI*, pages 256–265, 2007.
13. L. Jia and D. Walker. ILC: A foundation for automated reasoning about pointer programs. In *ESOP*, March 2006.
14. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*. Springer Verlag, April 2005.
15. N. Marti, R Affeldt, and A Yonezawa. Formal Verification of the Heap Manager of an Operating system using Separation Logic. In *ICFEM*, Macao, China, November 2006. Springer-Verlag.
16. 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.
17. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
18. Sam Owre, John M. Rushby, Natarajan Shankar, and David W. J. Stringer-Calvert. PVS: An experience report. In *FM-Trends*, pages 338–345, 1998.
19. M Parkinson, R Bornat, and P O'Hearn. Modular verification of a non-blocking stack. In *POPL*, Nice, France, January 2007.
20. Viorel Preoteasa. Mechanical verification of recursive procedures manipulating pointers using separation logic. In *FM*, pages 508–523, 2006.
21. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *CACM*, 8:102–114, 1992.
22. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, Copenhagen, Denmark, July 2002.
23. Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *LNCS*, pages 500–504. Springer-Verlag, July 2002. Copenhagen, Denmark.