Automated Verification of Shape and Size Properties via Separation Logic

Huu Hai Nguyen[1], Cristina David[2], Shengchao Qin[3], and Wei-Ngan Chin[1,2]

[1] Computer Science Programme, Singapore-MIT Alliance
[2] Department of Computer Science, National University of Singapore
[3] Department of Computer Science, Durham University
{nguyenh2,davidcri,chinwn}@comp.nus.edu.sg shengchao.qin@durham.ac.uk

Abstract. Despite their popularity and importance, pointer-based programs remain a major challenge for program verification. In this paper, we propose an automated verification system that is concise, precise and expressive for ensuring the safety of pointer-based programs. Our approach uses *user-definable* shape predicates to allow programmers to describe a wide range of data structures with their associated size properties. To support automatic verification, we design a new entailment checking procedure that can handle *well-founded* inductive predicates using *unfold/fold* reasoning. We have proven the soundness and termination of our verification system, and have built a prototype system.

1 Introduction

In recent years, separation logic has emerged as a contender for formal reasoning of heap-manipulating imperative programs. While the foundations of separation logic have been laid in seminal papers by Reynolds [17] and Isthiaq and O'Hearn [10], new automated reasoning tools based on separation formulae, such as [2, 8], are beginning to appear. Several major challenges are faced by the designers of such reasoning systems, including key issues on *automation* and *expressivity*. This paper's main goal is to raise the level of expressivity and verifiability that is possible with an automated verification system based on separation logic. We make the following technical contributions towards this overall goal :

– We provide a *shape predicate specification* mechanism that can capture a wide range of data structures together with size properties, such as various height-balanced trees, priority heap, sorted list, etc. We provide a mechanism to soundly approximate each shape predicate by a heap-independent *invariant* which plays an important role in entailment checking (Secs 2 and 4.1).
– We design a new procedure to check entailment of separation heap constraints. This procedure uses *unfold/fold* reasoning to deal with shape definitions. While the unfold/fold mechanism is not new, we have identified sufficient conditions for soundness and termination of automatic unfold/fold reasoning to support entailment checking, in the presence of user-defined shape predicates that may be recursive. (Secs 3.1, 4 and 5)
– We have implemented a prototype verification system with the above features and have also proven both its soundness and termination (Secs 6 and 7).

2 User-Definable Shape Predicates

Separation logic [17, 10] extends Hoare logic to support reasoning about shared mutable data structures. It adds two more connectives to classical logic : separating conjunction $*$, and separating implication $-\!\!*$. $h_1 * h_2$ asserts that two heaps described by h_1 and h_2 are domain-disjoint. $h_1 -\!\!* h_2$ asserts that if the current heap is extended with a disjoint heap described by h_1, then h_2 holds in the extended heap. In this paper we use only separating conjunction.

We propose an intuitive mechanism based on inductive predicates (or relations) to allow user specification of shapely data structures with size properties. Our shape specification is based on separation logic with support for disjunctive heap states. Furthermore, each shape predicate may have pointer or integer parameters to capture relevant properties of data structures. We use the following data node declarations for the examples in the paper. They are recursive data declarations with different number of fields.

```
data node { int val; node next }
data node2 { int val; node2 prev; node2 next }
data node3 { int val; node3 left; node3 right; node3 parent }
```

We use $\texttt{p}::\texttt{c}\langle\texttt{v}^*\rangle$ to denote two things in our system. When $\texttt{c}$ is a data name, $\texttt{p}::\texttt{c}\langle\texttt{v}^*\rangle$ stands for singleton heap $\texttt{p}\mapsto[(\texttt{f} : \texttt{v})]^*$ where $\texttt{f}^*$ are fields of data declaration $\texttt{c}$. When $\texttt{c}$ is a predicate name, $\texttt{p}::\texttt{c}\langle\texttt{v}^*\rangle$ stands for the formula $\texttt{c}(\texttt{p}, \texttt{v}^*)$. The reason we distinguish the first parameter from the rest is that each predicate has an implicit parameter `self` as the first one. Effectively, `self` is a "root" pointer to the specified data structure that guides data traversal and facilitates the definition of *well-founded* predicates (Sec 3.1). As an example, a singly linked list with length `n` is described by :

$$\texttt{ll}\langle\texttt{n}\rangle\equiv(\texttt{self}{=}\texttt{null}\wedge\texttt{n}{=}0)\vee(\exists\texttt{i},\texttt{m},\texttt{q}\cdot\texttt{self}::\texttt{node}\langle\texttt{i},\texttt{q}\rangle*\texttt{q}::\texttt{ll}\langle\texttt{m}\rangle\wedge\texttt{n}{=}\texttt{m}{+}1)\ \textbf{inv}\ \texttt{n}{\geq}0$$

The second parameter `n` captures a *derived value* that is computed rather than taken directly from the heap state. The above definition asserts that an `ll` list can be empty (the base case $\texttt{self}{=}\texttt{null}$) or consists of a head data node (specified by $\texttt{self}::\texttt{node}\langle\texttt{i},\texttt{q}\rangle$) and a separate tail data structure which is also an `ll` list ($\texttt{q}::\texttt{ll}\langle\texttt{m}\rangle$). The $*$ connector ensures that the head node and the tail reside in disjoint heaps. We also specify a default invariant $\texttt{n}{\geq}0$ that holds for all `ll` lists. Our predicate uses existential quantifiers for local values and pointers, such as $\texttt{i},\texttt{m},\texttt{q}$.

A more complex shape, doubly linked-list with length `n`, is described by :

$$\texttt{dll}\langle\texttt{p},\texttt{n}\rangle\equiv(\texttt{self}{=}\texttt{null}\wedge\texttt{n}{=}0)\vee(\texttt{self}::\texttt{node2}\langle\_,\texttt{p},\texttt{q}\rangle*\texttt{q}::\texttt{dll}\langle\texttt{self},\texttt{n}{-}1\rangle)\ \textbf{inv}\ \texttt{n}{\geq}0$$

The `dll` shape predicate has a parameter `p` that represents the `prev` field of the first node of the doubly linked-list. It captures a chain of nodes that are to be traversed via the `next` field starting from the current node `self`. The nodes accessible via the `prev` field of the `self` node are not part of the `dll` list. This example also highlights some shortcuts we may use to make shape specification easier. We use underscore _ to denote an anonymous variable. Non-parameter variables in the RHS of the shape definition, such as q, are considered existentially quantified. Furthermore, terms may be directly written as arguments of shape predicate or data node.

User-definable shape predicates provide us with more flexibility than some recent automated reasoning systems [1, 3] that are designed to work with only a small set of fixed predicates. Furthermore, our shape predicates can describe not only the *shape* of data structures, but also their *size* properties. This capability enables many applications, especially to support data structures with sophisticated invariants. For example, we may define a non-empty sorted list as below. The predicate also tracks the length, as well as the minimum and maximum elements of the list.

$$\texttt{sortl}\langle\texttt{n},\texttt{min},\texttt{max}\rangle \equiv (\texttt{self}{::}\texttt{node}\langle\texttt{min},\texttt{null}\rangle \wedge \texttt{min}{=}\texttt{max} \wedge \texttt{n}{=}1)$$
$$\vee\ (\texttt{self}{::}\texttt{node}\langle\texttt{min},\texttt{q}\rangle * \texttt{q}{::}\texttt{sortl}\langle\texttt{n}{-}1,\texttt{k},\texttt{max}\rangle \wedge \texttt{min}{\leq}\texttt{k})\ \textbf{inv}\ \texttt{min}{\leq}\texttt{max} \wedge \texttt{n}{\geq}1$$

The constraint $\texttt{min}{\leq}\texttt{k}$ guarantees that sortedness property is adhered between any two adjacent nodes in the list. We may now specify (and then verify) the following insertion sort algorithm :

```
node insert(node x, node vn) where
  x::sortl⟨n, sm, lg⟩ ∗ vn::node⟨v, _⟩ ∗→ res::sortl⟨n+1, min(v, sm), max(v, lg)⟩
{ if (vn.val≤x.val) then { vn.next:=x; vn }
  else if (x.next=null) then { x.next:=vn; vn.next:=null; x }
  else { x.next:=insert(x.next, vn); x }}

node insertion_sort(node y) where y::ll⟨n⟩ ∧ n>0 ∗→ res::sortl⟨n, _, _⟩
{ if (y.next=null) then y
   else { y.next:=insertion_sort(y.next); insert(y.next, y) }}
```

We use the notation $\Phi_{pr} \ast\!\!\to \Phi_{po}$ to capture a precondition Φ_{pr} and a postcondition Φ_{po} of a method. We also use an expression-oriented language where the last subexpression (e.g. $\texttt{e}_2$ from $\texttt{e}_1$;$\texttt{e}_2$) denotes the result of an expression. A special identifier res is also used in the postcondition to denote the result of a method. The postcondition of insertion_sort shows that the output list is sorted and has the same number of nodes as the input list.

3 Automated Verification

In this section, we first introduce a core object-based imperative language and then propose a set of forward verification rules to systematically check that preconditions are satisfied at call sites, and that the declared postcondition is successfully verified (assuming the precondition) for each method definition.

3.1 Language

We provide a simple imperative language in Figure 1. Our language is strongly typed and we assume programs and constraints are well-typed. The language supports data type declaration via *datat*, and shape predicate definition via *spred*. For each shape definition, we also declare a heap-independent invariant π_0 over the parameters $\{\texttt{self}, v^*\}$ that is valid for each instance of the predicate.

$$
\begin{array}{lll}
P & ::= tdecl^*\ meth^* & tdecl ::= datat \mid spred \\
datat & ::= \texttt{data}\ c\ \{\ field^*\ \} & field ::= t\ v \qquad t ::= c \mid \tau \\
\tau & ::= \texttt{int} \mid \texttt{bool} \mid \texttt{float} \mid \texttt{void} & \\
spred & ::= c\langle v^*\rangle \equiv \Phi\ \texttt{inv}\ \ \pi_0 & \\
meth & ::= t\ mn\ ((t\ v)^*)\ \texttt{where}\ \Phi_{pr} \ast\!\!\!\rightarrow \Phi_{po}\ \{e\} & \\
e & ::= \texttt{null} \mid k^{\tau} \mid v \mid v.f \mid v{:=}e \mid v_1.f{:=}v_2 \mid \texttt{new}\ c(v^*) & \\
 & \quad \mid e_1; e_2 \mid t\ v;\ e \mid mn(v^*) \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \\
 & \quad \mid \texttt{while}\ v\ \texttt{where}\ \Phi_{pr} \ast\!\!\!\rightarrow \Phi_{po}\ \texttt{do}\ e & \\
\Phi & ::= \bigvee(\exists v^* {\cdot} \kappa \wedge \pi)^* & \pi ::= \gamma \wedge \phi \\
\gamma & ::= v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \gamma_1 \wedge \gamma_2 & \\
\kappa & ::= \texttt{emp} \mid v{::}c\langle v^*\rangle \mid \kappa_1 * \kappa_2 & \\
\Delta & ::= \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v {\cdot} \Delta & \\
\phi & ::= b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi & \\
b & ::= \texttt{true} \mid \texttt{false} \mid v \mid b_1 = b_2 & a ::= s_1{=}s_2 \mid s_1{\leq}s_2 \\
s & ::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}} {\times} s \mid s_1 {+} s_2 \mid -s \mid max(s_1,s_2) \mid min(s_1,s_2) &
\end{array}
$$

Fig. 1. A Core Imperative Language

Each method *meth* and `while` loop is declared with pre- and post-conditions of the form $\Phi_{pr} \ast\!\!\!\rightarrow \Phi_{po}$. For simplicity, we assume that variable names declared in each method are all distinct and that pass-by-value parameter mechanism is used. Primed notation is used to capture the latest value of local variables and may appear in the postcondition of loops. For example, a simple loop with pre/post conditions is shown below :

$\texttt{while x<0 where true} \ast\!\!\!\rightarrow (\texttt{x}{>}0 \wedge \texttt{x}'{=}\texttt{x}) \vee (\texttt{x}{\leq}0 \wedge \texttt{x}'{=}0)\ \texttt{do}\ \{\ \texttt{x}{:=}\texttt{x}{+}1\ \}$

Here `x` and $\texttt{x}'$ denote the old and new values of variable `x` at the entry and exit of the loop, respectively.

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 presently restricted to pointer equality/inequality γ and Presburger arithmetic ϕ. Furthermore, Δ denotes a composite formula that could always be normalised into the Φ form (see Figure 3). The semantic model for the separation constraints is left in the appendix due to page limit.

Separation constraints are used in pre/post conditions and shape definitions. In order to handle them correctly without running into unmatched residual heap nodes, we require each separation constraint to be well-formed, as follows :

Definition 3.1 (Well-Formed Constraint) *A separation constraint Φ is well-formed if (i) every data node and shape predicate are reachable from their* accessible *variables, (ii) it is in a disjunctive normal form* $\bigvee(\exists v^*{\cdot}\kappa{\wedge}\gamma{\wedge}\phi)^*$ *where* κ *is for heap nodes,* γ *is for pointer constraint, and* ϕ *is for arithmetic formula.*

Definition 3.2 (Accessible) *A variable is said to be* accessible *w.r.t. a shape predicate if it is a parameter or it is a special variable, either* `self` *or* `res`*.*

Definition 3.3 (Reachable) *Given a heap constraint* $\kappa = \texttt{p::c}\langle\texttt{v}^*\rangle * \kappa_1$*, node* $\texttt{p::c}\langle\texttt{v}^*\rangle$ *is* reachable *from a variable* `q` *if and only if the following relation holds:*

$$\texttt{reach}(\kappa, \texttt{q}, \texttt{p::c}\langle\texttt{v}^*\rangle) =_{df} (\texttt{p}{=}\texttt{q}){\vee}(\kappa_1{=}\texttt{q::c}_\texttt{q}\langle..,\texttt{r},..\rangle{*}\kappa_2 \wedge \texttt{reach}(\kappa_2, \texttt{r}, \texttt{p::c}\langle\texttt{v}^*\rangle))$$

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 checking procedure to correctly match up nodes from the consequent with nodes from the antecedent of an entailment relation.

Arbitrary recursive shape relation can lead to non-termination in unfold/fold reasoning. To avoid that problem, we propose to use only *well-founded* shape predicates in our framework.

Definition 3.4 (Well-Founded Predicate) *A shape predicate is said to be* well-founded *if it satisfies four conditions, namely: (i) it is a well-formed constraint, (ii) the parameter* `self` *may only be bound to a data node and not a predicate, (iii) only* `self` *is allowed to be bound to a data node and (iv) every predicate is reachable from* `self`*.*

Note that the definitions above are syntactic and can easily be enforced. Two examples of well-founded shape predicates are `treep` – binary tree with parent pointer, and `avl` – binary tree with near balanced heights, as follows :

$$\begin{array}{l}
\texttt{treep}\langle\texttt{p}\rangle \equiv (\texttt{self}{=}\texttt{null}) \vee\ (\texttt{self::node3}\langle_,\texttt{l},\texttt{r},\texttt{p}\rangle * \texttt{l::treep}\langle\texttt{self}\rangle \\
\quad {*}\texttt{r::treep}\langle\texttt{self}\rangle)\ \ \textbf{inv}\ \ \texttt{true} \\
\texttt{avl}\langle\texttt{n},\texttt{h}\rangle \equiv (\texttt{self}{=}\texttt{null} \wedge \texttt{n}{=}0 \wedge \texttt{h}{=}0) \vee\ (\texttt{self::node2}\langle_,\texttt{p},\texttt{q}\rangle * \texttt{p::avl}\langle\texttt{n}_1,\texttt{h}_1\rangle \\
\quad {*}\texttt{q::avl}\langle\texttt{n}_2,\texttt{h}_2\rangle \wedge \texttt{n}{=}\texttt{n}_1{+}\texttt{n}_2{\wedge}\ \texttt{h}{=}1{+}\texttt{max}(\texttt{h}_1,\texttt{h}_2) \wedge -1{\leq}\texttt{h}_1{-}\texttt{h}_2{\leq}1)\ \ \textbf{inv}\ \ \texttt{n},\texttt{h}{\geq}0
\end{array}$$

In contrast, the following three shape definitions are <u>not</u> well-founded.

$$\begin{array}{ll}
\texttt{foo}\langle\texttt{n}\rangle & \equiv \texttt{self::foo}\langle\texttt{m}\rangle \wedge \texttt{n}{=}\texttt{m}{+}1 \\
\texttt{goo}\langle\rangle & \equiv \texttt{self::node}\langle_,_\rangle * \texttt{q::goo}\langle\rangle \\
\texttt{too}\langle\rangle & \equiv \texttt{self::node}\langle_,\texttt{q}\rangle * \texttt{q::node}\langle_,_\rangle
\end{array}$$

For `foo`, the `self` identifier is bound to a shape predicate. For `goo`, the heap node pointed by `q` is *not* reachable from variable `self`. For `too`, an extra data node is bound to a non-`self` variable. The first example may cause infinite unfolding, while the second example captures an unreachable (junk) heap that cannot be located by our entailment procedure. The last example is just a syntactic restriction to facilitate termination proof reasoning, and can be easily overcome by introducing intermediate predicates.

$$\frac{\textbf{[FV-PRED]} \quad XPure_0(\Phi) \Longrightarrow [0/\texttt{null}](\pi_{inv})}{\vdash c\langle v^*\rangle = \Phi \ \texttt{inv}\ \pi_{inv}}$$

$$\frac{\begin{array}{c}\textbf{[FV-CALL]}\\ t\ mn((t_i\ v_i)_{i=1}^{n})\ \texttt{where}\ \Phi_{pr} \ast\!\!\!\longrightarrow \Phi_{po}\ \{..\} \quad \rho{=}[v_i'/v_i] \\ \Delta{\vdash}\rho\Phi_{pr} * \Delta_1 \quad W = \{v_1,..,v_n\} \quad \Delta_2{=}(\Delta_1 *_W \Phi_{po})\end{array}}{\vdash \{\Delta\}\, m(v_1..v_n)\, \{\Delta_2\}}$$

$$\frac{\begin{array}{c}\textbf{[FV-METH]}\\ V{=}\{v_1..v_n\} \quad W{=}prime(V) \quad \Delta{=}\Phi_{pr}{\wedge}nochange(V) \quad \vdash \{\Delta\}\, e\, \{\Delta_1\} \quad (\exists W{\cdot}\Delta_1){\vdash}\Phi_{po} * \Delta_2\end{array}}{\vdash t_0\ mn(t_1\ v_1,..,t_n\ v_n)\ \texttt{where}\ \Phi_{pr} \ast\!\!\!\longrightarrow \Phi_{po}\ \{e\}}$$

Fig. 2. Some Forward Verification Rules

3.2 Forward Verification

With pre/post conditions declared for each method, we can now apply modular verification to its body using Hoare-style triples $\vdash \{\Delta_1\}\, e\, \{\Delta_2\}$. These are *forward verification* rules as we expect Δ_1 to be given before computing Δ_2. Three rules are given in Fig 2 while others are left in the appendix (Fig 7). They are used to track heap states as accurately as possible with path-, flow-, and context-sensitivity. For each call site, [FV−CALL] ensures that its method's precondition is satisfied. At each method definition, [FV−METH] checks that its postcondition holds for the method body assuming its precondition. At each shape definition, [FV−PRED] checks that its given invariant is a consequence of the well-founded heap formula. The soundness of the forward verification is also left in the appendix due to page limit.

Our rules currently allow both preconditions and shape predicates to contain contradictory/`false` heap state. A `false` precondition implies anything but can never be satisfied by any call site from a non-contradictory heap state. Similarly, a shape predicate with a `false` heap formula can never be constructed.

We now explain the operators/functions used in our verification rules. The operators $\wedge_{\{v\}}$ (in assignment rule) and $*_W$ (in method call 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 :

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

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 $*$. Function $nochange(V)$ returns a formula asserting that the unprimed and primed versions of each variable in V are equal; $prime(V)$ returns the primed form of all variables in V. We use $[e^*/v^*]$ to represent substitutions of v^* by e^*. A special case is $[0/\texttt{null}]$, which denotes replacement of `null` by 0. Normalization rules for separation constraints are given in Figure 3. *XPure* is described in the next section.

$$
\begin{array}{llll}
(\Delta_1 \vee \Delta_2) \wedge \pi & \leadsto (\Delta_1 \wedge \pi) \vee (\Delta_2 \wedge \pi) & (\gamma_1 \wedge \phi_1) \wedge (\gamma_2 \wedge \phi_2) & \leadsto (\gamma_1 \wedge \gamma_2) \wedge (\phi_1 \wedge \phi_2) \\
(\Delta_1 \vee \Delta_2) * \Delta & \leadsto (\Delta_1 * \Delta) \vee (\Delta_2 * \Delta) & (\exists x \cdot \Delta) \wedge \pi & \leadsto \exists y \cdot ([y/x]\Delta \wedge \pi) \\
(\kappa_1 \wedge \pi_1) * (\kappa_2 \wedge \pi_2) & \leadsto (\kappa_1 * \kappa_2) \wedge (\pi_1 \wedge \pi_2) & (\exists x \cdot \Delta_1) * \Delta_2 & \leadsto \exists y \cdot ([y/x]\Delta_1 * \Delta_2) \\
(\kappa_1 \wedge \pi_1) \wedge (\pi_2) & \leadsto \kappa_1 \wedge (\pi_1 \wedge \pi_2) & &
\end{array}
$$

Fig. 3. Normalization Rules

4 Entailment

We present in this section the entailment checking rules for the class of constraints used by our verification system.

4.1 Separation Constraint Approximation

$$
\frac{(c\langle v^*\rangle \equiv \Phi \texttt{ inv } \pi_0) \in P}{\mathit{Inv}_0(p::c\langle v^*\rangle) = [p/\texttt{self}, 0/\texttt{null}]\pi_0}
$$

$$
\frac{(c\langle v^*\rangle \equiv \Phi \texttt{ inv } \pi_0) \in P}{\mathit{Inv}_n(p::c\langle v^*\rangle) = [p/\texttt{self}, 0/\texttt{null}]\mathit{XPure}_{n-1}(\Phi)}
$$

$$
\mathit{XPure}_n(\bigvee(\exists v^* \cdot \kappa \wedge \pi)^*) =_{df} \bigvee(\exists v^* \cdot \mathit{XPure}_n(\kappa) \wedge [0/\texttt{null}]\pi)^*
$$

$$
\mathit{XPure}_n(\texttt{emp}) =_{df} \texttt{true}
$$

$$
\frac{\mathit{IsData}(c) \quad \mathit{fresh}\ i}{\mathit{XPure}_n(p::c\langle v^*\rangle) =_{df} \textbf{ex } i \cdot (p{=}i \wedge i{>}0)}
$$

$$
\frac{\mathit{IsPred}(c) \quad \mathit{fresh}\ i^* \quad \mathit{Inv}_n(p::c\langle v^*\rangle) = \textbf{ex } j^* \cdot \bigvee(\exists u^* \cdot \pi)^*}{\mathit{XPure}_n(p::c\langle v^*\rangle) =_{df} \textbf{ex } i^* \cdot [i^*/j^*] \bigvee(\exists u^* \cdot \pi)^*}
$$

$$
\mathit{XPure}_n(\kappa_1 * \kappa_2) =_{df} \mathit{XPure}_n(\kappa_1) \wedge \mathit{XPure}_n(\kappa_2)
$$

Fig. 4. *XPure* : Translating to Pure Form

Entailment between separation formulae (detailed in section 4.2) is reduced to entailment between pure formulae by successively removing heap nodes from the consequent until only a pure formula remains. When the consequent is pure, the heap formula in the antecedent is soundly approximated by a pure formula via function XPure_n. The function $\mathit{XPure}_n(\Phi)$, whose definition is given in Fig 4, returns a sound approximation of Φ as formula $\textbf{ex } i^* \cdot \bigvee(\exists v^* \cdot \pi)^*$ where i^* are (non-null) distinct symbolic addresses of heap nodes of Φ. The function $\mathit{IsData}(c)$ returns `true` if c is a data node, while $\mathit{IsPred}(c)$ returns `true` if c is a shape predicate.

We illustrate how this function works by the following example :

$$
\begin{array}{l}
\mathit{XPure}_n(\texttt{p}_1::\texttt{node}\langle \_, \_\rangle * \texttt{p}_2::\texttt{node}\langle \_, \_\rangle) \\
\quad = (\textbf{ex } \texttt{i}_1 \cdot (\texttt{p}_1{=}\texttt{i}_1 \wedge \texttt{i}_1{>}0)) \wedge (\textbf{ex } \texttt{i}_2 \cdot (\texttt{p}_2{=}\texttt{i}_2 \wedge \texttt{i}_2{>}0)) \\
\quad = \textbf{ex } \texttt{i}_1, \texttt{i}_2 \cdot (\texttt{p}_1{=}\texttt{i}_1 \wedge \texttt{i}_1{>}0 \wedge \texttt{p}_2{=}\texttt{i}_2 \wedge \texttt{i}_2{>}0 \wedge \texttt{i}_1{\neq}\texttt{i}_2)
\end{array}
$$

The following normalization rules are also used :

$$
\begin{array}{ll}
(\textbf{ex } I \cdot \phi_1) \vee (\textbf{ex } J \cdot \phi_2) & \leadsto \textbf{ex } I{\cup}J \cdot (\phi_1 \vee \phi_2) \\
\exists\ v \cdot (\textbf{ex } I \cdot \phi) & \leadsto \textbf{ex } I \cdot (\exists\ v \cdot \phi) \\
(\textbf{ex } I \cdot \phi_1) \wedge (\textbf{ex } J \cdot \phi_2) & \leadsto \textbf{ex } I{\cup}J \cdot \phi_1 \wedge \phi_2 \wedge \bigwedge_{i \in I, j \in J} i{\neq}j
\end{array}
$$

The **ex** i^* construct is converted to $\exists\, i^*$ when the formula is used as a pure formula. The soundness of $XPure_n$ is formalized by :

Lemma 4.1 (Sound Abstraction). *Given a separation constraint Φ where the invariants of the shape predicates appearing in Φ are semantic consequences of their respective predicate definitions, we have :* $\Phi \models XPure_n(\Phi)$
Proof *: By structural induction on Φ.*

Lemma 4.2 (Sound Invariant). *Given a shape predicate $c\langle v^*\rangle{\equiv}\Phi$, we have $\Phi \models Inv_n(\texttt{self}{::}c\langle v^*\rangle)$ if $XPure_n(\Phi) \Longrightarrow Inv_n(\texttt{self}{::}c\langle v^*\rangle)$.*
Proof: *By structural induction on Φ.*

These lemmas ensure the soundness of the entailment checking rule [**ENT−EMP**] (Fig. 5) and the forward verification rule [**FV−PRED**] (Fig. 2). Lemma 4.1 asserts that it is safe to approximate an antecedent by using *XPure* if all the predicate invariants are sound. Lemma 4.2 ensures that a supplied invariant that passes the [**FV−PRED**] check is a semantic consequence of the predicate. They also allow the possibility of obtaining a more precise invariant by applying *XPure* one or more times. For example, when given a pure invariant $\texttt{n}{\geq}\texttt{0}$ for the predicate $\texttt{ll}\langle\texttt{n}\rangle$, a single application returns $\textbf{ex}\ \texttt{i}{\cdot}(\texttt{self}{=}\texttt{0}{\wedge}\texttt{n}{=}\texttt{0} \vee \texttt{self}{=}\texttt{i}{\wedge}\texttt{i}{>}\texttt{0}{\wedge}\texttt{n}{>}\texttt{0})$ which is sound and more precise, as it relates the nullness of the `self` pointer with the size `n` of the list.

The invariants associated with shape predicates play an important role in our system. Without the knowledge $\texttt{m}{\geq}\texttt{0}$, the entailment $\texttt{x}{::}\texttt{node}\langle\_,\texttt{y}\rangle{*}\texttt{y}{::}\texttt{ll}\langle\texttt{m}\rangle \vdash \texttt{x}{::}\texttt{ll}\langle\texttt{n}\rangle \wedge \texttt{n}{\geq}\texttt{1}$ would not have succeeded due to $\texttt{n}{\geq}\texttt{1}$. Without the more precise derived invariant using *XPure* for predicate `ll`, the entailment $\texttt{x}{::}\texttt{ll}\langle\texttt{n}\rangle \wedge \texttt{n}{>}\texttt{0} \vdash \texttt{x}{\neq}\texttt{null}$ would not have succeeded either.

4.2 Separation Constraint Entailment

We express the main procedure for heap entailment by the relation

$$\Delta_A {\vdash}^{\kappa}_{V} \Delta_C * \Delta_R$$

which denotes

$$\kappa * \Delta_A {\vdash} \exists V{\cdot}(\kappa * \Delta_C) * \Delta_R$$

The purpose of heap entailment 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 residual heap state Δ_R. 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 k and V are derived. The entailment checking procedure is invoked with $\kappa = \texttt{emp}$ and $V = \emptyset$. The entailment checking rules are given in Fig 5. We discuss the matching rule in what follows, and leave unfold/fold rules to Sec 5.

The procedure works by successively matching up heap nodes that can be proven aliased. As the matching process is incremental, we keep the successfully

$$\frac{\begin{array}{c}[\textbf{ENT-EMP}]\\ \rho{=}[0/\texttt{null}]\\ \rho(\mathit{XPure}_n(\kappa_1{*}\kappa){\wedge}\pi_1){\Longrightarrow}\rho\exists V{\cdot}\pi_2\end{array}}{\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}\pi_2 * (\kappa_1{\wedge}\pi_1)}
\qquad
\frac{\begin{array}{c}[\textbf{ENT-MATCH}]\\ \mathit{XPure}_n(p_1{::}c\langle v_1^*\rangle{*}\kappa_1{*}\pi_1){\Longrightarrow}p_1{=}p_2 \quad \rho{=}[v_1^*/v_2^*]\\ \kappa_1{\wedge}\pi_1{\wedge}\mathit{freeEqn}(\rho,V){\vdash}^{\kappa{*}p_1::c\langle v_1^*\rangle}_{V-\{v_2^*\}}\rho(\kappa_2{\wedge}\pi_2)*\Delta\end{array}}{p_1{::}c\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}(p_2{::}c\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2)*\Delta}$$

$$\frac{\begin{array}{c}[\textbf{ENT-FOLD}]\\ \mathit{IsPred}(\mathsf{c}_2){\wedge}\mathit{IsData}(\mathsf{c}_1) \quad (\Delta^r,\kappa^r,\pi^r){\in}\mathit{fold}^{\kappa}(p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1, p_2{::}c_2\langle v_2^*\rangle)\\ \mathit{XPure}_n(p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{*}\pi_1){\Longrightarrow}p_1{=}p_2 \quad (\pi^a,\pi^c){=}\mathit{split}^{\{v_2^*\}}_{V}(\pi^r) \quad \Delta^r{\wedge}\pi^a{\vdash}^{\kappa^r}_{V}(\kappa_2{\wedge}\pi_2{\wedge}\pi^c)*\Delta\end{array}}{p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}(p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2)*\Delta}$$

$$\frac{\begin{array}{c}[\textbf{ENT-UNFOLD}]\\ \mathit{XPure}_n(p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{*}\pi_1){\Longrightarrow}p_1{=}p_2 \quad \mathit{IsPred}(\mathsf{c}_1){\wedge}\mathit{IsData}(\mathsf{c}_2)\\ \mathit{unfold}(p_1{::}c_1\langle v_1^*\rangle){*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}(p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2)*\Delta\end{array}}{p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}(p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2)*\Delta}
\qquad
\frac{\begin{array}{c}[\textbf{ENT-LHS-OR}]\\ \Delta_1{\vdash}^{\kappa}_{V}\Delta_3*\Delta_4\\ \Delta_2{\vdash}^{\kappa}_{V}\Delta_3*\Delta_5\end{array}}{\Delta_1{\vee}\Delta_2{\vdash}^{\kappa}_{V}\Delta_3*(\Delta_4{\vee}\Delta_5)}$$

$$\frac{\begin{array}{c}[\textbf{ENT-RHS-OR}]\\ \Delta_1{\vdash}^{\kappa}_{V}\Delta_i*\Delta_i^R\end{array}}{\Delta_1{\vdash}^{\kappa}_{V}(\Delta_2{\vee}\Delta_3)*\Delta_i^R} i{\in}\{2,3\}
\qquad
\frac{\begin{array}{c}[\textbf{ENT-RHS-EX}]\\ \Delta_1{\vdash}^{\kappa}_{V\cup\{w\}}([w/v]\Delta_2)*\Delta_3\\ \mathit{fresh}\ w \quad \Delta{=}\exists\ w\cdot\Delta_3\end{array}}{\Delta_1{\vdash}^{\kappa}_{V}(\exists\ v\cdot\Delta_2)*\Delta_3}
\qquad
\frac{\begin{array}{c}[\textbf{ENT-LHS-EX}]\\ [w/v]\Delta_1{\vdash}^{\kappa}_{V}\Delta_2*\Delta\\ \mathit{fresh}\ w\end{array}}{\exists v\cdot\Delta_1{\vdash}^{\kappa}_{V}\Delta_2*\Delta}$$

Fig. 5. Separation Constraint Entailment

matched nodes from antecedent in κ for better precision. For example, consider the following (valid) proof:

$$\frac{\dfrac{\begin{array}{c}(((\texttt{p}{=}\texttt{null}\wedge\texttt{n}{=}0)\vee(\texttt{p}{\neq}\texttt{null}\wedge\texttt{n}{>}0))\wedge\texttt{n}{>}0\wedge\texttt{m}{=}\texttt{n}) \implies \texttt{p}{\neq}\texttt{null}\\ \texttt{R}=(\texttt{n}{>}0\wedge\texttt{m}{=}\texttt{n})\end{array}}{\texttt{n}{>}0\wedge\texttt{m}{=}\texttt{n}\vdash_{\texttt{p::ll}\langle\texttt{n}\rangle}\texttt{p}{\neq}\texttt{null}*\texttt{R}}}{\texttt{p::ll}\langle\texttt{n}\rangle\wedge\texttt{n}{>}0\vdash\texttt{p::ll}\langle\texttt{m}\rangle\wedge\texttt{p}{\neq}\texttt{null}*\texttt{R}}$$

Had the predicate $\texttt{p::ll}\langle\texttt{n}\rangle$ not been kept and used, the proof would not have succeeded. Such an entailment would be useful when, for example, a list with positive length `n` is used as input for a function that requires a non-empty list.

Another feature of the entailment procedure is exemplified by the transfer of $\texttt{m}{=}\texttt{n}$ to the antecedent (and subsequently to the residue). In general, when a match occurs (rule [**ENT−MATCH**]) and an argument of the heap node coming from the consequent is free, the entailment procedure binds the argument to the corresponding variable from the antecedent and moves the equality to the antecedent. 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. This process is formalized by the function *freeEqn* below, where V is the set of existentially quantified variables :

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

For soundness, we perform a preprocessing step to ensure that variables appearing as arguments of heap nodes and predicates are i) distinct and ii) if they are free, they do not appear in the antecedent by adding (existentially quantified) fresh variables and equalities. This guarantees that the generated substitutions are well-defined. It also guarantees that the formula generated by *freeEqn* does not introduce any additional constraints over existing variables in the antecedent, as one side of each equation does not appear anywhere else in the antecedent. An additional outcome is that the order of picking nodes from the consequent for matching does not matter.

5 Unfold/Fold Mechanism

Unfold/fold operations can be used to handle well-founded inductive predicates in a deductive manner. In particular, we can unfold a predicate that appears in the antecedent that matches with a data node in the consequent. Correspondingly, we fold a predicate that appears in the consequent if it matches with a data node in the antecedent. The well-founded condition is sufficient to ensure termination.

5.1 Unfolding a Shape Predicate in the Antecedent

We apply an unfold operation on a predicate in the antecedent that matches with a data node in the consequent. Consider :

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

where $\Delta_{\mathtt{R}}$ captures the residual of entailment. For the entailment to succeed, we would unfold the $\mathtt{ll}\langle\mathtt{n}\rangle$ predicate in the antecedent twice to allow the two data nodes on the consequent to be matched up. This would result in the following reduction towards a residual state :

$$\begin{array}{ll}
\exists \mathtt{q}_1\cdot\mathtt{x}::\mathtt{node}\langle_,\mathtt{q}_1\rangle*\mathtt{q}_1::\mathtt{ll}\langle\mathtt{n}{-}1\rangle\wedge\mathtt{n}{>}3 & \vdash (\exists \mathtt{r}\cdot\mathtt{x}::\mathtt{node}\langle_,\mathtt{r}\rangle*\mathtt{r}::\mathtt{node}\langle_,\mathtt{y}\rangle\wedge\mathtt{y}{\neq}\mathtt{null})*\Delta_{\mathtt{R}}\\
\mathtt{q}_1::\mathtt{ll}\langle\mathtt{n}{-}1\rangle\wedge\mathtt{n}{>}3 & \vdash (\mathtt{q}_1::\mathtt{node}\langle_,\mathtt{y}\rangle\wedge\mathtt{y}{\neq}\mathtt{null})*\Delta_{\mathtt{R}}\\
\exists \mathtt{q}_2\cdot\mathtt{q}_1::\mathtt{node}\langle_,\mathtt{q}_2\rangle*\mathtt{q}_2::\mathtt{ll}\langle\mathtt{n}{-}2\rangle\wedge\mathtt{n}{>}3 & \vdash \mathtt{q}_1::\mathtt{node}\langle_,\mathtt{y}\rangle\wedge\mathtt{y}{\neq}\mathtt{null}*\Delta_{\mathtt{R}}\\
\mathtt{q}_2::\mathtt{ll}\langle\mathtt{n}{-}2\rangle\wedge\mathtt{n}{>}3\wedge\mathtt{q}_2{=}\mathtt{y} & \vdash \mathtt{y}{\neq}\mathtt{null}*\Delta_{\mathtt{R}}
\end{array}$$

Note that due to the well-founded condition, each unfolding exposes a data node that matches the data node in the consequent. Thus a reduction of the consequent immediately follows, which contributes to the termination of the entailment check. A formal definition of unfolding is given by the rule [UNFOLDING].

$$\frac{[\underline{\textsc{unfolding}}] \quad c\langle v^*\rangle{\equiv}\Phi \in P}{\mathit{unfold}(p::c\langle v^*\rangle) \ =_{df} \ [p/\mathtt{self}]\Phi}$$

5.2 Folding a Shape Predicate in the Consequent

We apply a fold operation when a data node in the antecedent matches with a predicate in the consequent. An example is :

$$\mathtt{x}::\mathtt{node}\langle 1,\mathtt{q}_1\rangle*\mathtt{q}_1::\mathtt{node}\langle 2,\mathtt{null}\rangle*\mathtt{y}::\mathtt{node}\langle 3,\mathtt{null}\rangle \ \vdash \ \mathtt{x}::\mathtt{ll}\langle\mathtt{n}\rangle\wedge\mathtt{n}{>}1*\Delta_{\mathtt{R}}$$

The fold step may be recursively applied but is guaranteed to terminate for well-founded predicate as it will reduce a data node in the antecedent for each recursive invocation. This reduction in the antecedent cannot go on forever. Furthermore, the fold operation may introduce bindings for the parameters of the folded predicate. In the above, we obtain $\mathtt{n}{=}2$ which may be transferred to the antecedent if $\mathtt{n}$ is free, but kept in the consequent otherwise. Since $\mathtt{n}$ is indeed free, our folding step would finally derive :

$$\mathtt{y}{::}\mathtt{node}\langle 3, \mathtt{null}\rangle \wedge \mathtt{n}{=}2 \ \vdash\ \mathtt{n}{>}1 * \Delta_{\mathtt{R}}$$

The effects of folding may seem similar to unfolding the predicate in the consequent. However, there is a subtle difference in their handling of bindings for free derived variables. If we choose to use unfolding on the consequent instead, these bindings may not be transferred to the antecedent. Consider the example below where $\mathtt{n}$ is free :

$$\mathtt{z}{=}\mathtt{null} \ \vdash\ \mathtt{z}{::}\mathtt{ll}\langle \mathtt{n}\rangle \wedge \mathtt{n}{>}{-}1 * \Delta_{\mathtt{R}}$$

By unfolding the predicate $\mathtt{ll}\langle\mathtt{n}\rangle$ in the consequent, we obtain :

$$\mathtt{z}{=}\mathtt{null} \vdash (\mathtt{z}{=}\mathtt{null}{\wedge}\mathtt{n}{=}0{\wedge}\mathtt{n}{>}{-}1){\vee}(\exists \mathtt{q}{\cdot}\mathtt{z}{::}\mathtt{node}\langle \_, \mathtt{q}\rangle{*}\mathtt{q}{::}\mathtt{ll}\langle \mathtt{n}{-}1\rangle{\wedge}\mathtt{n}{>}{-}1) * \Delta_{\mathtt{R}}$$

There are now two disjuncts in the consequent. The second one fails because it mismatches. The first one matches but still fails as the derived binding $\mathtt{n}{=}0$ was not transferred to the antecedent.

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.

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

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

A formal definition of folding is specified by rule [**FOLDING**]. Some heap nodes from κ are removed by the entailment procedure so as to match with the heap formula of predicate $p{::}c\langle v^*\rangle$. This 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 heap state. Our entailment also handles empty predicates correctly.

6 Soundness of Entailment

The following theorems state that our entailment check procedure(given in Fig. 5) is sound and terminating.

Theorem 6.1 (Soundness) *If entailment check $\Delta_1 \vdash \Delta_2 * \Delta$ succeeds, we have: for all s, h, if $s, h \models \Delta_1$ then $s, h \models \Delta_2 * \Delta$.*

Proof: Given in the technical report [15].

Theorem 6.2 (Termination) *The entailment check $\Delta_1 \vdash \Delta_2 * \Delta$ always terminates.*

Proof sketch: A well-founded measure exists for heap entailment. Matching and unfolding decrease nodes from the consequent. Fold operation has bounded recursive depth as each recursive fold operation always decreases the antecedent since shape predicate has the well-founded property. The size of antecedent is bounded despite unfolding since each unfold is always followed by a decrease of a data node from the consequent. At the end of a fold, a node from the consequent is also removed. A detailed proof is given in the technical report [15].

7 Implementation

We have built a prototype system using Objective Caml. The proof obligations generated by our verification are discharged by our entailment checking procedure with the help of Omega Calculator [16].

Programs	Verification Time (sec)
Linked List (size/length)	
delete	0.09
reverse	0.07
Circular List (size, cyclic structure)	
delete	0.09
count	0.16
Doubly Linked List (size, double links)	
append	0.16
flatten (from tree)	0.30
Sorted List (size, min, max, sortedness)	
delete	0.13
insertion_sort	0.27
selection_sort	0.41
bubble_sort	0.64
merge_sort	0.61
quick_sort	0.59

Programs	Verification Time (sec)
Binary Search Tree (min, max, sortedness)	
insert	0.20
delete	0.38
Priority Queue (size, height, max-heap)	
insert	0.45
delete_max	7.17
AVL Tree (size, height-balanced)	
insert	5.06
Red-Black Tree (size, black-height-balanced)	
insert	1.53
2-3 Tree (height-balanced)	
insert	24.41
Perfect Tree (perfectness)	
insert	0.26
Complete Tree (completeness)	
insert	1.50

Fig. 6. Verifying Data Structures with Arithmetic Properties

Fig 6 summarizes a suite of programs tested. These examples use complicated recursion and data structures with sophisticated shape and size properties. They help show that our approach is general enough to handle interesting data structures such as sorted lists, sorted trees, priority queues, various balanced trees, etc. in a uniform way. Verification time of a function includes time to verify all functions that it calls. The time required for shape and size verification is mostly within a couple of seconds. The average annotation cost (number of annotations/LOC ratio) for our examples is around 7%.

We have also investigated the precision/cost tradeoff of using $XPure_n$ and settled on $n = 1$ as the default. $XPure_0$ fails for many examples, while $XPure_2$ incurs substantial overheads without increasing precision for our examples.

8 Related Work

Separation Logic. The general framework of separation logic [17, 10] is highly expressive but undecidable. Likewise, [13] formalised the proof rules for handling abstract predicates (with scopes on visibility of predicates) but provided no automated procedure for checking the user supplied specifications. In the search for a decidable fragment of separation logic for automated verification, Berdine *et al.* [1] supports only a limited set of predicates *without* size properties, disjunctions and existential quantifiers. Similarly, Jia and Walker [11] postponed the handling of recursive predicates in their recent work on automated reasoning of pointer programs. Our approach is more pragmatic as we aim for a sound and terminating formulation of automated verification via separation logic but do not aim for completeness in the expressive fragment that we handle. On the inference front, Lee et al. [12] 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 properties. A recent work [8] has also formulated interprocedural shape inference but is restricted to just the list segment shape predicate. Sims [20] extends separation logic with fixpoint connectives and postponed substitution to express recursively defined formulae to model the analysis of while-loops. However, it is unclear how to check for entailment in their extended separation logic. While our work does not address the inference/analysis challenge, we have succeeded in providing direct support for automated verification via an expressive shape and size specification mechanism.
Shape Checking/Analysis. Many formalisms for shape analysis have been proposed for checking user programs' intricate manipulations of shapely data structures. One well-known work is Pointer Assertion Logic [14] by Moeller and Schwartzbach where shape specifications in monadic second-order logic are given by programmers for loop invariants and method pre/post conditions, and checked by their MONA tool. For shape inference, Sagiv et al. [19] presented a parameterised framework, called TVLA, using 3-valued logic formulae and abstract interpretation. Based on the properties expected of data structures, programmers must supply a set of predicates to the framework which are then used to analyse that certain shape invariants are maintained. However, most of these techniques were focused on analysing shape invariants, and did not attempt to track the size properties of complex data structures. An exception is the quantitative shape analysis of Rugina [18] where a data flow analysis was proposed to compute quantitative information for programs with destructive updates. By tracking unique points-to reference and its height property, their algorithm is

able to handle AVL-like tree structures. Even then, the author acknowledged the lack of a general specification mechanism for handling arbitrary shape/size properties.

Size Properties. In another direction of research, size properties have been most explored for declarative languages [9, 22, 6] as the immutability property makes their data structures easier to analyse statically. Size analysis was later extended to object-based programs [7] but was restricted to tracking either size-immutable objects that can be aliased and size-mutable objects that are unaliased, with no support for complex shapes. The Applied Type System (ATS) [5] was proposed for combining programs with proofs. In ATS, dependent types for capturing program invariants are extremely expressive and can capture many program properties with the help of accompanying proofs. Using linear logic, ATS may also handle mutable data structures with sharing. However, users must supply all expected properties, and precisely state where they are to be applied, with ATS playing the role of a proof-checker. Comparatively, we use a more limited class of constraint for shape and size analysis but supports automated modular verification.

Unfold/Fold Mechanism. Unfold/fold techniques were originally used for program transformation [4] on purely functional programs. A similar technique called unroll/roll was later used in alias types [21] to *manually* witness the isomorphism between a recursive type and its unfolding. Here, each unroll/roll step must be manually specified by programmer, in contrast to our approach which applies these steps automatically during entailment checking. In [1], an automated procedure that uses unroll/roll was given but it was hardwired to work for only `lseg` and `tree` predicates. Furthermore, it performs rolling by unfolding a predicate in the consequent which would miss bindings on free variables. Our unfold/fold mechanism is general, automatic and terminates for heap entailment checking.

9 Conclusion

We have presented a new approach to verifying pointer-based programs that can precisely track shape and size properties. Our approach is built on well-founded shape relations and well-formed separation constraints from which we have designed a sound procedure for heap entailment. We have implemented a verification system that is both precise and expressive. Our automated deduction mechanism is based on the unfold/fold reasoning of user-definable predicates that has been proven to be sound and terminating.

Acknowledgement

This work is supported by the Singapore-MIT Alliance and NUS research grant R-252-000-213-112.

References

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

2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, Springer LNCS 4111, 2006.
3. J. Bingham and Z. Rakamaric. A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs. In *VMCAI*, Springer LNCS 3855, pages 207–221, Charleston, U.S.A, January 2006.
4. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, January 1977.
5. C. Chen and H. Xi. Combining Programming with Theorem Proving. In *ACM SIGPLAN ICFP*, Tallinn, Estonia, September 2005.
6. W.N. Chin and S.C. Khoo. Calculating sized types. In *ACM SIGPLAN PEPM*, pages 62–72, Boston, United States, January 2000.
7. W.N. Chin, S.C. Khoo, S.C. Qin, C. Popeea, and H.H. Nguyen. Verifying Safety Policies with Size Properties and Alias Controls. In *ACM SIGSOFT ICSE*, St. Louis, Missouri, May 2005.
8. A. Gotsman, J. Berdine, and B. Cook. Interprocedural Shape Analysis with Separated Heap Abstractions. In *SAS*, Springer LNCS, Seoul, Korea, August 2006.
9. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *ACM POPL*, pages 410–423. ACM Press, January 1996.
10. S. Isthiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM POPL*, London, January 2001.
11. L. Jia and D. Walker. ILC: A foundation for automated reasoning about pointer programs. In *15th ESOP*, March 2006.
12. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*. Springer Verlag, April 2005.
13. M.J.Parkinson and G.M.Bierman. Separation logic and abstraction. In *ACM POPL*, pages 247–258, 2005.
14. A. Moeller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *ACM PLDI*, June 2001.
15. H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape, Size and Bag Properties via Separation Logic. Technical report, SoC, Natl Univ. of Singapore, July 2006. avail. at http://www.comp.nus.edu.sg/~chinwn/papers/verify-report.pdf.
16. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.
17. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, Copenhagen, Denmark, July 2002.
18. R. Rugina. Quantitative Shape Analysis. In *SAS*, Springer LNCS, Verona, Italy, August 2004.
19. S. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3), May 2002.
20. É-J. Sims. Extending separation logic with fixpoints and postponed substitution. *Theoretical Computer Science*, 351(2):258–275, 2006.
21. D. Walker and G. Morrisett. Alias Types for Recursive Data Structures. In *TIC*, Springer LNCS 2071, pages 177–206, 2000.
22. H. Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1998.

A Semantic Model

The semantics of our constraints is that of separation logic [17], with extensions to handle our shape predicates.

To define the model we assume sets *Loc* of locations (positive integer values), *Val* of primitive values, with $0 \in \mathit{Val}$ denoting `null`, *Var* of variables (program variables and other meta variables), and *ObjVal* of object values stored in the heap, with $c[f_1{\mapsto}\nu_1,..,f_n{\mapsto}\nu_n]$ denoting an object value of data type c where $\nu_1,..,\nu_n$ are current values of the corresponding fields $f_1,..,f_n$. Let $s,h \models \Phi$ denotes that stack s and heap h form a model of the constraint Φ, with h,s from the following concrete domains:

$$h \in \mathit{Heaps} =_{df} \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{ObjVal} \qquad\qquad s \in \mathit{Stacks} =_{df} \mathit{Var} \rightarrow \mathit{Val} \cup \mathit{Loc}$$

Function $dom(f)$ returns the domain of function f. Note that we use $\mapsto$ to denote mappings, not the points-to assertion in separation logic, which has been replaced by $p{::}c\langle v^*\rangle$ in our notation. The model $s \models \pi$ for pure constraint is standard and left to the technical report [15], while the model for separation constraint is defined below.

Definition A.1 (Model for Separation Constraint)

$$\begin{array}{lll}
s,h \models \Phi_1 \vee \Phi_2 & \texttt{iff} & s,h \models \Phi_1 \ \mathit{or}\ s,h \models \Phi_2 \\
s,h \models \exists v^* \cdot \kappa \wedge \pi & \texttt{iff} & \exists \nu^* \cdot s[v^* {\mapsto} \nu^*], h \models \kappa \ \mathit{and}\ s[v^* {\mapsto} \nu^*] \models \pi \\
s,h \models \kappa_1 * \kappa_2 & \texttt{iff} & \exists h_1, h_2 \cdot h_1 \perp h_2 \ \mathit{and}\ h = h_1 \cdot h_2 \ \mathit{and} \\
 & & s,h_1 \models \kappa_1 \ \mathit{and}\ s,h_2 \models \kappa_2 \\
s,h \models \texttt{emp} & \texttt{iff} & dom(h) = \emptyset \\
s,h \models p{::}c\langle v_{1..n}\rangle & \texttt{iff} & \texttt{data}\ c\ \{t_1\, f_1,..,t_n\, f_n\} \in P,\ h = [s(p) {\mapsto} r], \\
 & & \mathit{and}\ r = c[f_1 {\mapsto} s(v_1),..,f_n {\mapsto} s(v_n)] \\
 & \texttt{or} & (c\langle v_{1..n}\rangle \equiv \Phi) \in P \ \mathit{and}\ s,h \models [p/\texttt{self}]\Phi
\end{array}$$

Note that $h_1 \perp h_2$ indicates h_1 and h_2 are domain-disjoint, $h_1 \cdot h_2$ denotes the union of disjoint heaps h_1 and h_2.

We intend to approximate each separation constraint by a formula of the form: $\beta ::= \mathbf{ex}\ i \cdot \beta \mid \bigvee(\exists v^* \cdot \pi)^*$ where $\mathbf{ex}\ i$ construct is being used to capture a distinct symbolic address i that has been abstracted from a heap node or predicate. This abstraction has the following model $s,h \models \beta$, namely :

Definition A.2 (Model for Heap Approximation)

$$\begin{array}{lll}
s,h \models \bigvee(\exists v^* \cdot \pi)^* & \texttt{iff} & s \models \bigvee(\exists v^* \cdot \pi)^* \\
s,h \models \mathbf{ex}\ i \cdot \beta & \texttt{iff} & (p{=}i \wedge i{>}0) \in \beta \quad \mathit{and} \quad s, h{-}\{s(p)\} \models [p/i]\beta
\end{array}$$

Furthermore, we may soundly relate a separation constraint Φ and its abstraction β by the relation $\Phi \models \beta$, defined as follows :

$$\forall s,h \cdot (s,h \models \Phi \implies s,h \models \beta)$$

B Forward Verification

B.1 Forward Verification Rules

The complete set of forward verification rules are given in Fig. 7. Note that path-sensitivity is captured by [FV−IF] rule, flow-sensitivity is captured by [FV−SEQ] rule, and context sensitivity by the [FV−CALL] rule.

$$\boxed{\textbf{FV-PRED}} \qquad \frac{XPure_0(\Phi) \Longrightarrow [0/\texttt{null}](\pi_{inv})}{\vdash c\langle v^*\rangle = \Phi\ \texttt{inv}\ \pi_{inv}}$$

$$\boxed{\textbf{FV-IF}} \qquad \frac{\vdash \{\Delta\wedge v'\}\, e_1\, \{\Delta_1\} \qquad \vdash \{\Delta\wedge\neg v'\}\, e_2\,\{\Delta_2\}}{\vdash \{\Delta\}\, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\, \{\Delta_1\vee\Delta_2\}}$$

$$\boxed{\textbf{FV-CONST}} \qquad \frac{\Delta_1 = (\Delta\wedge eq_\tau(\texttt{res},k))}{\vdash\{\Delta\}\, k^\tau\, \{S\}}$$

$$\boxed{\textbf{FV-LOCAL}} \qquad \frac{\vdash\{\Delta\}\,e\,\{\Delta_1\}}{\vdash\{\Delta\}\,\{t\ v;\ e\}\,\{\exists v,v'\cdot\Delta_1\}}$$

$$\boxed{\textbf{FV-NEW}} \qquad \frac{\Delta_1=(\Delta * \texttt{res}{::}c\langle v_1',..,v_n'\rangle)}{\vdash\{\Delta\}\,\texttt{new}\ c(v_1,..,v_n)\,\{\Delta_1\}}$$

$$\boxed{\textbf{FV-VAR}} \qquad \frac{\Delta_1=(\Delta\wedge\texttt{res}=v')}{\vdash\{\Delta\}\,v\,\{\Delta_1\}}$$

$$\boxed{\textbf{FV-ASSIGN}} \qquad \frac{\vdash\{\Delta\}\,e\,\{\Delta_1\} \quad \Delta_2=\exists\texttt{res}\cdot(\Delta_1\wedge_{\{v\}} v'=\texttt{res})}{\vdash\{\Delta\}\,v{:=}e\,\{\Delta_2\}}$$

$$\boxed{\textbf{FV-SEQ}} \qquad \frac{\begin{array}{c}\vdash\{\Delta\}\,e_1\,\{\Delta_1\} \\ \vdash\{\Delta_1\}\,e_2\,\{\Delta_2\}\end{array}}{\vdash\{\Delta\}\,e_1;e_2\,\{\Delta_2\}}$$

$$\boxed{\textbf{FV-CALL}} \qquad \frac{\begin{array}{c} t_0\ m(t_1\ v_1,..,t_n\ v_n)\ \texttt{where}\ \Phi_{pr} \mathrel{\ast\!\!\!\longrightarrow} \Phi_{po}\ \{..\} \\ \rho=[v_i'/v_i] \quad \Delta\vdash\rho\Phi_{pr}*\Delta_1 \quad W=\{v_1,..,v_n\} \quad \Delta_2=(\Delta_1 *_W \Phi_{po})\end{array}}{\vdash\{\Delta\}\,m(v_1..v_n)\,\{\Delta_2\}}$$

$$\boxed{\textbf{FV-FIELD-READ}} \qquad \frac{\begin{array}{c}\Delta\vdash v'{::}c\langle v_1,..,v_n\rangle * \Delta_1 \quad fresh\ v_1..v_n \\ \Delta_2 = \exists v_1..v_n\cdot(\Delta_1 * v'{::}c\langle v_1,..,v_n\rangle\wedge\texttt{res}=v_i)\end{array}}{\vdash\{\Delta\}\,v.f_i\,\{\Delta_2\}}$$

$$\boxed{\textbf{FV-FIELD-UPDATE}} \qquad \frac{\begin{array}{c}\Delta\vdash v'{::}c\langle v_1,..,v_n\rangle * \Delta_1 \quad fresh\ v_1..v_n \\ \Delta_2 = \exists v_1..v_n\cdot(\Delta_1 * v'{::}[v_0'/v_i]c\langle v_1,..,v_n\rangle)\end{array}}{\vdash\{\Delta\}\,v.f_i{:=}v_0\,\{\Delta_2\}}$$

$$\boxed{\textbf{FV-WHILE}} \qquad \frac{\begin{array}{c}W=\{v\}\cup vars(e) \quad \Delta\vdash\Phi_{pr}*\Delta_1 \quad \Delta_2=\Delta_1 *_W \Phi_{po} \quad \Delta_3=\Phi_{pr}\wedge nochange(W) \\ \vdash\{\Delta_3\wedge v'\}\,e\,\{\Delta_4\} \quad \Delta_4\vdash\Phi_{pr}*\Delta_5 \quad \Delta_6=\Delta_5 *_W\Phi_{po} \quad (\Delta_3\wedge\neg v')\vee\Delta_6\vdash\Phi_{po}*\Delta_7\end{array}}{\vdash\{\Delta\}\,\texttt{while}\ v\ \texttt{where}\ \Phi_{pr}\mathrel{\ast\!\!\!\longrightarrow}\Phi_{po}\ \texttt{do}\ e\,\{\Delta_2\}}$$

$$\boxed{\textbf{FV-METH}} \qquad \frac{V=\{v_1..v_n\} \quad W=prime(V) \quad \Delta=\Phi_{pr}\wedge nochange(V) \quad \vdash\{\Delta\}\,e\,\{\Delta_1\} \quad (\exists W\cdot\Delta_1)\vdash\Phi_{po}*\Delta_2}{\vdash t_0\ mn(t_1\ v_1,..,t_n\ v_n)\ \texttt{where}\ \Phi_{pr}\mathrel{\ast\!\!\!\longrightarrow}\Phi_{po}\ \{e\}}$$

Fig. 7. A Complete Set of Forward Verification Rules

B.2 Soundness of Verification

The soundness of our verification rules is defined with respect to a small-step dynamic semantics, which is defined using the transition relation $\langle s,h,e\rangle \hookrightarrow \langle s_1,h_1,e_1\rangle$, which means if e is evaluated in stack s, heap h, then e reduces in one step to e_1 and generates new stack s_1 and new heap h_1. Full definition of the relation can be found in the technical report [15]. We also need to extract the post-state of a heap constraint by:

Definition B.1 (Poststate) *Given a constraint Δ, $Post(\Delta)$ captures the relation between primed variables of Δ. That is :*

$$\begin{array}{l} Post(\Delta) \ =_{df}\ \rho\ (\exists V\cdot\Delta), \qquad \textit{where} \\ V=\{v_1,..,v_n\}\ \textit{denotes all unprimed program variables in}\ \Delta \\ \rho=[v_1/v_1',..,v_n/v_n'] \end{array}$$

Theorem B.1 (Preservation) *If*

$$\vdash \{\Delta\}\ e\ \{\Delta_2\} \quad s,h \models \mathit{Post}(\Delta) \quad \langle s,h,e\rangle \hookrightarrow \langle s_1,h_1,e_1\rangle$$

Then there exists Δ_1*, such that* $s_1,h_1 \models \mathit{Post}(\Delta_1)$ *and* $\vdash \{\Delta_1\}\ e_1\ \{\Delta_2\}$.

Proof: By structural induction on e. Details are in the technical report [15].

Theorem B.2 (Progress) *If* $\vdash \{\Delta\}\ e\ \{\Delta_1\}$*, and* $s,h \models \mathit{Post}(\Delta)$*, then either* e *is a value, or there exist* s_1, h_1*, and* e_1*, such that* $\langle s,h,e\rangle \hookrightarrow \langle s_1,h_1,e_1\rangle$.

Proof: By structural induction on e. Details are in the technical report [15].

Theorem B.3 (Safety) *Consider a close term* e *without free variables in which all methods have been successfully verified. Assuming unlimited stack/heap spaces and that* $\vdash \{\mathtt{true}\}\ e\ \{\Delta\}$*, then either* $\langle [], [], e\rangle \hookrightarrow^* \langle [], h, v\rangle$ *terminates with a value* v *that is subsumed by the postcondition* Δ*, or it diverges* $\langle [], [], e\rangle \not\hookrightarrow^*$.

Proof: Follows directly from Theorems B.2 and B.1.