Automated Verification of Shape and Size Properties via Separation Logic

Huu Hai Nguyen[1], Cristina David[2], Shengchao Qin[3], Wei-Ngan Chin[1,2]
[1]Singapore-MIT Alliance, [2]Department of Computer Science, National University of Singapore, [3]Department of Computer Science, Durham University

***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.**

***Index Terms*— Verification, Separation Logic**

I. Introduction

In recent years, separation logic has emerged as a contender for formal reasoning about heap-manipulating imperative programs. While the foundations of separation logic have been laid in seminal papers by Reynolds [16] and Isthiaq and O'Hearn [10], new automated reasoning tools based on separation logic, 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.
- 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 the procedure in the presence of recursive user-defined shape predicates.
- We have implemented a prototype verification system with the above features and have also proven both its soundness and termination.

Wei-Ngan Chin is with the Department of Computer Science, School of Computing, National University of Singapore,3 Science Drive 2, Singapore 117543, Republic of Singapore. Email: chinwn@comp.nus.edu.sg
Cristina David is with the Department of Computer Science, School of Computing, National University of Singapore,3 Science Drive 2, Singapore 117543, Republic of Singapore. Email: davidcri@comp.nus.edu.sg
Huu Hai Nguyen is the author for correspondence. He is a student in the Computer Science Programme, Singapore-MIT Alliance. Email: nguyenh2@comp.nus.edu.sg
Shengchao Qin is with the Department of Computer Science, Durham University. Email: shengchao.qin@durham.ac.uk.

II. User-Definable Shape Predicates

Separation logic [16], [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 $\texttt{p::c}\langle \texttt{v}^* \rangle$ to denote two things in our system. When $\texttt{c}$ is a data name, $\texttt{p::c}\langle \texttt{v}^* \rangle$ 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, $\texttt{p::c}\langle \texttt{v}^* \rangle$ 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. As an example, a singly linked list with length `n` is described by :

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

Note that the parameter `n` captures a *derived* value. The above definition asserts that an `ll` list can be empty (the base case $\texttt{self=null}$) or consists of a head data node (specified by $\texttt{self::node}\langle\texttt{i},\texttt{q}\rangle$) and a separate tail data structure which is also an $\texttt{ll}$ list ($\texttt{q::ll}\langle\texttt{m}\rangle$). 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 $\texttt{ll}$ 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 $\texttt{n}$, is described by :

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

The $\texttt{dll}$ shape predicate has a parameter $\texttt{p}$ that represents the $\texttt{prev}$ field of the first node of the doubly linked-list. It captures a chain of nodes that are to be traversed via the $\texttt{next}$ field starting from the current node $\texttt{self}$. The nodes accessible via the $\texttt{prev}$ field of the $\texttt{self}$ node are not part of the $\texttt{dll}$ list. This example also highlights some shortcuts we may use to make shape specification shorter. We use underscore _ to denote an anonymous variable. Non-parameter variables in the RHS of the shape definition, such as $\texttt{q}$, 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, the minimum and maximum elements of the list.

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

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 $\Phi_{pr} *\!\!\!\rightarrow \Phi_{po}$ 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 $\texttt{res}$ is also used in the postcondition to denote the result of a method. The postcondition of $\texttt{insertion\_sort}$ shows that the output list is sorted and has the same number of nodes as the input list.

III. Automated Verification

We use an object-based imperative language. Let P be the program being checked. With pre/post conditions declared for each method in P, 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.

We present the detailed verification of the first branch of the $\texttt{insert}$ function from Sec II. Note that program variables appear primed in formulae whereas logical variables unprimed. The proof is straightforward, except for the last step where a disjunctive heap state is folded to form a shape predicate. This step is performed by our entailment checking procedure.

```
{x'::sortl⟨n, mi, ma⟩ * vn'::node⟨v, _⟩} // precondition
    if (vn.val ≤ x.val) then {
{(x'::node⟨mi, null⟩ * vn'::node⟨v, _⟩∧
    mi=ma ∧ n=1 ∧ v≤mi)
  ∨ (∃q, k · x'::node⟨mi, q⟩ * q::sortl⟨n−1, k, ma⟩*
    vn'::node⟨v, _⟩ ∧ mi≤k ∧ mi≤ma ∧ n≥2 ∧ v≤mi)}
    // unfold and conditional
      vn.next := x;
{(x'::node⟨mi, null⟩ * vn'::node⟨v, x'⟩∧
    mi=ma ∧ n=1 ∧ v≤mi)
  ∨ (∃q, k · x'::node⟨mi, q⟩ * q::sortl⟨n−1, k, ma⟩*
    vn'::node⟨v, x'⟩ ∧ mi≤k ∧ mi≤ma ∧ n≥2 ∧ v≤mi)}
    // field update
      vn
{(x'::node⟨mi, null⟩ * vn'::node⟨v, x'⟩∧
    mi=ma ∧ n=1 ∧ v≤mi ∧ res=vn')
  ∨ (∃q, k · x'::node⟨mi, q⟩ * q::sortl⟨n−1, k, ma⟩*
    vn'::node⟨v, x'⟩ ∧ mi≤k ∧ mi≤ma ∧ n≥2 ∧ v≤mi∧
    res=vn')} // returned value
    }
{res::sortl⟨n+1, min(v, mi), max(v, ma)⟩}
    // fold to postcondition
```

IV. Entailment

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

Entailment between separation formulae 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.

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 procedure works by successively matching up heap nodes that can be proven aliased. As the matching process is incremental, we keep the successfully matched nodes from antecedent in κ for better precision. For example, consider the following (valid) proof:

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

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 $\texttt{n}$ is used as input for a function that requires a non-empty list.

V. 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 [15].

Fig 1 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%.

VI. Related Work

Separation Logic. The general framework of separation logic [16], [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 [19] 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. [18] 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 [17] 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], [21], [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 [20] to *manually* witness the isomorphism between a recursive type and its unfolding. Here, each unroll/roll

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. 1. Verifying Data Structures with Arithmetic Properties

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.

VII. 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.

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

[16] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, Copenhagen, Denmark, July 2002.

[17] R. Rugina. Quantitative Shape Analysis. In *SAS*, Springer LNCS, Verona, Italy, August 2004.

[18] S. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3), May 2002.

[19] É-J. Sims. Extending separation logic with fixpoints and postponed substitution. *Theoretical Computer Science*, 351(2):258–275, 2006.

[20] D. Walker and G. Morrisett. Alias Types for Recursive Data Structures. In *TIC*, Springer LNCS 2071, pages 177–206, 2000.

[21] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.