Automated Verification of Shape, Size and Bag Properties via User-Defined Predicates in Separation Logic *

Wei-Ngan Chin[a], Cristina David[a], Huu Hai Nguyen[a], and Shengchao Qin[b†]

[a]Department of Computer Science, National University of Singapore, Singapore

[b]School of Computing, University of Teesside, United Kingdom

Despite their popularity and importance, pointer-based programs remain a major challenge for program verification. In recent years, separation logic has emerged as a contender for formal reasoning of pointer-based programs. Recent works have focused on specialized provers that are mostly based on fixed sets of predicates. In this paper, we propose an automated verification system for ensuring the safety of pointer-based programs, where specifications handled are concise, precise and expressive. Our approach uses *user-definable* predicates to allow programmers to describe a wide range of data structures with their associated *shape*, *size* and *bag* (multi-set) properties. To support automatic verification, we design a new entailment checking procedure that can handle *well-founded* predicates (that may be recursively defined) using *unfold/fold* reasoning. We have proven the soundness and termination of our verification system and built a prototype system to demonstrate the viability of our approach.

1. Introduction

Separation logic supports reasoning about shared mutable data structures, i.e., structures where an updatable field can be referenced from more than one point. Using it, the specification of heap memory operations and pointer manipulations can be made more precise (with the help of must-aliases) and concise (with the help of frame conditions). While the foundations of separation logic have been laid in seminal papers by Reynolds [51] and Ishtiaq and O'Hearn [25], new automated reasoning tools based on separation logic [4,19] have gradually appeared. Several recent works [3,16] have designed specialised solvers that work for a fixed set of predicates (e.g. the predicate `lseg` to describe a segment of linked-list nodes). This paper focuses on an automated reasoner that works for user-defined predicates.

When designing a static reasoning mechanism for programs, two key issues that we need to consider are *automation* and *expressivity*. Automation comes in two main flavors based either on automated *verification* or on automated *inference*. In automated verification for imperative programs, pre/post conditions are typically specified for each method/procedure (and an invariant given for each loop) before the reasoning system automatically checks if each given program code is correct with respect to the given pre/post/invariant annotations. In automated inference [53], these annotations are expected to be derived by the reasoning system. Intraprocedural

*This paper is an expanded version of our VMCAI'07 and ICECCS'07 papers.

†Author of Correspondence

inference is expected to derive loop invariants, while interprocedural inference is also expected to derive pre/post conditions for methods/procedures. While inference can be said to be more useful in general, it must be said that automated verification is of great importance too, and it can complement inference in several ways. Firstly, programmers' insights may be captured via annotations to handle difficult examples that inference system may be unable to handle. Secondly, the verification system may act as an independent checker on the inference system. Thirdly, the verification system plays a useful role within a "proof-carrying code" system [42], where annotations of untrusted components must always be verified prior to their actual execution. Furthermore, an automated verification system allows us to explore the boundary of what is achievable in software verification which has been identified as a Grand Challenge [23,27] for computing research.

Expressivity is another major issue for automated reasoning systems. By allowing more properties to be easily captured, where possible, our verification tool can support better safety and give higher assurance on program correctness. This paper's main goal is to raise the level of expressivity that is possible with an automated verification system based on separation logic, so as to support the specification and verification of shape, size and bag properties of imperative programs. We make the following technical contributions towards this overall goal:

- We provide a *user-specified* predicate specification mechanism that can capture a wide range of data structures with different kinds of shapes. By shapes, we mean the expected forms of some linked data structures, such as cyclic lists, doubly-linked list or even height-balanced trees and sorted lists/trees. Moreover, we provide a novel mechanism to soundly approximate each predicate describing a data structure by a heap-independent pure formula which plays an important role in entailment proving. This allows our proof obligations to be eventually discharged by classical provers, such as Omega or Isabelle (Secs 2 and 4).

  There are data structures that are beyond the capability of the current system. This is due to the fact that, in our approach, references between the objects of a data structure are captured by passing object references and fields as parameters to predicate invocations. Consequently, our predicates cannot precisely capture data structures with non-local references, which do not have a direct relationship with fields of surrounding objects, but rather are determined by some global constraint.

- We improve the expressiveness of our automated verification tool by allowing it to capture shape, size and bag properties from each predicate that is being used to define some data structure. The size properties may capture sophisticated data structure invariants, such as orderedness (for sorted list/trees) and also balanced height properties (for AVL-trees). The bag constraints enable expressing reachability properties, as they can capture the nodes (or values) reachable inside a heap predicate. For instance, our specification can capture all elements of a list, our verification system can then prove the preservation of the elements inside the list after sorting. These abstract properties are important as they are easily specified by users, but are not automatically handled by existing verification systems based on separation logic (Sec 3).

- We design a new procedure to prove entailment of separation heap constraints. This procedure uses *unfold/fold* reasoning to deal with predicate definitions that describe some

data structures with sophisticated shapes/properties. While the unfold/fold mechanism may not be totally new, we have identified sufficient conditions for soundness and termination of the procedure in the presence of user-defined recursive predicates (Sec 4).

- We have implemented a prototype verification system with the above features and have also proven both its soundness and termination (Secs 5 and 6).

We briefly survey the state-of-the-art on research that focuses on using separation logic for either program analysis or verification. The general framework of separation logic [51,25] is highly expressive but undecidable. In the search for a decidable fragment of separation logic for automated verification, Berdine et al. [3] support only a limited set of predicates *without* size properties, disjunctions and existential quantifiers. Similarly, Jia and Walker [26] postpone 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. In VMCAI'07 [44], we present the proposal for a verification system that supports user-defined predicates with size properties. In ICECCS'07 [9], we extend the proposal with a bag/set specification mechanism. The current paper is a journal version of these two papers. We have added clarification regarding the role and mechanism of implicit vs explicit instantiation, and provided proofs on the soundness of our verification system.

On the inference front, Lee et al. [38] conduct 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. Gotsman et al. [19] also formulate an interprocedural shape inference which is restricted to just the list segment shape predicate. Sims [54] extends separation logic with fixpoint connectives and postpones 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 specification mechanism through user-specified predicates with shape, size and bag properties. In the following sections, we provide some details on the symbolic mechanisms used to provide automated program verification for a procedural language with support for pointers to heap-based data structures.

This work is organized in eight sections. After the introduction, Sec 2 presents the language and specifications. Sec 3 and Sec 4 describe the forward verification and entailment rules, respectively, whereas their soundness is proved in Sec 5. Sec 6 summarizes the experimental results, Sec 7 reviews some related works, and Sec 8 concludes our work. The proofs for our soundness rules are given in the Appendix.

2. Language and Specifications

In this section, we first introduce a core imperative language and then depict our specification language which supports user-defined shape predicates with shape, size and bag properties.

2.1. Language

We provide a simple imperative language in Figure 1. A program comprises a list of type declarations ($tdecl^*$) and a list of method declarations ($meth^*$). We use the supscript * to help

$$
\begin{array}{lll}
P & ::= & \mathit{tdecl}^* \ \mathit{meth}^* \\
\mathit{tdecl} & ::= & \mathit{datat} \mid \mathit{spred} \\
\mathit{datat} & ::= & \texttt{data}\ c\ \{\ \mathit{field}^*\ \} \\
\mathit{field} & ::= & \mathit{type}\ v \\
\mathit{type} & ::= & c \mid \tau \\
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{float} \mid \texttt{void} \\
\mathit{meth} & ::= & \mathit{type}\ \mathit{mn}\ ((\texttt{ref}\ \mathit{type}\ v)^*, (\mathit{type}\ v)^*)\ \texttt{where}\ \mathit{mspec}\ \{e\} \\
e & ::= & \texttt{null} \mid k^\tau \mid v \mid v.f \mid v{:=}e \mid v_1.f{:=}v_2 \mid \texttt{new}\ c(v^*) \\
 & & \mid e_1; e_2 \mid \mathit{type}\ v;\ e \mid \mathit{mn}(v^*) \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2
\end{array}
$$

Figure 1. A Core Imperative Language

denote a list of items, for example v^* denotes a list of variables, $v_1, .., v_n$. With regard to the used terminals, c denotes the name of a user-defined data type, v, v_1, v_2 stand for variable names, mn represents a method name, k is a numeric constant, and f denotes a field name. For simplicity, we shall assume that programs and specification formulas we use are well-typed. To simplify the presentation but without loss of expressiveness, we allow only one-level field access like $v.f$ (rather than $v.f_1.f_2...$), and we allow only boolean variables (but not expressions) to be used as the test conditions for conditionals. (for brevity, we use the variable v in the test condition for conditionals to denote a boolean variable). The language supports data type declaration via *datat*, and shape predicate[3] definition via *spred*. The syntax for shape predicates is given in the next subsection.

The following data node declarations can be expressed in our language and will be used as examples throughout the paper. Note that they are recursive data declarations with different numbers 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 }
```

Each method *meth* is associated with a pre/post specification *mspec*, the syntax of which will be given in the next subsection. For simplicity, we assume that variable names declared inside each method are all distinct.

Pass-by-reference parameters are marked with `ref`. For formalization convenience, they are grouped together. This pass-by-reference mechanism is useful for supporting reference parameters of languages such as C♯. As an example of pass-by-reference parameters, the following function allows the actual parameters of $\{\texttt{x}, \texttt{y}\}$ to be swapped at its callers' sites.

```
void swap(ref node2 x, ref node2 y) where ··· { node2 z:=x ; x:=y ; y:=z }
```

Furthermore, these parameters allow each iterative loop to be directly converted to an equivalent tail-recursive method, where mutation on parameters are made visible to the caller via pass-by-reference. This technique of translating away iterative loops is standard and is helpful in further minimising our core language.

[3]Shape predicates are predicates specifying data structure shapes. Our shape predicates can also specify size and bag properties of data structures.

The standard insertion sort algorithm can be written in our language as follows:

```
node insert(node x, node vn) where ···
{ 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 ···
{ if (y.next=null) then y
  else {
   y.next:=insertion_sort(y.next);
   insert(y.next,y)}}
```

The `insert` method takes a sorted list `x` and a node `vn` that is to be inserted in the correct location of its sorted list. The `insertion_sort` method recursively applies itself (sorting) to the tail of its input list, namely `y.next`, before inserting the first node, namely `y`, into its now sorted tail. Note that we use an expression-oriented language where the last subexpression (e.g. e_2 from $e_1;e_2$) denotes the result of an expression. The missing method specifications (to be filled in the place of $\cdots$), denoted by $mspec$, are described in the next section.

2.2. The Specification Language

Separation logic [51,25] extends Hoare logic [21] to support reasoning about shared mutable data structures. One connective that it adds to classical logic is separation conjunction $*$. The separation formula $p_1 * p_2$ means that the heap can be split into two disjoint parts in which p_1 and p_2 hold, respectively. Our work will make use of this connective in our specifications. In our approach, the verifier takes as input a command and a precondition. It then derives the strongest postcondition upon termination of the command and checks if the strongest postcondition implies the declared postcondition.

We propose a mechanism based on predicates (that may be recursively defined) to allow user specification of data structure shapes with size and reachability properties. Our shape specification is based on separation logic with support for disjunctive heap states. Furthermore, each shape predicate may have pointer, integer or bag parameters to capture relevant properties of data structures.

Separation logic [51,25] uses the notation $\mapsto$ to denote singleton heaps, e.g. the formula $\mathtt{p}{\mapsto}[\mathtt{val}:3,\mathtt{next}:\mathtt{l}]$ represents a singleton heap referred to by `p`, where $[\mathtt{val}:3,\mathtt{next}:\mathtt{l}]$ is a data record containing fields `val` and `next`. On the other hand, separation logic also uses predicate formulas to denote more complicated shapes, e.g. $\mathtt{lseg}(\mathtt{p},\mathtt{q})$ represents list segments starting from the head pointer `p` and containing all the data nodes until the `q` pointer is reached. In our system, we unify these two different representations into one form: $\mathtt{p}::\mathtt{c}\langle\mathtt{v}^*\rangle$. When `c` is a data type name, $\mathtt{p}::\mathtt{c}\langle\mathtt{v}^*\rangle$ stands for a singleton heap $\mathtt{p}{\mapsto}[(\mathtt{f}:\mathtt{v})^*]$ where $\mathtt{f}^*$ are fields of data declaration `c`. When `c` is a predicate name, $\mathtt{p}::\mathtt{c}\langle\mathtt{v}^*\rangle$ stands for the predicate formula $\mathtt{c}(\mathtt{p},\mathtt{v}^*)$. The reason we distinguish the first parameter from the rest is that each predicate has an implicit parameter `root` as its first parameter. Effectively, this is a "root" pointer to the specified data structure that guides data traversal and facilitates the definition of *well-founded* predicates (given later in this section). As an example, an acyclic linked list (that terminates with a `null` reference) can be described by:

$$\begin{array}{lcl}\mathtt{root}::\mathtt{ll}\langle\mathtt{n}\rangle & \equiv & (\mathtt{root}{=}\mathtt{null}\wedge\mathtt{n}{=}0)\vee\\ & & (\exists\mathtt{i},\mathtt{m},\mathtt{q}\cdot\mathtt{root}::\mathtt{node}\langle\mathtt{i},\mathtt{q}\rangle{*}\mathtt{q}::\mathtt{ll}\langle\mathtt{m}\rangle\wedge\mathtt{n}{=}\mathtt{m}{+}1)\\ \textbf{inv}\ \mathtt{n}{\geq}0 & & \end{array}$$

The parameter n captures a *derived* value that denotes the length of the acyclic list starting from root pointer. The above definition asserts that an ll list can be empty (the base case root=null) or consists of a head data node (specified by root::node⟨i, q⟩) and a separate tail data structure which is also an ll list (q::ll⟨m⟩). The $*$ connector ensures that the head node and the tail reside in disjoint heaps. We also specify a default invariant n≥0 that holds for all ll lists. (This invariant can be verified by checking that each disjunctive branch of the predicate definition always implies its stated invariant. In the case of ll predicate, the disjunctive branch with $n = 0$ implies the given invariant n≥0. Similarly, the $n = m + 1$ branch together with m≥0 from the invariant of q::ll⟨m⟩ also implies the given invariant n≥0.) Our predicate uses existential quantifiers for local values and pointers, such as i, m, q. The syntax for inductive shape predicates is given in Figure 2. For each shape definition *spred*, the heap-independent invariant π over the parameters $\{root, v^*\}$ holds for each instance of the predicate. Types need not be given in our specification as we have an inference algorithm to automatically infer non-empty types for specifications that are well-typed. For the ll predicate, our type inference can determine that m, n, i are of int type, while $root, q$ are of the node type. As the construction of type inference algorithm is quite standard for a language without polymorphism [47], its description is omitted in the current paper. Note that arbitrary recursive shape relation can lead to non-termination in our reasoning. We avoid this problem by proposing a notion of well-founded shape predicates, which will be discussed later in the current section.

The use of separation logic enables more precise and concise reasoning for heap memory, as it can easily support must-aliasing and local reasoning. Regarding must-aliasing, when we specify that x::node⟨3, y⟩∗y::node⟨5, x⟩ to be a precondition of some method, we can immediately determine that x, y are non-aliased, namely x≠y due to the use of the separation conjunction, while $x.next = y$ and $y.next = x$ are must-aliases for the two fields from the heap formula. In contrast, if we had used the formula x::node⟨3, y⟩∧y::node⟨5, x⟩, we may not be able to determine if x, y are aliased with each other, or not. Regarding local reasoning about heap-allocated data structures [16,46], it means that reasoning about a command concerns only the part of the heap that the command reads or writes, i.e. the commands footprint. Note that local reasoning is also present in the original formulation of Hoare logic [21] with the substitution treatment of assignment, but is lost if heap-based data structure, and thus aliasing, is introduced to the programming language. This loss of locality is noted as the pointer swing problem by Hoare and He [22]. Due to local reasoning, in our system, a precondition guarantees the existence of all memory locations that the procedure accesses. Hence, we can assume that *only* the heap memory specified in the precondition of each method may be modified by the method's body. This makes specifications using separation logic shorter by omitting the need to write **modifies** clauses that are necessary in traditional specification languages, such as JML [37] or Spec♯[1].

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

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

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 root. The nodes accessible via the prev field of the root node are not part of the dll list. This example also highlights some shortcuts we may use to make

shape specifications shorter. We use underscore _ to denote an anonymous variable. All the variables (including anonymous variables) in the RHS of the shape definition, which are not parameters of the given predicate, such as q, are existentially quantified. Furthermore, terms may be directly written as arguments of shape predicate or data node, while the root parameter on the LHS can be omitted as it is an implicit parameter that must be present for each of our predicate definitions.

User-definable shape predicates provide us with more flexibility than some recent automated reasoning systems [3,5] 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* and *bag* properties. (Examples with bag properties will be described later in Sec 2.2.1.) This capability enables many applications, including those requiring the support for data structures with more complex 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.

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

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 insertion sort algorithm mentioned earlier (see Sec 2.1 for the code) :

$$\begin{array}{l}\texttt{node insert(node x, node vn) where}\\ \quad \texttt{x::sortl}\langle \texttt{n}, \texttt{mi}, \texttt{ma}\rangle * \texttt{vn::node}\langle \texttt{v}, \_\rangle \ *{\mapsto}\\ \quad \texttt{res::sortl}\langle \texttt{n}{+}1, \texttt{min(v, mi)}, \texttt{max(v, ma)}\rangle\end{array} \qquad \begin{array}{l}\texttt{node insertion\_sort(node y)}\\ \quad \texttt{where y::ll}\langle \texttt{n}\rangle \wedge \texttt{n}{>}0 \ *{\mapsto}\\ \quad \texttt{res::sortl}\langle \texttt{n}, \_, \_\rangle\end{array}$$

Note that we use $\Phi_{pr} *{\mapsto} \Phi_{po}$ to capture a precondition Φ_{pr} and a postcondition Φ_{po} of a method, as an abbreviation of the standard representation requires Φ_{pr}; ensures Φ_{po} [1,37]. A special identifier res is used in the postcondition to denote the result of the method. Later in the verification system, we also use it to denote the value of the latest expression. The precondition of insertion_sort ensures that y points to a non-empty singly linked list (the fact that the list is non-empty is given by the constraint $\texttt{n}{>}0$), whereas the postcondition shows that the output list is sorted and has the same number of nodes, n, as the input list. Regarding the insert method, the precondition assumes that the method takes a sorted list of size n pointed by x, and a node vn that is to be inserted in the correct location in the sorted list. The postcondition asserts that the method returns a pointer to a sorted list of size $\texttt{n}{+}1$, whose minimum stored value is the minimum between the smallest value before the insertion, mi, and the newly inserted value, v. Similarly, the maximum stored value is the maximum between the largest value before the insertion, ma, and the newly inserted value, v.

The separation formulas we use are in a disjunctive normal form (eg. Φ, Φ_{pr}, Φ_{po} in Figure 2). 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/disequality γ, Presburger arithmetic s, ϕ ([49]) and bag constraint φ, ϕ. Furthermore, Δ denotes a composite formula that could always be safely translated into the Φ form which captures a disjunct of heap states, denoted by κ, that are in separation conjunction. [4] Δ will be used in the rest of the paper for denoting an abstract

[4] This translation is elaborated later in Figure 5.

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

Figure 2. The Specifications

state. The constraint domains ϕ for properties are currently chosen, due to the availability of the corresponding solvers. However, we envisage the use of more complex constraint domains in the future, with the adoption of new constraint solvers/provers in our system. In the rest of the paper, we will use the following bag operators [55]: bag union $\sqcup$, bag intersection $\sqcap$, bag subsumption $\sqsubset$, and bag cardinality $|\mathtt{B}|$.

As we have already seen, separation formulas 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 given by the following definitions:

Definition 2.1 (Accessible) *A variable is* accessible *if it is a method parameter, or it is a special variable, either* `root` *or* `res`.

Definition 2.2 (Reachable) *Given a heap constraint κ and a pointer constraint γ, the heap nodes in κ that are reachable from a set of pointers* $\mathtt{S}$ *can be computed by the following function.*

$$
\begin{array}{lll}
\texttt{reach}(\kappa, \gamma, \mathtt{S}) & =_{\mathit{df}} & \mathtt{p}::\mathtt{c}\langle \mathtt{v}^* \rangle * \texttt{reach}(\kappa - (\mathtt{p}::\mathtt{c}\langle \mathtt{v}^* \rangle), \gamma, \mathtt{S} \cup \{\mathtt{v} | \mathtt{v} \in \{\mathtt{v}^*\}, \mathit{IsPtr}(\mathtt{v})\}) \\
 & & \texttt{if } \exists \mathtt{q} \in \mathtt{S} \cdot (\gamma \Longrightarrow \mathtt{p}{=}\mathtt{q}) \ \wedge \ \mathtt{p}::\mathtt{c}\langle \mathtt{v}^* \rangle \in \kappa \\
\texttt{reach}(\kappa, \gamma, \mathtt{S}) & =_{\mathit{df}} & \texttt{emp, otherwise}
\end{array}
$$

Note that $\kappa - (\mathtt{p}::\mathtt{c}\langle \mathtt{v}^* \rangle)$ removes a term $\mathtt{p}::\mathtt{c}\langle \mathtt{v}^* \rangle$ from κ, while $\mathit{IsPtr}(\mathtt{v})$ determines if $\mathtt{v}$ is of pointer type.

For illustration, consider the example given below:

$$
\begin{array}{lll}
\texttt{reach}(\mathtt{p}::\texttt{node}\langle \_, \mathtt{q} \rangle * \mathtt{q}::\texttt{ll}\langle \mathtt{n} \rangle, \texttt{true}, \{\mathtt{p}\}) & =_{\mathit{df}} & \mathtt{p}::\texttt{node}\langle \_, \mathtt{q} \rangle * \texttt{reach}(\mathtt{q}::\texttt{ll}\langle \mathtt{n} \rangle, \texttt{true}, \{\mathtt{p}, \mathtt{q}\}) \\
 & =_{\mathit{df}} & \mathtt{p}::\texttt{node}\langle \_, \mathtt{q} \rangle * \mathtt{q}::\texttt{ll}\langle \mathtt{n} \rangle
\end{array}
$$

Definition 2.3 (Well-Formed Formulas) *A separation formula is well-formed if*

- *it is in a disjunctive normal form* $\bigvee(\exists v^* \cdot \kappa_i \wedge \gamma_i \wedge \phi_i)^*$ *where* κ_i *is for heap formula, and* $\gamma_i \wedge \phi_i$ *is for pure, i.e. heap-independent, formula, and*
- *all occurrences of heap nodes are reachable from its accessible variables,* $\mathtt{S}$*. That is, we have* $\forall i \cdot \kappa_i = \mathtt{reach}(\kappa_i, \gamma_i, \mathtt{S})$*, modulo associativity and commutativity of the separation conjunction* $*$*.*

For example, consider the separation formula $\mathtt{p_1}::\mathtt{node}\langle\_, \mathtt{null}\rangle * \mathtt{p_2}::\mathtt{node}\langle\_, \mathtt{null}\rangle$ and the set of accessible variables $\mathtt{S}=\{\mathtt{p_1}\}$. The formula is not well-formed as $\mathtt{p_2}$ is not reachable from $\mathtt{p_1}$.

$$\mathtt{reach}(\mathtt{p_1}::\mathtt{node}\langle\_, \mathtt{null}\rangle * \mathtt{p_2}::\mathtt{node}\langle\_, \mathtt{null}\rangle, \mathtt{true}, \{\mathtt{p_1}\}) \quad =_{df} \quad \mathtt{p_1}::\mathtt{node}\langle\_, \mathtt{null}\rangle$$

In our specifications, we allow `root` to appear only in predicate bodies, and `res` in post-conditions. 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 nodes from the consequent with nodes from the antecedent of an entailment relation.

Arbitrary recursive shape relations 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 2.4 (Well-Founded Predicates) *A shape predicate is said to be* well-founded *if it satisfies the following conditions:*

- *its body is a well-formed formula,*
- *for all heap nodes* $p::c\langle v^*\rangle$ *occurring in the body,* c *is a data type name iff* $p = \mathtt{root}$*.*

Note that the definitions above are syntactic and can easily be enforced. An example of well-founded shape predicates is `avl` - binary tree with near balanced heights, as follows :

$$\begin{array}{l}\mathtt{avl}\langle \mathtt{n}, \mathtt{h}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{n}{=}0 \wedge \mathtt{h}{=}0) \\ \quad \vee (\mathtt{root}::\mathtt{node2}\langle\_, \mathtt{p}, \mathtt{q}\rangle * \mathtt{p}::\mathtt{avl}\langle \mathtt{n_1}, \mathtt{h_1}\rangle * \mathtt{q}::\mathtt{avl}\langle \mathtt{n_2}, \mathtt{h_2}\rangle \\ \quad \wedge \mathtt{n}{=}1{+}\mathtt{n_1}{+}\mathtt{n_2} \wedge \mathtt{h}{=}1{+}\mathtt{max}(\mathtt{h_1}, \mathtt{h_2}) \wedge -1{\leq}\mathtt{h_1}{-}\mathtt{h_2}{\leq}1) \quad \mathbf{inv}\ \mathtt{n}, \mathtt{h}{\geq}0\end{array}$$

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

$$\begin{array}{lcl}\mathtt{foo}\langle \mathtt{n}\rangle & \equiv & \mathtt{root}::\mathtt{foo}\langle \mathtt{m}\rangle \wedge \mathtt{n}{=}\mathtt{m}{+}1 \\ \mathtt{goo}\langle\rangle & \equiv & \mathtt{root}::\mathtt{node}\langle\_, \_\rangle * \mathtt{q}::\mathtt{goo}\langle\rangle \\ \mathtt{too}\langle\rangle & \equiv & \mathtt{root}::\mathtt{node}\langle\_, \mathtt{q}\rangle * \mathtt{q}::\mathtt{node}\langle\_, \_\rangle\end{array}$$

For `foo`, the `root` identifier is bound to a shape predicate. For `goo`, the heap node pointed by `q` is *not* reachable from variable `root`. For `too`, an extra data node is bound to a non-`root` 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 illustrates the syntactic restriction imposed to facilitate termination of proof reasoning, which can be easily overcome by introducing intermediate predicates. For example, we may use:

$$\begin{array}{lcl}\mathtt{too}\langle\rangle & \equiv & \mathtt{root}::\mathtt{node}\langle\_, \mathtt{q}\rangle * \mathtt{q}::\mathtt{tmp}\langle\rangle \\ \mathtt{tmp}\langle\rangle & \equiv & \mathtt{root}::\mathtt{node}\langle\_, \_\rangle\end{array}$$

where `tmp` is the intermediate predicate added to satisfy our well-founded condition.

Our specification language allows bag/multiset properties to be specified in shape predicates and method specifications. This extra expressivity will be illustrated next by some examples.

2.2.1. Bag of Values/Addresses

The earlier specification of sorting captures neither the in-situ reuse of memory cells nor the fact that all the elements of the list are preserved by sorting. The reason is that the shape predicate captures only pointers and numbers but does not capture the set of reachable nodes in a heap predicate. A possible solution to this problem is to extend our specification mechanism to capture either a set or a bag of values. For generality and simplicity, we propose to only use the bag (or multi-set) notation that permits duplicates, though set notation could also be supported. The shape specifications from the previous section are revised as follows:

$$\begin{array}{l}\texttt{ll2}\langle \texttt{n}, \texttt{B}\rangle \equiv (\texttt{root}{=}\texttt{null}\wedge\texttt{n}{=}0\wedge\texttt{B}{=}\{\}) \\ \quad \vee(\texttt{root}{::}\texttt{node}\langle \_, \texttt{q}\rangle{*}\texttt{q}{::}\texttt{ll2}\langle \texttt{n}{-}1, \texttt{B}_1\rangle\wedge\texttt{B}{=}\texttt{B}_1{\sqcup}\{\texttt{root}\}) \ \textbf{inv}\ \texttt{n}{\geq}0\wedge|\texttt{B}|{=}\texttt{n}\end{array}$$

$$\begin{array}{l}\texttt{sortl2}\langle \texttt{B}, \texttt{mi}, \texttt{ma}\rangle \equiv (\texttt{root}{::}\texttt{node}\langle \texttt{mi}, \texttt{null}\rangle\wedge\texttt{mi}{=}\texttt{ma}\wedge\texttt{B}{=}\{\texttt{root}\}) \\ \quad \vee (\texttt{root}{::}\texttt{node}\langle \texttt{mi}, \texttt{q}\rangle{*}\texttt{q}{::}\texttt{sortl2}\langle \texttt{B}_1, \texttt{k}, \texttt{ma}\rangle\wedge\texttt{B}{=}\texttt{B}_1{\sqcup}\{\texttt{root}\}\wedge\ \texttt{mi}{\leq}\texttt{k}) \\ \quad \textbf{inv}\ \texttt{mi}{\leq}\texttt{ma}\wedge\texttt{B}{\neq}\{\}\end{array}$$

Each predicate of the form $\texttt{ll2}\langle \texttt{n}, \texttt{B}\rangle$ or $\texttt{sortl2}\langle \texttt{B}, \texttt{mi}, \texttt{ma}\rangle$ now captures a bag of addresses B for all the data nodes of its data structure (or heap predicate). With this extension, we can provide a more comprehensive specification for in-situ sorting, as follows :

$$\begin{array}{l}\texttt{node insert(node x, node vn) where} \\ \quad \texttt{x}{::}\texttt{sortl2}\langle \texttt{B}, \texttt{mi}, \texttt{ma}\rangle * \texttt{vn}{::}\texttt{node}\langle \texttt{v}, \_\rangle \ {*}{\mapsto} \\ \quad \texttt{res}{::}\texttt{sortl2}\langle \texttt{B}{\sqcup}\{\texttt{vn}\}, \texttt{min}(\texttt{v}, \texttt{mi}), \texttt{max}(\texttt{v}, \texttt{ma})\rangle \ \{\cdots\} \\ \texttt{node insertion\_sort(node y) where} \\ \quad \texttt{y}{::}\texttt{ll2}\langle \texttt{n}, \texttt{B}\rangle \wedge \texttt{B}{\neq}\{\} \ {*}{\mapsto}\ \texttt{res}{::}\texttt{sortl2}\langle \texttt{B}, \_, \_\rangle \ \{\cdots\}\end{array}$$

The precondition of `insert` assumes that the method takes a sorted list pointed by `x` and a node `vn` that is to be inserted in the correct location in the sorted list. The addresses of all nodes stored in the list pointed by `x` are contained in the bag `B`, whereas the minimum and maximum values are represented by `mi` and `ma`, respectively. The postcondition asserts that the method returns a pointer to a sorted list containing all nodes from the initial list, `B`, union with the new node inserted, `vn`. In the resulted list, the minimum value stored is the minimum between the smallest value before the insertion, `mi`, and the newly inserted value, `v`. Similarly, the maximum value stored is the maximum between the largest value before the insertion, `ma`, and the newly inserted value, `v`. The precondition of `insertion_sort` ensures that `y` points to a non-empty singly linked list (the fact that the list is non-empty is given by the constraint $\texttt{B}{\neq}\{\}$), whereas the postcondition shows that the output list is sorted and contains the same nodes `B` as the input list. We stress that this bag mechanism to capture the reachable nodes in a shape predicate is quite general. For example, instead of heap addresses, we may also revise our linked list view to capture a bag of reachable values, and its length, as follows:

$$\begin{array}{l}\texttt{ll3}\langle \texttt{n}, \texttt{B}\rangle \equiv (\texttt{root}{=}\texttt{null}\wedge\texttt{n}{=}0\wedge\texttt{B}{=}\{\})\vee \\ \quad (\texttt{root}{::}\texttt{node}\langle \texttt{a}, \texttt{q}\rangle{*}\texttt{q}{::}\texttt{ll3}\langle \texttt{n}{-}1, \texttt{B}_1\rangle\wedge\texttt{B}{=}\texttt{B}_1{\sqcup}\{\texttt{a}\}) \quad \textbf{inv}\ \texttt{n}{\geq}0 \wedge |\texttt{B}|{=}\texttt{n}\end{array}$$

Capturing a bag of values allows us to reason about the collection of values in a data structure, and permits relevant properties to be specified and automatically verified (when equipped with

an appropriate constraint solver), as highlighted by two examples below:

```
data pair{node v1; node v2}
pair partition(node x, int p) where
x::ll3⟨n, A⟩ ∗↦ res::pair⟨r1, r2⟩ ∗ r1::ll3⟨n1, B1⟩∗r2::ll3⟨n2, B2⟩
∧A=B1⊔B2 ∧ n=n1 + n2 ∧ (∀a∈B1·a≤p)∧(∀a∈B2·a>p)
{ if (x=null) then new pair(null, null)
  else { pair t; t:=partition(x.next, p);
         if (x.val≤p) then { x.next:=t.v1; t.v1:=x }
                      else { x.next:=t.v2; t.v2:=x };
         t } }

bool allPos(node x) where
   x::ll3⟨n, B⟩ ∗↦ x::ll3⟨n, B⟩ ∧ ((∀a∈B·a≥0)∧res ∨ (∃a∈B·a<0)∧¬res)
{ if (x=null) then true
  else if (x.val<0) then false else allPos(x.next) }
```

Note that both universal and existential properties over bags can be expressed. The first example returns a pair of lists that have been partitioned from a single input list according to an integer pivot. This partition function and its pre/post specification can be used to prove the total correctness of the quicksort algorithm. The second example uses existentially and universally quantified formulae to determine if at least one negative number is present in an input list, or not. Note that the postcondition of `allPos` preserves the fact that `x` is still pointing to a singly-linked list with the length `n` and the bag/set of values `B`: $\texttt{x::ll3}\langle \texttt{n}, \texttt{B}\rangle$. These expressive specifications can be handled by our separation logic prover in conjunction with relevant classical provers, such as MONA [45] and Isabelle [30].

3. Automated Verification

An overview of our automated verification system is given in Figure 3. The front-end of the system is a standard Hoare-style forward verifier, which invokes the entailment prover. In this section, we present the forward verifier which comprises a set of forward verification rules to systematically check that the precondition is satisfied at each call site, and that the declared postcondition is successfully verified (assuming the given precondition) for each method definition. Note that we allow the precondition of a method to be `false`. The body of any such method can always be successfully verified. However, such a method must not be invoked by a program at locations that are possibly reachable, as otherwise such program can never be verified. This relaxation does not affect the soundness of our verification system. The back-end entailment prover will be given in Sec 4.

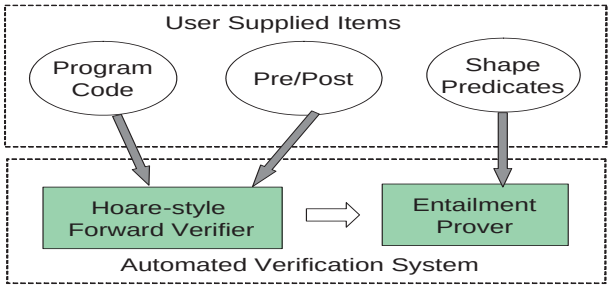

Figure 3: Our Approach to Verification

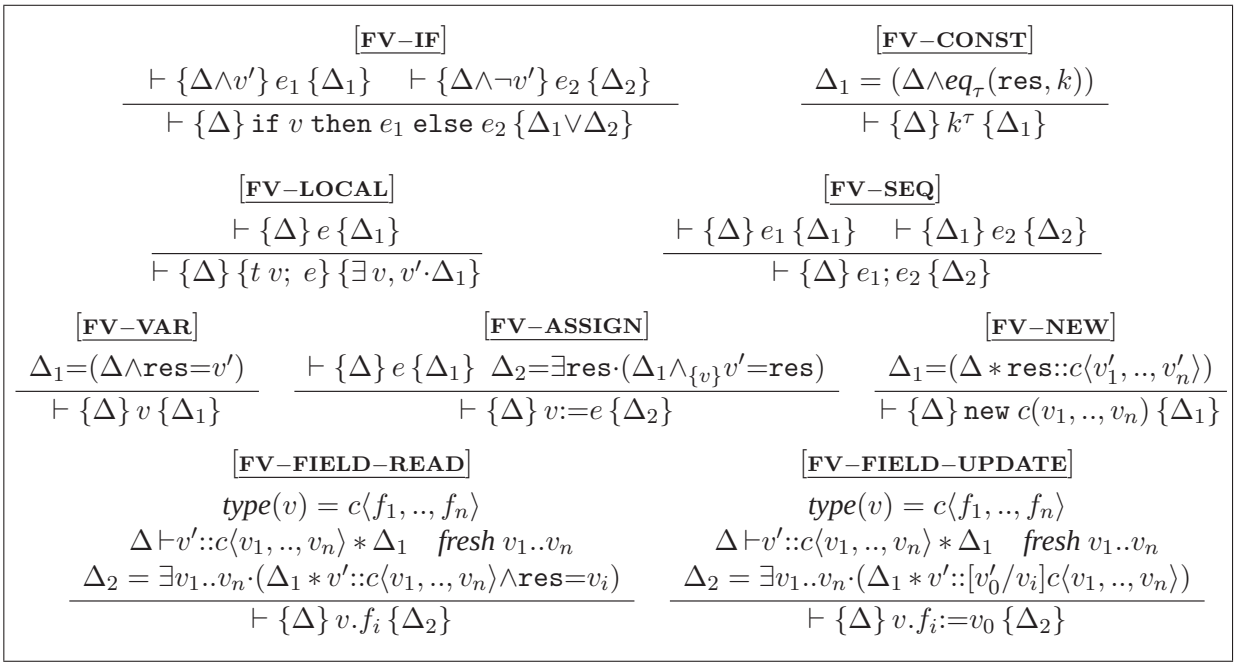

Figure 4. Some Forward Verification Rules

3.1. Forward Verifier

We use P to denote the program being checked. With pre/post conditions declared for each method in P, we can apply modular verification to a method's body using Hoare-style triples $\vdash \{\Delta_1\}\ e\ \{\Delta_2\}$. These are *forward verification* rules that expect Δ_1 to be given before computing Δ_2. Note that in our system, each abstract state (e.g. Δ_1, Δ_2) may contain both unprimed and primed versions of program variables (e.g. $\mathtt{x}, \mathtt{x}'$), where unprimed version ($\mathtt{x}$) denotes the initial value and primed version ($\mathtt{x}'$) represents the latest value of the variable. Auxiliary logical variables only appear as unprimed.

We now explain the operators/functions used in our verification rules. We first define a *composition with update* operator. 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 $\mathbf{op}_X$ is defined as:

$$\Delta_1\ \mathbf{op}_X\ \Delta_2 =_{df} \exists\ r_1..r_n \cdot (\rho_1\ \Delta_1)\ \mathbf{op}\ (\rho_2\ \Delta_2)$$
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 $\mathbf{op}$ is either $\wedge$ or $*$. To illustrate the operator, consider the following example. Suppose variable $\mathtt{x}$ is initialized by a program to 1, which is represented by

$$\mathtt{x}{=}1 \wedge \mathtt{x}'{=}1$$

The program executes the assignment $\mathtt{x}{:=}\mathtt{x}{+}2$. The updated state is computed by

$$(\mathtt{x}{=}1 \wedge \mathtt{x}'{=}1) \wedge_{\{\mathtt{x}\}} (\mathtt{x}'{=}\mathtt{x}{+}2) \equiv (\exists \mathtt{r}_1.\mathtt{x}{=}1 \wedge \mathtt{r}_1{=}1 \wedge \mathtt{x}'{=}\mathtt{r}_1{+}2) \equiv (\mathtt{x}{=}1 \wedge \mathtt{x}'{=}3)$$

which correctly reflects both the initial state and the updated state. Instances of this operator will be used in the verification rule for assignment (as $\wedge_{\{v\}}$ in [FV–ASSIGN]) and in the verification rule for method invocation (as $*_{V \cup W}$ in [FV–CALL]).

An equality operator $\texttt{eq}_\tau$ (to be used in the rule for constant expressions [FV–CONST]) converts boolean constants and `null` to their corresponding integer values, but ignores floating point constants. The function *prime*(V) returns the primed form of all variables in V. The function *nochange*(V) returns a formula asserting that the unprimed and primed versions of each variable in V are equal. These two functions will be used in the verification rule for a method declaration ([FV–METH]). The notation $[e^*/v^*]$ used in a few rules represents substitutions of v^* by e^*. A special case is $[0/\texttt{null}]$, which denotes replacement of `null` by 0. We will use the variable `P` later in the verification rule for method invocation ([FV–CALL]) to denote the entire program and it is used primarily to retrieve method declarations. As mentioned in last section, we use the special identifier `res` to denote the value of the latest expression during the verification.

Normalization rules for separation formulae are given in Figure 5. Note that the separation conjunction operator $*$ is commutative, associative, and distributive over disjunction. In separation logic, the separation conjunction between a formula and a pure (i.e. heap independent) formula is logically equivalent to a normal conjunction, i.e., $\Delta * \pi = \Delta \wedge \pi$ [51]. This justifies the third translation rule.

$(\Delta_1 \vee \Delta_2) \wedge \pi$	$\rightsquigarrow (\Delta_1 \wedge \pi) \vee (\Delta_2 \wedge \pi)$	$(\exists x \cdot \Delta) \wedge \pi$	$\rightsquigarrow \exists y \cdot ([y/x]\Delta \wedge \pi)$
$(\Delta_1 \vee \Delta_2) * \Delta$	$\rightsquigarrow (\Delta_1 * \Delta) \vee (\Delta_2 * \Delta)$	where variable y	is fresh not present in π
$(\kappa_1 \wedge \pi_1) * (\kappa_2 \wedge \pi_2)$	$\rightsquigarrow (\kappa_1 * \kappa_2) \wedge (\pi_1 \wedge \pi_2)$	$(\exists x \cdot \Delta_1) * \Delta_2$	$\rightsquigarrow \exists y \cdot ([y/x]\Delta_1 * \Delta_2)$
$(\gamma_1 \wedge \phi_1) \wedge (\gamma_2 \wedge \phi_2)$	$\rightsquigarrow (\gamma_1 \wedge \gamma_2) \wedge (\phi_1 \wedge \phi_2)$	where variable y	is fresh not present in Δ_2
$(\kappa_1 \wedge \pi_1) \wedge (\pi_2)$	$\rightsquigarrow \kappa_1 \wedge (\pi_1 \wedge \pi_2)$		

Figure 5. Normalization Rules to the Φ-form

A part of the forward verification rules are given in Figure 4. They are used to track heap states as precisely as possible using path-sensitivity (the conditional rule [FV–IF]), flow-sensitivity (the sequencing rule [FV–SEQ]), and context sensitivity (the method invocation rule [FV–CALL]).

Methods are verified using the rule [FV–METH], given below.

$$\frac{V{=}\{v_m..v_n\}\ \ W{=}\mathit{prime}(V)\ \ \Delta{=}\Phi_{pr}{\wedge}\mathit{nochange}(V)\ \ \vdash \{\Delta\}\, e\, \{\Delta_1\}\ \ (\exists W{\cdot}\Delta_1){\vdash}\Phi_{po} * \Delta_2}{\vdash t_0\ \mathit{mn}(\texttt{ref}\ t_1\ v_1, .., \texttt{ref}\ t_{m-1}\ v_{m-1}, t_m\ v_m, .., t_n\ v_n)\ \texttt{where}\ \Phi_{pr} \mathrel{*\!\!\rightarrow} \Phi_{po}\ \{e\}}\ [\text{FV–METH}]$$

In order to verify a method's body, the verifier assumes the method's precondition. Furthermore, the *nochange* function initializes the current values of parameters to their initial (unprimed) values, since each abstract state in our verification uses primed variables to denote the latest (current) values of program variables and the precondition Φ_{pr} is given only in terms of unprimed variables. The initial assumption Δ is then propagated through the body e of the procedure. At the end of the procedure, the current (primed) values of the pass-by-value parameters are existentially quantified from the poststate Δ_1, so that their values are not visible by the postcondition, hence by callers of the procedure. A method postcondition may capture

only part of the heap at the end of the method, leaving some leaked heap nodes in Δ_2, if any. For the case of a programming language with garbage collector, these leaked memory nodes do not pose any problem, as they can be automatically recovered at runtime. For a programming language without garbage collector, the information contained in the formula Δ_2 would be useful for memory leaks detection, which, as an orthogonal issue to the properties we verify in this paper, has not been incorporated into our current system. We can disallow such memory leaks by requiring the heap part of the formula Δ_2 to be $\texttt{emp}$.

When a procedure is called, the rule [FV–CALL] ensures that its precondition is satisfied at the call site. The pass-by-value parameters, V, are equated to their initial values through the *nochange* function, as their final values are not visible to the method's callers. Afterwards, the residual heap state, Δ_1, from checking procedure's precondition is composed with its postcondition to become the poststate, Δ_2, of the procedure call.

$$\frac{\begin{array}{c}[\textbf{FV–CALL}]\\ t\ mn(\texttt{ref}\ (t_i\ v_i)_{i=1}^{m-1}, (t_j\ v_j)_{j=m}^{n})\ \texttt{where}\ \Phi_{pr} \ast\!\!\rightarrow \Phi_{po}\ \{e\} \in \texttt{P} \qquad V{=}\{v_m..v_n\}\\ W{=}\{v_1..v_{m-1}\} \quad \rho{=}[v'_k/v_k]_{k=1}^{n} \quad \Delta{\vdash}\rho\Phi_{pr} * \Delta_1 \quad \Delta_2{=}((\Delta_1 \wedge \mathit{nochange}(V)) \ast_{V\cup W} \Phi_{po})\end{array}}{\vdash \{\Delta\}\ mn(v_1, .., v_{m-1}, v_m, .., v_n)\ \{\Delta_2\}}$$

For each shape definition, [FV–PRED] checks that its given invariant is a consequence of the well-founded heap formula.

$$\frac{\begin{array}{c}[\textbf{FV–PRED}]\\ \mathit{XPure}_0(\Phi) \Longrightarrow [0/\texttt{null}](\pi)\end{array}}{\vdash c\langle v^*\rangle = \Phi\ \texttt{inv}\ \pi}$$

As it will be explained in Sec 4, the entailment between separation formulae is reduced to entailment between pure formulae by successively removing heap nodes from the consequent. When the consequent is pure, the heap formula in the antecedent can be soundly approximated by function XPure_n, which translates a given separation formula to its pure counterpart. By an extra unfolding of its predicates, XPure_{n+1} function could give a more precise approximation than XPure_n. The formalization for XPure_n will be presented in Sec 4 (Fig. 6). For illustration, we explain how [FV–PRED] rule is used to justify the invariant $\texttt{n}{\geq}\texttt{0}$ for the predicate $\texttt{ll}$ given in Sec 2. Let Φ be the body of the $\texttt{ll}$ predicate, i.e. $\Phi \equiv (\texttt{root}{=}\texttt{null}{\wedge}\texttt{n}{=}\texttt{0}) \vee (\texttt{root::node}\langle \_, \texttt{r}\rangle{*}\texttt{r::ll}\langle \texttt{n}{-}\texttt{1}\rangle)$. Briefly, for $\texttt{n}{=}\texttt{0}$, XPure_0 uses the definition of the $\texttt{ll}$ predicate, replaces all occurrences of $\texttt{null}$ by $\texttt{0}$ (so that the implication check can be passed to a pure logic solver):

$$\begin{array}{rcl}\mathit{XPure}_0(\Phi) & =_{df} & \textbf{ex}\ \texttt{j} \cdot (\texttt{root}{=}\texttt{0}{\wedge}\texttt{n}{=}\texttt{0}){\vee}(\texttt{root}{=}\texttt{j}{\wedge}\texttt{j}{>}\texttt{0} \wedge \mathit{XPure}_0(\texttt{r::ll}\langle \texttt{n}{-}\texttt{1}\rangle))\\ & \equiv & \textbf{ex}\ \texttt{j} \cdot (\texttt{root}{=}\texttt{0}{\wedge}\texttt{n}{=}\texttt{0}){\vee}(\texttt{root}{=}\texttt{j}{\wedge}\texttt{j}{>}\texttt{0}{\wedge}\texttt{n}{-}\texttt{1}{\geq}\texttt{0})\end{array}$$

Note that the construct $\textbf{ex}\ \texttt{j}$ captures a symbolic address $\texttt{j}$ that has been abstracted from the heap node $\texttt{root::node}\langle \_, \texttt{r}\rangle$. Now, that we computed $\mathit{XPure}_0(\Phi)$, we can check that the invariant is a consequence of the heap formula.

$$\begin{array}{rl} & (\mathit{XPure}_0(\Phi){\Rightarrow}\ [\texttt{0}/\texttt{null}]\texttt{n}{\geq}\texttt{0})\\ \equiv & (\textbf{ex}\ \texttt{j} \cdot ((\texttt{root}{=}\texttt{0}{\wedge}\texttt{n}{=}\texttt{0}) \vee (\texttt{root}{=}\texttt{j}{\wedge}\texttt{j}{>}\texttt{0}{\wedge}\texttt{n}{-}\texttt{1}{\geq}\texttt{0})){\Rightarrow}\ \texttt{n}{\geq}\texttt{0})\end{array}$$

The soundness of the forward verification is formulated in Sec 5.

4. Entailment Prover

Proof obligations generated by software verification systems are typically discharged by a theorem prover, or a combination of theorem provers. For instance, ESC/Java [18] uses Simplify [15]; Spec$^\sharp$ [1] is compiled to Boogie [1], which in turn uses Simplify and, recently, Z3 [14]; Jahob [32,33] uses combinations of multiple theorem provers by its own combination approach.

Verification conditions generated by software verifiers typically involve multiple theories. There are a number of different approaches to processing logical formulas involving multiple theories. Nelson-Oppen is a well-known approach for combining quantifier-free formulas in stably infinite theories over disjoint signatures (theories not sharing function or predicate symbols) [43]. Simplify [15] and CVC [56] are two widely used implementations of the approach. Another approach is Satisfiability Modulo Theories (SMT) [2]. This approach tries to decide whether a formula ϕ is satisfiable with respect to background theories for which specialized decision procedures exist. Z3 is an efficient implementation based on this approach [14].

Our formulas are a combination of separation logic and heap-independent logics. None of the popular existing approaches is tailored for combinations involving separation logic. Our approach is designed to effectively handle an important fragment of the combined logic that commonly arises in practical software verification problems. As shown in the verification rules in Sec 3, our verification system generates the entailment relation of formulas, abbreviated as *heap entailment*, of the form

$$\Delta_A \vdash^{\kappa}_{V} \Delta_C * \Delta_R \quad (4.1)$$

which is shortcut for

$$\kappa * \Delta_A \vdash \exists V \cdot (\kappa * \Delta_C) * \Delta_R \quad (4.2)$$

Our entailment prover deals with such heap entailments. To prove the heap entailment (4.1) is to check whether heap nodes in the antecedent Δ_A are sufficiently precise to cover all nodes from the consequent Δ_C, and (in case they are) to compute a *residual heap state* Δ_R (also known as "frame" in the frame inference [7]), which represents what was not consumed from the antecedent after matching up with the formula from the consequent. κ is the history of nodes from the antecedent that have been used to match nodes from the consequent, V is the list of existentially quantified variables from the consequent. Note that κ and V are derived during the entailment proof. The entailment checking procedure is initially invoked with $\kappa = \texttt{emp}$ and $V = \emptyset$. The entailment proving rules are explained in the rest of the section. In Section 5, we will show that our entailment checking procedure is *sound*, in the sense that, if we can find a proof (and a residual heap state Δ_R) for (4.1), then the LHS of (4.2) semantically entails the RHS of (4.2), that is, all models of the LHS are also models of RHS. Our heap entailment may *fail* in that it can not find a residual heap state Δ_R for (4.1) after trying all possible entailment proving rules. In many cases, this will indicate that there does not exist Δ_R such that LHS of (4.2) semantically entails RHS of (4.2). However, since the completeness of our entailment prover is open, we cannot rule out the possibility that there is such Δ_R but our prover cannot discover it using the current set of rules. This will be addressed in future work.

We now briefly discuss the key steps that we may use in such an entailment proof. Firstly, we present the reduction from entailment between disjunctive formulas with existential quantifiers to entailment between quantifier-free conjunctive formulas.

Disjunction

An entailment with a disjunctive antecedent succeeds if both disjuncts entail the consequent. On the other hand, entailment with disjunctive consequent succeeds if either of the disjuncts succeeds.

$$\frac{[\textbf{ENT}-\textbf{LHS}-\textbf{OR}]}{\Delta_1 \vdash^{\kappa}_{V} \Delta_3 * \Delta_4 \qquad \Delta_2 \vdash^{\kappa}_{V} \Delta_3 * \Delta_5}{\Delta_1 \vee \Delta_2 \vdash^{\kappa}_{V} \Delta_3 * (\Delta_4 \vee \Delta_5)} \qquad \frac{[\textbf{ENT}-\textbf{RHS}-\textbf{OR}]}{\Delta_1 \vdash^{\kappa}_{V} \Delta_i * \Delta^R_i}{\Delta_1 \vdash^{\kappa}_{V} (\Delta_2 \vee \Delta_3) * \Delta^R_i} i \in \{2,3\}$$

Existential Quantifiers

Existentially quantified variables from the antecedent are simply lifted out of the entailment relation by replacing them with fresh variables. On the other hand, we keep track of the existential variables coming from the consequent by adding them to V.

$$\frac{[\textbf{ENT}-\textbf{RHS}-\textbf{EX}]}{\Delta_1 \vdash^{\kappa}_{V \cup \{w\}} ([w/v]\Delta_2) * \Delta \quad \mathit{fresh}\ w}{\Delta_1 \vdash^{\kappa}_{V} (\exists\, v \cdot \Delta_2) * \Delta} \qquad \frac{[\textbf{ENT}-\textbf{LHS}-\textbf{EX}]}{[w/v]\Delta_1 \vdash^{\kappa}_{V} \Delta_2 * \Delta \quad \mathit{fresh}\ w}{\exists v \cdot \Delta_1 \vdash^{\kappa}_{V} \Delta_2 * \Delta}$$

Next, we now present reduction of entailment between two quantifier-free conjunctive formulae to entailment between two pure formulae.

Consequent with Empty Heap

The base case for our entailment checker occurs when the consequent is a pure formula, in which case the [ENT–EMP] rule is applied. The rule first approximates the antecedent of the entailment, including the heap formulae that have been matched previously and kept in κ. It then invokes an off-the-shelf theorem prover to check if the approximation of the antecedent implies the heap-independent consequent. This strategy offers us the flexibility to use different logics for the pure part.

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

Matching up heap nodes from the antecedent and the consequent

The rule [ENT–MATCH] works by successively matching up heap nodes that can be proven aliased.

$$\frac{[\textbf{ENT}-\textbf{MATCH}]}{\mathit{XPure}_n(p_1 :: c\langle v^*_1 \rangle * \kappa_1 * \pi_1) \Longrightarrow p_1 = p_2 \quad \rho = [v^*_1/v^*_2] \quad \kappa_1 \wedge \pi_1 \wedge \mathit{freeEqn}(\rho, V) \vdash^{\kappa * p_1 :: c\langle v^*_1 \rangle}_{V - \{v^*_2\}} \rho(\kappa_2 \wedge \pi_2) * \Delta}{p_1 :: c\langle v^*_1 \rangle * \kappa_1 \wedge \pi_1 \vdash^{\kappa}_{V} (p_2 :: c\langle v^*_2 \rangle * \kappa_2 \wedge \pi_2) * \Delta}$$

$\mathit{XPure}_n(p_1 :: c\langle v^*_1 \rangle * \kappa_1 * \pi_1) \Longrightarrow p_1 = p_2$ checks if p_1 and p_2 can be proved to be aliased based on information in the antecedent of an entailment. If two aliased atomic heap formulas have the same name, which means they are two objects of the same type, or two instances of the same predicate, we require their components to be the same. The unification of the two aliased heap

formula is accomplished by the application of substitution ρ to the remaining of the consequent. We also remove v_2^* from the set of existentially quantified variables since variables v_2^* have been substituted away.

When a match occurs 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. These bindings play the role of parameter instantiations during forward reasoning, and can be accumulated into the antecedent to allow the subsequent program state (from residual heap state) to be aware of their instantiated values. This process is formalized by the function *freeEqn*, where V is the set of existentially quantified variables:

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

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 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.

As the matching process is incremental, we keep the successfully matched nodes from antecedent in κ for better precision. For example, consider the following entailment proof:

$$\frac{\dfrac{(((\mathtt{p}{=}\mathtt{null} \wedge \mathtt{n}{=}0) \vee (\mathtt{p}{\neq}\mathtt{null} \wedge \mathtt{n}{>}0)){\wedge}\mathtt{n}{>}0 \wedge \mathtt{m}{=}\mathtt{n}) \implies \mathtt{p}{\neq}\mathtt{null} \quad (\mathtt{input}\ \mathit{XPure}_1)}{(\mathit{XPure}_1(\mathtt{p}{::}\mathtt{ll}\langle\mathtt{n}\rangle) \wedge \mathtt{n}{>}0 \wedge \mathtt{m}{=}\mathtt{n} \implies \mathtt{p}{\neq}\mathtt{null}) \quad \Delta_R = (\mathtt{n}{>}0 \wedge \mathtt{m}{=}\mathtt{n}) \quad (\mathtt{by}\ [\textsc{ent-emp}])}}{\mathtt{n}{>}0 \wedge \mathtt{m}{=}\mathtt{n} \vdash^{\mathtt{p}::\mathtt{ll}\langle\mathtt{n}\rangle} \mathtt{p}{\neq}\mathtt{null} * \Delta_R \quad (\mathtt{by}\ [\textsc{ent-match}])}$$
$$\mathtt{p}{::}\mathtt{ll}\langle\mathtt{n}\rangle \wedge \mathtt{n}{>}0 \vdash \mathtt{p}{::}\mathtt{ll}\langle\mathtt{m}\rangle \wedge \mathtt{p}{\neq}\mathtt{null} * \Delta_R$$

Had the predicate $\mathtt{p}{::}\mathtt{ll}\langle\mathtt{n}\rangle$ not been kept and used, the proof would not have succeeded since we require this predicate and $\mathtt{n}{>}0$ to determine that $\mathtt{p}{\neq}\mathtt{null}$. 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. Note the transfer of $\mathtt{m}{=}\mathtt{n}$ to the antecedent (and subsequently to the residual heap state Δ_R).

Apart from the matching operation, two other essential operations that may be required in an entailment proof are (1) unfolding a shape predicate and (2) folding some data nodes back to a shape predicate.

Unfolding a shape predicate in the antecedent

If a predicate instance in the antecedent is aliased with an object in the consequent, we unfold it. Unfolding basically replaces the predicate instance by its predicate definition, normalizes the resulting formula, and resumes entailment checking.

Each unfolding either exposes an object that matches the object in the consequent, or reduces the atomic heap formula in the antecedent $p_1{::}c_1\langle v_1^*\rangle$ to a pure formula. The former case results in a reduction of the consequent by using [ENT–MATCH]. In the latter case, the entailment either (i) fails immediately since the checker is unable to find an aliased heap node or, (ii) if the resulted pure formula reveals additional aliasing information, the entailment checker continues with a new aliased heap node from the antecedent. If the new aliased heap node is an object, a match occurs and thus a reduction of the consequent. Otherwise a new unfolding is called on.

This process cannot go forever as every time it happens, one predicate from the antecedent is removed and no new predicate instance is generated. Overall, the termination of the entailment checking procedure is not compromised, as we prove in Theorem 5.5.

$$\frac{\begin{array}{c}[\textsc{ent-unfold}]\\ \mathit{XPure}_n(p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{*}\pi_1){\Longrightarrow}p_1{=}p_2 \quad \mathit{IsPred}(\mathtt{c_1}){\wedge}\mathit{IsData}(\mathtt{c_2})\\ \mathit{unfold}(p_1{::}c_1\langle v_1^*\rangle){*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}(p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2)*\Delta\end{array}}{p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}(p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2)*\Delta}$$

$$\frac{\begin{array}{c}[\textsc{unfolding}]\\ c\langle v^*\rangle{\equiv}\Phi\ \mathtt{inv}\ \pi\in P\end{array}}{\mathit{unfold}(p{::}c\langle v^*\rangle)\ =_{df}\ [p/\mathtt{root}]\Phi}$$

The function *IsPred*(c) (resp. *IsData*(c)) returns `true` if c is a shape predicate (resp. a data node). For illustration, consider the following example.

$$\mathtt{x{::}ll3\langle n,B\rangle{\wedge}n{>}2}\ \vdash\ (\exists \mathtt{r}{\cdot}\mathtt{x{::}node\langle r,y\rangle{\wedge}y{\neq}null{\wedge}r}\in\mathtt{B})*\Delta_{\mathtt{R}}$$

where $\Delta_{\mathtt{R}}$ captures the residual heap state of entailment (to be computed). Note that a predicate $\mathtt{x{::}ll3\langle n,B\rangle}$ from the antecedent and a data node $\mathtt{x{::}node\langle r,y\rangle}$ from the consequent are co-related via the same variable `x`. For the entailment to succeed, we would first unfold the $\mathtt{ll3\langle n,B\rangle}$ predicate in the antecedent ([ENT−UNFOLD]):

$$\begin{array}{l}\exists \mathtt{q_1,v}\cdot\mathtt{x{::}node\langle v,q_1\rangle{*}q_1{::}ll3\langle n{-}1,B_1\rangle{\wedge}n{>}2{\wedge}B{=}B_1\sqcup\{v\}}\\ \quad\vdash(\exists \mathtt{r}{\cdot}\mathtt{x{::}node\langle r,y\rangle{\wedge}y{\neq}null}\wedge\mathtt{r}\in\mathtt{B})*\Delta_{\mathtt{R}}\end{array}$$

After removing the existential quantifiers ([ENT−RHS−OR], [ENT−LHS−OR]), we obtain:

$$\begin{array}{l}\mathtt{x{::}node\langle v,q_1\rangle{*}q_1{::}ll3\langle n{-}1,B_1\rangle{\wedge}n{>}2}\ \wedge\ \mathtt{B{=}B_1\sqcup\{v\}}\\ \quad\vdash(\mathtt{x{::}node\langle r,y\rangle{\wedge}y{\neq}null}\wedge\mathtt{r}\in\mathtt{B})*\Delta_{\mathtt{R}}\end{array}$$

The data node in the consequent is then matched up ([ENT−MATCH]), giving:

$$\mathtt{q_1{::}ll3\langle n{-}1,B_1\rangle{\wedge}n{>}2{\wedge}B{=}B_1\sqcup\{v\}{\wedge}q_1{=}y}\ \vdash\ (\mathtt{q_1{\neq}null}\wedge\mathtt{v}\in\mathtt{B})*\Delta_{\mathtt{R}}$$

Folding against a shape predicate in the consequent

If a predicate instance in the consequent does not have a matching predicate instance in the antecedent, we attempt to generate one by folding the antecedent.

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

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

When a fold against a predicate $p_2{::}c_2\langle v_2^*\rangle$ is performed, the constraints related to variables v_2^* are significant. The *split* function projects these constraints out and differentiates those constraints based on free variables. These constraints on free variables can be transferred to the antecedent to support the variables' instantiations.

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

A formal definition of folding is specified by the rule [FOLDING]. Some heap nodes from κ are removed by the entailment procedure so as to match with the heap formula of the 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 with a couple of specialised rules.

For illustration consider the following example.

$$\texttt{x::node}\langle 1,\texttt{q}_1\rangle{*}\texttt{q}_1\texttt{::node}\langle 2,\texttt{null}\rangle{*}\texttt{y::node}\langle 3,\texttt{null}\rangle \vdash\ (\texttt{x::ll3}\langle \texttt{n},\texttt{B}\rangle{\wedge}\texttt{n}{>}1{\wedge}1\in\texttt{B}) * \Delta_{\texttt{R}}$$

The data node $\texttt{x::node}\langle 1,\texttt{q}_1\rangle$ from the antecedent and the predicate $\texttt{x::ll3}\langle \texttt{n},\texttt{B}\rangle$ from the consequent are co-related by the variable x. In this case, we apply the folding operation to the first two nodes from the antecedent against the shape predicate from the consequent. After that, a matching operation is invoked since the folded predicate now matches with the predicate in the consequent.

The fold step may be recursively applied but is guaranteed to terminate for well-founded predicates 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 $\exists \texttt{n}_1,\texttt{n}_2\cdot \texttt{n}{=}\texttt{n}_1{+}1 \wedge \texttt{n}_1{=}\texttt{n}_2{+}1 \wedge \texttt{n}_2{=}0$ and $\exists \texttt{B}_1,\texttt{B}_2\cdot \texttt{B}{=}\texttt{B}_1{\cup}\{2\} \wedge \texttt{B}_1{=}\{1\}{\cup}\texttt{B}_2 \wedge \texttt{B}_2{=}\{\}$, where $\texttt{n}_1,\texttt{n}_2,\texttt{B}_1,\texttt{B}_2$ are existential variables introduced by the folding process, and are subsequently eliminated. These binding formulae may be transferred to the antecedent if n and B are free (for instantiation). Otherwise, they will be kept in the consequent. Since n and B are indeed free, our folding operation would finally derive:

$$\texttt{y::node}\langle 3,\texttt{null}\rangle \wedge \texttt{n}=2 \wedge \texttt{B}=\{1,2\} \vdash\ (\texttt{n}{>}1 \wedge 1\in\texttt{B}) * \Delta_{\texttt{R}}$$

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 n is free :

$$\texttt{z}{=}\texttt{null} \ \vdash\ \texttt{z::ll3}\langle \texttt{n},\texttt{B}\rangle \wedge \texttt{n}{>}{-}1 * \Delta_{\texttt{R}}$$

By unfolding the predicate $\texttt{ll3}\langle \texttt{n}\rangle$ in the consequent, we obtain :

$$\begin{array}{l} \texttt{z}{=}\texttt{null} \vdash (\texttt{z}{=}\texttt{null}{\wedge}\texttt{n}{=}0{\wedge}\texttt{B}=\{\}{\wedge}\texttt{n}{>}{-}1) \\ \qquad\qquad \vee(\exists \texttt{q},\texttt{v}{\cdot}\texttt{z::node}\langle \texttt{v},\texttt{q}\rangle{*}\texttt{q::ll3}\langle \texttt{n}{-}1,\texttt{B}_1\rangle{\wedge}\texttt{B}=\texttt{B}_1\cup\{\texttt{v}\}{\wedge}\texttt{n}{>}{-}1) * \Delta_{\texttt{R}} \end{array}$$

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

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

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

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

$$\frac{\mathit{IsData}(c) \quad \mathit{fresh}\ i}{\mathit{XPure}_n(p{::}c\langle v^*\rangle) =_{df} \mathbf{ex}\ i{\cdot}(p{=}i{\wedge}i{>}0)} \qquad \frac{\mathit{IsPred}(c) \quad \mathit{fresh}\ i^* \quad \mathit{Inv}_n(p{::}c\langle v^*\rangle) = \mathbf{ex}\ j^* \cdot \bigvee(\exists u^*{\cdot}\pi)^*}{\mathit{XPure}_n(p{::}c\langle v^*\rangle) =_{df} \mathbf{ex}\ i^* \cdot [i^*/j^*]\bigvee(\exists u^*{\cdot}\pi)^*}$$

Figure 6. *XPure* : Translating to Pure Form

Approximating separation formula by pure formula

In our entailment proof, the 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 this happens, the heap formula in the antecedent can be soundly approximated by function XPure_n. The index n is a parameter that indicates how precise the caller wants the approximation to be. A related function that XPure_n uses is the Inv_n function. This function, along with XPure_n, computes and updates shape predicate invariants with more precise invariants. The definition of Inv_n is given by the following rules:

$$\frac{(c\langle v^*\rangle \equiv \Phi\ \mathbf{inv}\ \pi_0) \in P}{\mathit{Inv}_0(p{::}c\langle v^*\rangle) =_{df} [p/\mathtt{root}, 0/\mathtt{null}]\pi_0} \qquad \frac{(c\langle v^*\rangle \equiv \Phi\ \mathbf{inv}\ \pi_0) \in P}{\mathit{Inv}_n(p{::}c\langle v^*\rangle) =_{df} [p/\mathtt{root}]\mathit{XPure}_{n-1}(\Phi)}$$

In the base case, when $n = 0$, Inv_n returns the user-supplied invariant. All occurrences of null are replaced by 0 so that we can pass the returned formula to a pure logic solver. Parameters of the predicates are replaced by the corresponding actuals. When $n > 0$, Inv_n invokes XPure_{n-1} to compute a more precise invariant based on the body of the predicate.

The function $\mathit{XPure}_n(\Phi)$, whose definition is given in Fig 6, returns a sound approximation of Φ as a formula of the form: $\beta ::= ((\bigvee(\exists v^*{\cdot}\pi)^*) \mid (\mathbf{ex}\ i \cdot \beta))$ [5], 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 Φ. *XPure* differentiates between symbolic addresses coming from disjoint regions of the heap described by formulas conjoined by the separating conjunction $*$:

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

where $\wedge$ is further normalized as follows:

$$(\mathbf{ex}\ I \cdot \phi_1) \wedge (\mathbf{ex}\ J \cdot \phi_2) \ \rightsquigarrow\ \mathbf{ex}\ I \cup J \cdot \phi_1 \wedge \phi_2 \wedge \textstyle\bigwedge_{i\in I, j\in J} i \neq j$$

[5]Here β is defined as either $\bigvee(\exists v^*{\cdot}\pi)^*$ or recursively as $\mathbf{ex}\ i \cdot \beta$.

We illustrate how the approximation functions work by computing $\mathit{XPure}_1(\mathtt{p}{::}\mathtt{ll}\langle\mathtt{n}\rangle)$. Let Φ be the body of the ll predicate, i.e. $\Phi \equiv (\mathtt{root}{=}\mathtt{null}{\wedge}\mathtt{n}{=}0) \vee (\mathtt{root}{::}\mathtt{node}\langle\_,\mathtt{r}\rangle{*}\mathtt{r}{::}\mathtt{ll}\langle\mathtt{n}{-}1\rangle)$.

$$
\begin{array}{lll}
\mathit{Inv}_0(\mathtt{p}{::}\mathtt{ll}\langle\mathtt{n}\rangle) & =_{df} & \mathtt{n} \geq 0 \\
\mathit{XPure}_0(\Phi) & =_{df} & \mathbf{ex}\ \mathtt{j} \cdot (\mathtt{root}{=}0{\wedge}\mathtt{n}{=}0){\vee}(\mathtt{root}{=}\mathtt{j}{\wedge}\mathtt{j}{>}0 \wedge \mathit{Inv}_0(\mathtt{r}{::}\mathtt{ll}\langle\mathtt{n}{-}1\rangle)) \\
 & = & \mathbf{ex}\ \mathtt{j} \cdot (\mathtt{root}{=}0{\wedge}\mathtt{n}{=}0){\vee}(\mathtt{root}{=}\mathtt{j}{\wedge}\mathtt{j}{>}0{\wedge}\mathtt{n}{-}1{\geq}0) \\
\mathit{Inv}_1(\mathtt{p}{::}\mathtt{ll}\langle\mathtt{n}\rangle) & =_{df} & [\mathtt{p}/\mathtt{root}]\mathit{XPure}_0(\Phi) \\
 & = & \mathbf{ex}\ \mathtt{j} \cdot (\mathtt{p}{=}0{\wedge}\mathtt{n}{=}0) \vee (\mathtt{p}{=}\mathtt{j}{\wedge}\mathtt{j}{>}0{\wedge}\mathtt{n}{-}1{\geq}0) \\
\mathit{XPure}_1(\mathtt{p}{::}\mathtt{ll}\langle\mathtt{n}\rangle) & =_{df} & \mathbf{ex}\ \mathtt{i} \cdot [\mathtt{i}/\mathtt{j}]\mathit{Inv}_1(\mathtt{p}{::}\mathtt{ll}\langle\mathtt{n}\rangle) \\
 & = & \mathbf{ex}\ \mathtt{i} \cdot (\mathtt{p}{=}0{\wedge}\mathtt{n}{=}0) \vee (\mathtt{p}{=}\mathtt{i}{\wedge}\mathtt{i}{>}0{\wedge}\mathtt{n}{-}1{\geq}0)
\end{array}
$$

The following normalization rules are also used to propagate $\mathbf{ex}$ to the leftmost :

$$
\begin{array}{ll}
(\mathbf{ex}\ I \cdot \phi_1){\vee}(\mathbf{ex}\ J \cdot \phi_2) & \rightsquigarrow \mathbf{ex}\ I{\cup}J \cdot (\phi_1 \vee \phi_2) \\
\exists\, v \cdot (\mathbf{ex}\ I \cdot \phi) & \rightsquigarrow \mathbf{ex}\ I \cdot (\exists\, v \cdot \phi)
\end{array}
$$

The $\mathbf{ex}\ i^*$ construct is converted to $\exists\, i^*$ when the formula is used as a pure formula. For instance, the above $\mathit{XPure}_1(\mathtt{p}{::}\mathtt{ll}\langle\mathtt{n}\rangle)$ is converted to $\exists \mathtt{i} \cdot (\mathtt{p}{=}0{\wedge}\mathtt{n}{=}0) \vee (\mathtt{p}{=}\mathtt{i}{\wedge}\mathtt{i}{>}0{\wedge}\mathtt{n}{-}1{\geq}0)$, which is further reduced to $(\mathtt{p}{=}0{\wedge}\mathtt{n}{=}0){\vee}(\mathtt{p}{>}0{\wedge}\mathtt{n}{-}1{\geq}0)$.

The soundness of the heap approximation (given in the next section) ensures that it is safe to approximate an antecedent by using *XPure*, starting from a given sound invariant (checked by [FV–PRED] in Sec 3). The heap approximation also allows the possibility of obtaining a more precise invariant by unfolding the definition of a predicate one or more times, prior to applying the XPure_0 approximation with the predicate's invariant. For example, when given a pure invariant $\mathtt{n}{\geq}0$ for the predicate $\mathtt{ll}\langle\mathtt{n}\rangle$, the XPure_0 approximation is simply the pure invariant $\mathtt{n}{\geq}0$ itself. However, the XPure_1 approximation would invoke a single unfold before the XPure_0 approximation is applied, yielding $\mathbf{ex}\ \mathtt{i}{\cdot}(\mathtt{root}{=}0{\wedge}\mathtt{n}{=}0 \vee \mathtt{root}{=}\mathtt{i}{\wedge}\mathtt{i}{>}0{\wedge}\mathtt{n}{-}1{\geq}0)$, which is sound and more precise than $\mathtt{n}{\geq}0$, since the former can relate the nullness of the root pointer with the size n of the list.

The invariants associated with shape predicates play an important role in our system. Without the knowledge $\mathtt{m}{\geq}0$, the proof search for the entailment $\mathtt{x}{::}\mathtt{node}\langle\_,\mathtt{y}\rangle * \mathtt{y}{::}\mathtt{ll}\langle\mathtt{m}\rangle \vdash \mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle \wedge \mathtt{n}{\geq}1$ would not have succeeded (failing to establish $\mathtt{n}{\geq}1$). Without a more precisely derived invariant using XPure_1 on predicate ll, the proof search for the entailment $\mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle \wedge \mathtt{n}{>}0 \vdash \mathtt{x}{\neq}\mathtt{null}$ would not have succeeded either.

Implicit vs Explicit Instantiations

In the preceding subsections, we have presented a technique for the *implicit instantiation* of free variables during the matching and the folding operations. This technique allows the bindings of free variables to be transferred to the antecedent during entailment proving, but kept the substitutions for existential variables within the consequent itself. This dual treatment of free and existential variables is meant to restrict the instantiation mechanism to only those which are strictly required for entailment proving.

In this subsection, we shall provide an alternative technique for the *explicit* instantiation of free variables. Our main purpose is to clarify the role of the instantiation mechanism and to provide a justification for the implicit instantiation technique being used in our current version of entailment proving. To clarify the instantiation technique, let us consider a simple data type

which carries a pair of integer values:

$$\texttt{data pair}\ \{\ \texttt{int x};\ \texttt{int y}\}$$

Let us also provide a simple method which checks if the sum of the two fields from the given pair is positive, before returning the second field as the method's result.

$$\begin{array}{l}\texttt{int foo(pair p) where}\\ \quad \exists \texttt{a} \cdot \texttt{p::pair}\langle \texttt{a}, \texttt{b}\rangle \wedge \texttt{a+b>0} \ *\!\!\rightarrow \texttt{res} = \texttt{b}\\ \{\texttt{if}\ (\texttt{p.x} + \texttt{p.y}) \leq 0\ \texttt{then error}()\ \texttt{else p.y}\}\end{array}$$

If the expected precondition does not hold, the above method raises an error by calling a special $\texttt{error}()$ primitive. Furthermore, we shall assume that the pair object is leaked (or garbage collected for some programming languages) after invoking this method. Take note that logical variables (other than program variables) that are used by *both* precondition and postcondition shall be marked as free variables, while those that are used in either precondition or postcondition alone, shall be existentially bound. For our example, logical variable $\texttt{b}$ is *free* since it is used in both precondition and postcondition. In contrast, logical variable $\texttt{a}$ is existentially *bound* since it is only used in the precondition. Our entailment prover distinguishes free from bound variables in order to decide which bindings may be propagated to the residual heap state. Let us re-visit our earlier implicit instantiation technique by examining the following entailment proof.

$$\dfrac{\dfrac{(\texttt{c} = 2 \wedge \texttt{b} = 3 \implies \exists \texttt{a} \cdot \texttt{a} = \texttt{c} \wedge \texttt{a+b>0}) \quad \Delta_{\texttt{R}} = (\texttt{c} = 2 \wedge \texttt{b} = 3)}{\texttt{c} = 2 \wedge \texttt{b} = 3 \vdash^{\texttt{p::pair}\langle 2,3\rangle}_{\{\texttt{a}\}} (\texttt{a} = \texttt{c} \wedge \texttt{a+b>0}) * \Delta_{\texttt{R}} \qquad (\text{by } [\textbf{ENT-EMP}])}}{\texttt{p::pair}\langle \texttt{c}, 3\rangle \wedge \texttt{c} = 2 \vdash \exists \texttt{a} \cdot \texttt{p::pair}\langle \texttt{a}, \texttt{b}\rangle \wedge \texttt{a+b>0} * \Delta_{\texttt{R}} \quad (\text{by } [\textbf{ENT-RHS-OR}], [\textbf{ENT-MATCH}])}$$

During matching of the $\texttt{pair}$ data nodes, the binding $\texttt{b} = 3$ is moved to the antecedent due to free variable $\texttt{b}$, while the binding $\texttt{a} = \texttt{c}$ for bound variable $\texttt{a}$ is kept in the consequent. Hence, only the instantiation of $\texttt{b} = 3$ is propagated to the residual heap state which can then be linked with the postcondition $\texttt{res} = \texttt{b}$.

We shall now propose an alternative technique for the instantiation of free variables. To do that, we introduce a new notation $(\exists \texttt{v}{:}\mathcal{I} \cdot \Delta)$ that explicitly marks $\texttt{v}$ as a variable to be instantiated. This new notation is meant for each consequent that has been taken from a method's precondition for entailment proving. For our earlier method's precondition, we can mark the free variable $\texttt{b}$, as follows: $(\exists \texttt{a} \exists \texttt{b}{:}\mathcal{I} \cdot \texttt{p::pair}\langle \texttt{a}, \texttt{b}\rangle \wedge \texttt{a+b>0})$.

With this new notation, free variables are being treated as existential variables, except that their bindings in the consequent may be transferred to the residual heap state. To incorporate this effect, we modify the rule for $\texttt{emp}$ consequent of entailment prover to:

$$\dfrac{\rho{=}[0/\texttt{null}] \quad (\mathit{XPure}_n(\kappa_1 * \kappa) \wedge \rho\pi_1 \Longrightarrow \rho \exists V \cdot \pi_2) \quad B = V - \{v | v{:}\mathcal{I} \in V\} \quad \pi_{\mathcal{I}} = (\exists B \cdot \pi_2)}{\kappa_1 \wedge \pi_1 \vdash^{\kappa}_{V} \pi_2 * (\kappa_1 \wedge (\pi_1 \wedge \pi_{\mathcal{I}}))}\ [\textbf{ENT-EMP}']$$

Note that the residual heap state will now explicitly capture the bindings for free variables that have been generated in the consequent via $\pi_{\mathcal{I}}$. Using this modified rule, we can perform entailment proving for our earlier example, as follows:

$$\frac{\begin{array}{c}(\mathtt{c}=2 \implies \exists \mathtt{a},\mathtt{b}\cdot \mathtt{a}=\mathtt{c}\wedge\mathtt{b}=3\wedge \mathtt{a}{+}\mathtt{b}{>}0) \quad \pi_{\mathcal{I}}=((\exists \mathtt{a}\cdot(\mathtt{a}=\mathtt{c}\wedge\mathtt{b}=3\wedge \mathtt{a}{+}\mathtt{b}{>}0))\\ \Delta_{\mathtt{R}}=(\mathtt{c}=2\wedge(\mathtt{b}=3\wedge \mathtt{c}{>}{-}3))\end{array}}{\mathtt{c}=2 \vdash^{\mathtt{p}::\mathtt{pair}\langle 2,3\rangle}_{\{\mathtt{a},(\mathtt{b}:\mathcal{I})\}} (\mathtt{a}=\mathtt{c}\wedge\mathtt{b}=3\wedge \mathtt{a}{+}\mathtt{b}{>}0)*\Delta_{\mathtt{R}} \quad (\text{by } [\textsc{ent-emp}'])}$$
$$\frac{}{\mathtt{p}::\mathtt{pair}\langle \mathtt{c},3\rangle\wedge \mathtt{c}=2 \vdash \exists \mathtt{a}\exists \mathtt{b}{:}\mathcal{I}\cdot \mathtt{p}::\mathtt{pair}\langle \mathtt{a},\mathtt{b}\rangle\wedge \mathtt{a}{+}\mathtt{b}{>}0*\Delta_{\mathtt{R}}\ (\text{by } [\textsc{ent-rhs-or}],[\textsc{ent-match}])}$$

This technique allows *any* free variables to be explicitly instantiated, and is slightly more general than the implicit technique which can only instantiate free variables that are present as arguments of data nodes or predicates. Nevertheless, both techniques have a similar objective of performing parameter instantiation for the precondition at each method call. Our current implementation uses implicit instantiation which is simpler and incremental, but is slightly less general than the explicit instantiation technique. As a future work, we will implement also the explicit instantiation technique and compare both techniques in more detail.

4.1. Forward Verification Example

We present the detailed verification of the first branch of the `insert` method from Sec 2. While code is in bold face, program states are inside $\{\}$. Note that program variables appear primed in formulae to denote the latest values, whereas logical variables are always unprimed.

(1). $\{\mathtt{x}'::\mathtt{sortl}\langle \mathtt{n},\mathtt{mi},\mathtt{ma}\rangle * \mathtt{vn}'::\mathtt{node}\langle \mathtt{v},\_\rangle\}$ // [FV-METH](*initialize precondition*)

$\mathbf{if}\ (\mathbf{vn.val} \leq \mathbf{x.val})\ \mathbf{then}\ \{$

(2). $\{(\mathtt{x}'::\mathtt{node}\langle \mathtt{mi},\mathtt{null}\rangle * \mathtt{vn}'::\mathtt{node}\langle \mathtt{v},\_\rangle \wedge \mathtt{mi}{=}\mathtt{ma}\wedge \mathtt{n}{=}1\wedge \mathtt{v}{\leq}\mathtt{mi})$
$\vee\ (\exists \mathtt{q},\mathtt{k}\cdot \mathtt{x}'::\mathtt{node}\langle \mathtt{mi},\mathtt{q}\rangle{*}\mathtt{q}::\mathtt{sortl}\langle \mathtt{n}{-}1,\mathtt{k},\mathtt{ma}\rangle{*}\mathtt{vn}'::\mathtt{node}\langle \mathtt{v},\_\rangle$
$\wedge \mathtt{mi}{\leq}\mathtt{k}\wedge \mathtt{mi}{\leq}\mathtt{ma}\wedge \mathtt{n}{\geq}2\wedge \mathtt{v}{\leq}\mathtt{mi})\}$ // [FV-IF], [UNFOLDING]

$\mathbf{vn.next} := \mathbf{x};$

(3). $\{(\mathtt{x}'::\mathtt{node}\langle \mathtt{mi},\mathtt{null}\rangle * \mathtt{vn}'::\mathtt{node}\langle \mathtt{v},\mathtt{x}'\rangle \wedge \mathtt{mi}{=}\mathtt{ma}\wedge \mathtt{n}{=}1\wedge \mathtt{v}{\leq}\mathtt{mi})$
$\vee\ (\exists \mathtt{q},\mathtt{k}\cdot \mathtt{x}'::\mathtt{node}\langle \mathtt{mi},\mathtt{q}\rangle * \mathtt{q}::\mathtt{sortl}\langle \mathtt{n}{-}1,\mathtt{k},\mathtt{ma}\rangle{*}\mathtt{vn}'::\mathtt{node}\langle \mathtt{v},\mathtt{x}'\rangle$
$\wedge \mathtt{mi}{\leq}\mathtt{k}\wedge \mathtt{mi}{\leq}\mathtt{ma}\wedge \mathtt{n}{\geq}2\wedge \mathtt{v}{\leq}\mathtt{mi})\}$ // [FV-FIELD-UPDATE]

$\mathbf{vn}$

(4). $\{(\mathtt{x}'::\mathtt{node}\langle \mathtt{mi},\mathtt{null}\rangle * \mathtt{vn}'::\mathtt{node}\langle \mathtt{v},\mathtt{x}'\rangle \wedge \mathtt{mi}{=}\mathtt{ma}\wedge \mathtt{n}{=}1\wedge \mathtt{v}{\leq}\mathtt{mi}\wedge \mathtt{res}{=}\mathtt{vn}')$
$\vee\ (\exists \mathtt{q},\mathtt{k}\cdot \mathtt{x}'::\mathtt{node}\langle \mathtt{mi},\mathtt{q}\rangle{*}\ \mathtt{q}::\mathtt{sortl}\langle \mathtt{n}{-}1,\mathtt{k},\mathtt{ma}\rangle * \mathtt{vn}'::\mathtt{node}\langle \mathtt{v},\mathtt{x}'\rangle$
$\wedge \mathtt{mi}{\leq}\mathtt{k}\wedge \mathtt{mi}{\leq}\mathtt{ma}{\wedge}\mathtt{n}{\geq}2\wedge \mathtt{v}{\leq}\mathtt{mi}\wedge \mathtt{res}{=}\mathtt{vn}')\}$ // [FV-VAR]

$\}$

(5). $\{\mathtt{res}::\mathtt{sortl}\langle \mathtt{n}{+}1,\mathtt{min}(\mathtt{v},\mathtt{mi}),\mathtt{max}(\mathtt{v},\mathtt{ma})\rangle\}$
// [FV-METH](*checking postcondition*), [FOLDING]

To facilitate the illustration, we label the abstract states by $(1),..,(5)$. The state (1) is obtained by initialising the precondition using the *nochange* operation in the [FV-METH] rule. This is

necessary because all abstract program states in our system contain both unprimed and primed variables, where primed variables denote the latest values of program variables and unprimed variables denote either initial values of program variables or values of logical variables. The abstract state (2) is obtained by unfolding the predicate $\mathtt{x'}::\mathtt{sortl}\langle \mathtt{n}, \mathtt{mi}, \mathtt{ma}\rangle$ and then distributing the formula $\mathtt{vn'}::\mathtt{node}\langle \mathtt{v}, \_\rangle \wedge \mathtt{v} \leq \mathtt{mi}$ over the two disjunctions obtained by unfolding. Note that $\mathtt{v} \leq \mathtt{mi}$ is obtained from the if-condition. The rule [UNFOLDING] replaces the predicate $\mathtt{x'}::\mathtt{sortl}\langle \mathtt{n}, \mathtt{mi}, \mathtt{ma}\rangle$ by its definition.

The effect of the field update $\mathbf{vn.next} := \mathbf{x}$; is recorded in state (3) by changing the heap node $\mathtt{vn'}::\mathtt{node}\langle \mathtt{v}, \_\rangle$ to $\mathtt{vn'}::\mathtt{node}\langle \mathtt{v}, \mathtt{x'}\rangle$ using the [FV–FIELD–UPDATE] rule. By the [FV–VAR] rule, the effect of the last expression $\mathbf{vn}$ in the branch is recorded in state (4) using the formula $\mathtt{res}{=}\mathtt{vn'}$. The verification of this branch finishes by proving that state (4) entails the postcondition (5) according to the [FV–METH] rule. The rule [FOLDING] used in this last step folds a formula which matches with a predicate's definition back to the predicate. In this case, it folds state (4) to state (5).

5. Soundness

In this section we formalize the soundness properties for both the forward verifier and the entailment prover.

5.1. Semantic Model

The semantics of our separation heap formula is similar to the model given for separation logic [51], except that we have extensions to handle our user-defined shape predicates.

To define the model we assume sets *Loc* of locations (positive integer values), *Val* of primitive values, with $0 \in$ *Val* denoting `null`, *Var* of variables (program and logical 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 $s, h \models \Phi$ denote the model relation, i.e. the stack s and heap h satisfy the constraint Φ, with h, s from the following concrete domains:

$$\begin{array}{lcll} h & \in & \mathit{Heaps} & =_{df} \quad \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{ObjVal} \\ s & \in & \mathit{Stacks} & =_{df} \quad \mathit{Var} \rightarrow \mathit{Val} \cup \mathit{Loc} \end{array}$$

Note that each heap h is a finite partial mapping while each stack s is a total mapping, as in the classical separation logic [51,25]. 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 relation for separation heap formulas is defined below. The model relation for pure formula $s \models \pi$ denotes that the formula π evaluates to `true` in s.

Definition 5.1 *[Model for Separation Constraint]*

$$
\begin{array}{lll}
s,h \models \Phi_1 \vee \Phi_2 & \texttt{iff} & s,h \models \Phi_1 \textit{ or } s,h \models \Phi_2 \\
s,h \models \exists v_{1..n} \cdot \kappa \wedge \pi & \texttt{iff} & (\exists \nu_{1..n} \cdot s[v_1 \mapsto \nu_1, .., v_n \mapsto \nu_n], h \models \kappa) \textit{ and } (s[v_1 \mapsto \nu_1, .., v_n \mapsto \nu_n] \models \pi) \\
s,h \models \kappa_1 * \kappa_2 & \texttt{iff} & \exists h_1, h_2 \cdot\ h_1 \# h_2 \textit{ and } h = h_1 \cdot h_2 \textit{ and} \\
 & & s, h_1 \models \kappa_1 \textit{ and } s, h_2 \models \kappa_2 \\
s,h \models \texttt{emp} & \texttt{iff} & \textit{dom}(h) = \emptyset \\
s,h \models p::c\langle v_{1..n} \rangle & \texttt{iff} & \texttt{data}\ c\ \{t_1\ f_1, .., t_n\ f_n\} \in P,\ h = [s(p) \mapsto r], \\
 & & \textit{and } r = c[f_1 \mapsto s(v_1), .., f_n \mapsto s(v_n)] \\
 & \texttt{or} & (c\langle v_{1..n} \rangle \equiv \Phi\ \texttt{inv}\ \pi) \in P \textit{ and } s,h \models [p/\texttt{root}]\Phi
\end{array}
$$

Note that $h_1 \# h_2$ indicates h_1 and h_2 are domain-disjoint, i.e. $\textit{dom}(h_1) \cap \textit{dom}(h_2) = \emptyset$. $h_1 \cdot h_2$ denotes the union of disjoint heaps h_1 and h_2. The definition for $s,h \models p::c\langle v^* \rangle$ is split into two cases: (1) c is a data node defined in the program *P*; (2) c is a shape predicate defined in the program *P*. In the first case, h has to be a singleton heap. In the second case, the shape predicate c may be inductively defined. Note that the semantics for an inductively defined shape predicate denotes an implicit notion of the least fixpoint for the set of states (s,h) satisfying the predicate [54]. The monotonic nature of our shape predicate definition guarantees the existence of the descending chain of unfoldings, thus the existence of the least solution.

The heap abstraction $\beta ::= ((\bigvee(\exists v^* \cdot \pi)^*) \mid (\mathbf{ex}\ i \cdot \beta))$ given in last section has the following model:

Definition 5.2 (Model for Heap Approximation)

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

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

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

5.2. Soundness of Verification

The soundness of our verification rules is defined with respect to a small-step operational 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 Appendix A. We use the relation $\hookrightarrow^*$ to denote the transitive closure of the transition relation $\hookrightarrow$. We also need to extract the post-state of a heap constraint by:

Definition 5.3 (Poststate) *Given a constraint* Δ*,* $\textit{Post}(\Delta)$ *captures the relation between primed variables of* Δ*. That is :*

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

For example, given $\Delta = \texttt{x}'::\texttt{node}\langle 3, \texttt{null} \rangle \wedge \texttt{y}{=}5 \wedge \texttt{y}' {>} \texttt{y}{+}1$, $\textit{Post}(\Delta) = \texttt{x}::\texttt{node}\langle 3, \texttt{null} \rangle \wedge \texttt{y} {>} 6$.

Theorem 5.1 (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 Appendix.

Theorem 5.2 (Progress) *If* $\vdash \{\Delta\}\ e\ \{\Delta_2\}$*, 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 Appendix.

Theorem 5.3 (Safety) *Consider a closed term* e *(i.e. a term with no free variables* [6]*) in which all methods have been successfully verified. Assuming unlimited stack/heap spaces and that* $\vdash \{\texttt{true}\}\ e\ \{\Delta\}$*, 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 (i.e. never terminates)* $\langle [],[],e\rangle \not\hookrightarrow^*$.

Proof: Follows from Theorems 5.2 and 5.1 and an auxiliary lemma given in the appendix (Lemma B.1). Details are in the Appendix.

5.3. Soundness of Entailment

The following theorems state that our entailment proving procedure (given in Sec. 4) is sound and always terminates. Proofs are given in the Appendix.

Theorem 5.4 (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 Appendix.

Theorem 5.5 (Termination) *The entailment check* $\Delta_1 \vdash \Delta_2 * \Delta$ *always terminates.*

Proof: Given in the Appendix.

The soundness of the heap approximation procedure *XPure*$_n$ is formalized as follows:

Definition 5.4 (Sound Invariant) *Given a shape predicate* $c\langle v^*\rangle \equiv \Phi\ \mathbf{inv}\ \pi$*, the invariant* π *is* sound *if* *XPure*$_0(\Phi) \Longrightarrow [0/\texttt{null}]\pi$.

Lemma 5.6 (Sound Abstraction) *Given a separation constraint* Φ *where the invariants of the predicates appearing in* Φ *are sound, we have* $\Phi \vdash$ *XPure*$_n(\Phi)$.

Proof: Given in the Appendix.

Lemma 5.6 ensures that if sound invariants are given, it is safe to approximate an antecedent by using *XPure*$_n$. It also allows the possibility of obtaining a more precise invariant by applying *XPure* one or more times (i.e. using *XPure*$_{n+1}$ instead of *XPure*$_n$).

Programs	LOC	No size/bag	Omega Calculator	Isabelle Prover	MONA Prover	Isabelle Prover	MONA Prover
Linked List			*size/length*			*bag/set*	
delete	9	0.02	0.06	17.23	0.12	16.56	0.12
reverse	13	0.02	0.09	13.27	0.1	12.1	0.11
Circular List			*size + cyclic structure*			*bag/set + cyclic structure*	
delete (first)	13	0.01	0.06	14.71	0.12	17.96	0.17
count	29	0.04	0.15	31.94	0.22	39.16	0.29
Double List			*size + double links*			*bag/set + double links*	
append	22	0.05	0.1	23.35	0.22	22.33	0.12
flatten (from tree)	32	0.08	0.5	87.3	11.85	54.23	0.47
Sorted List			*size + min + max + sortedness*			*bag/set+ sortedness*	
delete	20	0.02	0.19	34.09	1.01	13.12	0.25
insertion_sort	32	0.07	0.31	80.9	5.22	27.3	0.21
selection_sort	45	0.10	0.46	135.1	1.5	35.17	0.39
bubble_sort	37	0.16	0.78	127.7	1.16	65.37	0.82
merge_sort	78	0.11	0.61	142.9	8.63	72.53	1.3
quick_sort	70	0.19	0.84	14.8	15.92	28.43	0.71
Binary Search Tree			*min + max + sortedness*			*bag/set + sortedness*	
insert	22	0.08	0.37	72.82	11.92	24.37	0.54
delete	48	0.06	0.53	97.5	11.62	24.39	0.7
Priority Queue			*size+ height+ max-heap*			*bag/set+ size+ max-heap*	
insert	39	0.15	0.45	192.8	2.69	39.59	2.93
delete_max	104	0.55	11.09	648.3	642	77.57	*failed*
AVL Tree			*height+ height-balanced*			*bag/set+ height + height-balanced*	
insert	114	2.77	15.25	85.47	15.05	119.14	29.96
delete	239	2.48	14	106.1	14.24	*failed*	53.22
Red-Black Tree			*size + black-height + height-balanced*			*bag/set+ black-height + height-balanced*	
insert	161	0.97	1.64	307	4.51	211.56	8.63
delete	278	0.95	7.72	653.3	26.62	309.3	7.51

Figure 7. Verification Times (in seconds) for Data Structures with Arithmetic and Bag/Set Constraints

6. Implementation

We have built a prototype system using Objective Caml. The proof obligations generated by our verification are discharged using either a constraint solver or a theorem prover. This is organised as an option in our system and currently covers automatic provers, such as Omega Calculator [49], Isabelle [45], and MONA [30].

Figure 7 summarizes a suite of programs tested. Tests were performed on an Intel Pentium D

[6] In other words, it indicates that all variables in e are locally declared in e.

3.00 GHz. For each example we report:

- the number of code lines (the second column)
- the timings for verifying pointer safety, where only separation/shape information is taken into account, but not size or bag properties (the third column). These timings reveal how much of the verification time is due to the entailment proving of pure formulas.
- the time taken by the verification process when considering separation/shape and size properties. The pure proof obligations were discharged with either Omega (the fourth column), Isabelle (the fifth column), or MONA (the sixth column). Verification time of a function includes the time to verify all functions that it calls.
- the time taken by the verification process when considering separation/shape properties and the bag/set of reachable values inside the data structures. The pure proof obligations were discharged with either Isabelle (the seventh column), or MONA (the eighth column).

The average annotation cost (lines of annotations/lines of code ratio) for our examples is around 7%. Regarding the properties we capture for each data structure, they are summarized below:

- For **single-linked list, circular list** and **doubly-linked list**, the specifications capture the *size* of the list (the total number of nodes). Additionally, for circular list and doubly-linked list, they also capture the *cyclic structure* and the *double links*, respectively. The last two columns contain the verification timings when capturing the set of reachable values as a *bag/set*.
- For **sorted list**, we track the *size* of the list, the minimum (*min*) and the maximum (*max*) elements from the list. The *sortedness* property is expressed using the *min* element, as shown in Section 2.2. For the case when the specification contains the entire *bag/set* of reachable values, we can directly express the sortedness property over the *bag/set*, without explicitly capturing the *min* value (the sixth and seventh columns):

$$\begin{array}{l}\mathtt{sortl3}\langle \mathtt{B}\rangle \equiv (\mathtt{root}{=}\mathtt{null}\wedge \mathtt{B}{=}\{\}) \\ \quad \vee (\mathtt{root}{::}\mathtt{node}\langle \mathtt{v},\mathtt{q}\rangle{*}\mathtt{q}{::}\mathtt{sortl3}\langle \mathtt{B}_1\rangle \wedge \mathtt{B}{=}\mathtt{B}_1{\sqcup}\{\mathtt{v}\} \wedge \ \forall\, \mathtt{x}{\in}\mathtt{B}_1{\cdot}\mathtt{v}{\leq}\mathtt{x})\end{array}$$

- **Binary search tree** requires the elements within the tree to be in sorted order (the *sortedness* property). Our specification captures this property by tracking either the *min/max* values within the tree (the third, fourth and fifth columns), or the entire *bag/set* of reachable values (the sixth and seventh columns).
- For the case of **priority queue**, we track the *size*, the *height* and the highest priority of the elements inside the heap, *max-heap*. The last two columns contain the timings obtained when the specification captures the *bag/set* of reachable values.
- The specification for the **AVL tree** tracks the total number of nodes in the tree, denoted by the *size* property, and its *height*. Additionally, it has an invariant that ensures the *height-balanced* property, meaning that the left and right subtrees are nearly balanced,

as illustrated earlier in Sec 2.2. When tracking the reachable values inside the tree with *bag/set* (the sixth and seventh column), in order to maintain the *height-balanced* invariant, we still need to track the *height* of the AVL tree.

- For the **red-black tree**, we track the *size* (the total number of nodes) and the *black-height* (the height when considering only the black nodes). The specification also ensures the *height-balanced* property, meaning that for all the nodes, each pair of left and right sub-trees have the same black-height. In the last two columns we capture the set of reachable values as a *bag/set*.

Next, we summarize our experience regarding the verification of arithmetic constraints and bag/set constraints, respectively. Regarding **arithmetic constraints**, the time required for shape and size verification is mostly within a couple of seconds when using the Omega Calculator to discharge the proof obligations (the fourth column). In order to have a reference point for the Omega timings, we tried solving the same constraints with two other theorem provers : Isabelle (the fifth column) and MONA (the sixth column). For the former, we only use an automatic but incomplete tactic of the prover. The latter is an implementation of the weak monadic second-order logics WS1S and WS2S ([17]). Therefore, first-order variables can be compared and be subjected only to addition with constants. As Presburger arithmetic ([50]) allows the addition of arbitrary linear arithmetic terms, we converted its formulas into WS1S by encoding naturals as Base-2 bit strings. From our experiments we conclude that the verification process is dominated by entailment proving of pure formulas, which is faster with a specialised solvers, such as Omega Presburger constraints. The timings for verifying shapes only (without size/bag proving) are benign, as reflected in the third column.

With concern to **bag/set constraints**, bag constraints were solved using the multiset theory of Isabelle (the seventh column), while weak monadic second-order theory of 1 successor WS1S from MONA was used to handle set constraints (the eighth column). Due to the incompleteness of the automatic prover that we used from Isabelle, the proof for the `delete` method from the avl tree failed. On the other hand, as Mona translates WS1S formulas into minimum DFAs (Deterministic Finite Automata), the translation may cause a state-space explosion. In our case, we confronted such a problem when verifying the method for deleting the root of a priority heap, `delete_max`, for which the size of the corresponding automaton exceeded the available memory space. From the experiments we can conclude that, when the verification succeeds, it is faster with Mona than with Isabelle.

One remark regarding the verification of bag/set constraints is that, when using Mona for discharging the proof obligations, the properties verified are less precise than with Isabelle. This is due to the fact that from Isabelle we employ the bag (multiset) theory, whereas in Mona we can only use WS1S for set constraints. For illustration, let us consider the specification of the `insert` method for a singly-linked list:

$$\begin{array}{lcl} \texttt{root} :: \texttt{ll}\langle \texttt{B}\rangle & \equiv & (\texttt{root}{=}\texttt{null} \wedge \texttt{B}{=}\{\}) \vee \\ & & (\texttt{root}{::}\texttt{node}\langle \texttt{v}, \texttt{q}\rangle * \texttt{q}{::}\texttt{ll}\langle \texttt{B}_1\rangle \wedge \texttt{B}{=}\texttt{B}_1 \sqcup \{\texttt{v}\}) \end{array}$$

$$\texttt{void insert}(\texttt{node x},\ \texttt{int a})\ \texttt{where}$$
$$\texttt{x}{::}\texttt{ll}\langle \texttt{B}\rangle \ *\!\!\rightarrow\ \texttt{x}{::}\texttt{ll}\langle \texttt{B}_1\rangle \wedge \texttt{B}_1{=}\texttt{B} \sqcup \{\texttt{a}\}$$

For the predicate $\texttt{ll}\langle \texttt{B} \rangle$, $\texttt{B}$ denotes the bag/list of values stored inside the corresponding list. When verifying the method using Isabelle, the constraint $\texttt{B}_1{=}\texttt{B} \sqcup \{\texttt{a}\}$ specifies that only one new node with value $\texttt{a}$ was inserted into the list. However, after verifying the same method using Mona, we can only conclude that at least one node with the value $\texttt{a}$ was inserted into the list.

To speed up the verification process, we have undertaken some performance engineering and rerun the tests. One direction was motivated by the observation that the verification process is dominated by the entailment proving of pure formulas. Consequently, in order to speed up the verification process, we have to speed up the calls to the external solvers. One technique of simplifying these calls, is to replace a single such call with multiple calls corresponding to each disjunct from the antecedent and each conjunct from the consequent, respectively. Following from the [ENT–LHS–OR] rule in Section 4 for handling disjunction on the LHS during the entailment of separation logic formulas, we have applied the same idea for entailments between pure formulas. Our experiments have showed that performing multiple calls to the solvers with smaller formulas is faster than performing only one call with a bigger formula to be discharged.

Apart from the aforementioned speeding up technique, an important future work is to design a safe decomposition strategy for breaking larger predicates, into a number of smaller orthogonal predicates for modular verification. We expect *code modularity*, *decomposed shape views* and *multi-core parallelism* to be important techniques for performance engineering of automated verification system.

The programs we have tested are written using data structures with sophisticated shape, size and bag properties, such as sorted lists, sorted trees, priority queues, balanced trees. Our approach is general enough to handle such interesting data structure properties in an uniform way. Note that our system currently cannot handle map, sequence or non-linear properties as such properties would require specific provers for them. The examples we have tested so far in our experiments are small to medium size programs. The success in verifying such programs confirms the viability of our approach, and allows us to use our system to verify data structure libraries. For large-size programs, significant effort would be required, e.g. in providing user-annotations on method specifications and loop invariants. We envisage that inference mechanisms would be useful to help reduce user-annotations and improve level of automation.

There are also data structures that are beyond the capability of the current system. Since the references between the objects of a data structure are captured by passing object references and fields as parameters to predicate invocations, our predicates cannot precisely capture data structures with non-local references. For instance, certain data structures with fields described by field constraints [59], or those with probabilistically determined fields, such as skip lists [48] are currently not captured by our predicates. These data structures have a common property: certain pointer fields of the objects are non-local in that they do not have a direct relationship with fields of surrounding objects, but rather are determined by some global constraint.

7. Related Work

7.1. Formalisms for Shape Checking/Analysis

Many formalisms for shape analysis are proposed for checking user programs' intricate manipulations of shapely data structures. One well-known work is the **Pointer Assertion Logic** [41] by Moeller and Schwartzbach, which is a highly expressive mechanism to describe in-

variants of graph types [31]. The **Pointer Assertion Logic Engine (PALE)** uses Monadic Second-Order Logic over Strings and Trees as the underlying logic and the tool MONA [30] as the prover. PALE invariants are not designed to handle arithmetic, hence it is not possible to encode height-balanced priority queue in PALE. Moreover, PALE is unsound in handling procedure calls [41], whereas we would like to have a sound verifier. Harwood et al. [20] describe a **UTP theory** for objects and sharing in languages like Java or C++. Their work focuses on a denotational model meant to provide a semantical foundation for refinement-based reasoning or Hoare-style axiomatic reasoning. Our work focuses more on practical verification for heap-manipulating programs.

In an object-oriented setting, the **Dafny language** [39] uses dynamic frames (introduced by Kassios [28]) in its specifications. The term frame refers to a set of memory locations, and an expression denoting a frame is dynamic in the sense that as the program executes, the set of locations denoted by the frame can change. A dynamic frame is thus denoted by a set-valued expression (in particular, a set of object references), and this set is idiomatically stored in a field. Methods in Dafny use modifies and reads clauses, which frame the modifications of methods and dependencies of functions. By comparison, separation logic provides a reasoning logic that hides the explicit representation of dynamic frames.

For shape inference, Sagiv et al. [53] present a parameterized 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 are focused on analysing shape invariants, and do not attempt to track the size and bag properties of complex data structures. An exception is the quantitative shape analysis of Rugina [52] where a data flow analysis is 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 acknowledge the lack of a general specification mechanism for handling arbitrary shape/size properties.

7.2. Size Properties

In another direction of research, size properties are mostly explored for declarative languages [24,60,10] as the immutability property makes their data structures easier to analyse statically. Size analysis is also extended to object-based programs [11] but is 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)** [8] is 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 in a precise manner. 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. In comparison, we use a more limited class of constraint for shape, size and bag analysis but support automated modular verification.

7.3. Set/Bag Properties

Set-based analysis is proposed to verify data structure consistency properties in the work of Kuncak et al. [34], where a decision procedure is given for a first order theory that combines

set and Presburger arithmetic. This result may be used to build a specialised mixed constraint solver but it currently has high algorithmic complexity.

Lahiri and Qadeer [35] report an intra-procedural reachability analysis for well-founded linked lists using first-order axiomatization. Reachability analysis is related to set/bag property that we capture but implemented by transitive closure at the predicate level.

7.4. Unfold/Fold Mechanism

Unfold/fold techniques are originally used for program transformation [6] on purely functional programs. A similar technique called unroll/roll is later used in alias types [58] 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.

An automated procedure that uses unroll/roll is given by Berdine et al. [3], but it is hardwired to work for only `lseg` and `tree` predicates. Furthermore, it performs rolling by unfolding a predicate in the consequent which may miss bindings on free variables. Our unfold/fold mechanism is general, automatic and terminates for heap entailment checking.

7.5. Classical Verifiers

Program verifiers that are based on Hoare-style logic have been around longer than those based on separation logic. We describe some major efforts in this direction.

ESC/Java. Extended Static Checking for Java (ESC/Java) [18], developed at Compaq Systems Research Center, aims to detect more errors than "traditional" static checking tools, such as type checkers, but is not designed to be a program verification system. The stated goals of ESC/Java are scalability and usability. For that, it forgoes soundness for the potential benefits of more automation and faster verification time. Hence, ESC/Java suffers from both false negatives (programs that pass the check may still contain errors that ESC/Java is designed to handle), and false positives (programs flagged as erroneous are in fact correct programs). On the contrary, our verifier is a sound program verifier as it does not suffer from false negatives: if a program is verified, it is guaranteed to meet its specifications for all possible program executions.

ESC/Java2. The ESC/Java effort is continued with ESC/Java2 [13], which adds support for current versions of Java, and also verifies more JML [37] constructs. One significant addition is the support for model fields and method calls within annotations [12]. Since ESC/Java2 continues to use Simplify [15] as its underlying theorem prover which does not support transitive closure operations, it may have difficulties in verifying properties of heap-based data structures that require reachability properties, such as collections of values stored in container data structures.

Spec$^\sharp$. Spec$^\sharp$ [1] is a programming system developed at Microsoft Research. It is an attempt at verifying programs written for the C$^\sharp$ programming language. It adds constructs tailored to program verification, such as pre- and post-conditions, frame conditions, non-null types, model fields and object invariants. Spec$^\sharp$ programs are verified by the Boogie verifier [1], which uses Z3 [14] to discharge its proof obligations. Spec$^\sharp$ also supports runtime assertion checking.

Spec$^\sharp$ supports object invariants but leaves the decision of when to enforce/assume object invariants to the user. In order to verify object invariant modularly, Spec$^\sharp$ employs an ownership scheme that allows an object o to own its representation – objects that are reachable from o and are part of o's abstract state. The ownership scheme in Spec$^\sharp$ forces a top-down unpacking of the objects for updates, and a bottom-up packing for re-establishing the object invariant. The

packing and unpacking of objects are done explicitly by having programmers writing special commands in method bodies.

In our system, instead of using special fields in method contracts to indicate whether an invariant should be enforced, users directly use predicates. Hence, there is no need for explicitly packing and unpacking the objects in the method body. Consequently, users are shielded from the details of the verification methodology, which are largely irrelevant, from a user's point of view.

Jahob. The main focus of Jahob [32,33] is on reasoning techniques for data structure verification that combines multiple theorem provers to reason about expressive logical formulas. Jahob uses a subset of the Isabelle/HOL [45] language as its specification language, and works on instantiatable data structures, as opposed to global data structures used in its predecessor, Hob [36]. Like SPEC$^\sharp$, Jahob supports ghost variables and specification assignments which places onus on programmers to help in the verification process by providing suitable instantiations of these specification variables.

EVE Proofs. EVE Proofs [57] is an automatic verifier for Eiffel [40]. The tool translates Eiffel programs to Boogie [1]. EVE Proofs is integrated in the Eiffel Verification Environment. The authors acknowledge the importance of frame conditions in modular verification. When a routine is called, the verifier is invalidating all knowledge about the locations which may have changed. Therefore it is essential to constrain the effect a routine has on the system to preserve as much information as possible. As Eiffel does not offer a way to specify the frame condition, the authors introduced an automatic extraction of modifies clauses. Their approach uses the postcondition to extract a list of locations which constitute the modifies clause.

Although the approach uses the dynamic type for the pre- and postcondition of a routine call, it uses the static type for the frame condition. This can lead to unsoundness in the system. As opposed to EVE Proofs, our approach does not have to infer frame conditions, courtesy to the frame rule of separation logic [51]. The crucial power of the frame rule is that it allows a global property to be derived from a local one, without looking at other parts of the program.

Another restriction of EVE Proofs regards the methodology for invariants, which has to take into account that objects can temporarily violate the invariant, but also that an object can call other objects while being in an inconsistent state. As this is not considered at the moment, the current implementation of invariants can introduce unsoundness in the system.

As a comparison, we shall discuss some features in our current verification system that differ from those used in traditional verifiers. Our use of user-defined predicates, which capture the properties to be analysed, removes the need for model fields and having object invariants tied to class/type declarations. Regarding ghost specification variables, they are not required since we provide support for automatically instantiating the predicates' parameters. Furthermore, we make use of unfold/fold reasoning to handle the properties of recursive data structures. This obviates the need for specifying *transitive closure* relations that are used by classical verifier, such as Jahob, when tracking recursive properties. Lastly, as separation logic employs local reasoning via a frame rule, our approach does not require a separate `modifies` clause to be prescribed.

8. Conclusion

We have presented in this paper an automated approach to verifying heap-manipulating imperative programs. Compared with other separation logic based automated verification systems [3,16,19,38], our approach has made the following advances: (1) other systems mainly focus on only the separation domain, while we work on a combined domain where not only separation properties (defining the shape of data structures), but also other properties (such as size and bag) can be specified; (2) other systems support only a few built-in predicates over the separation domain, while we allow arbitrary user-specified (inductive) predicates over the domain combined with shape, size and bag properties, which greatly improves the expressiveness of our specification mechanism; (3) most existing systems focus on the verification of the pointer safety, while our approach can verify, in addition to the pointer safety, other properties that require the presence of numerical information such as size and bag. Our approach is built on well-founded shape relations and well-formed separation constraints from which we have designed a novel sound procedure for entailment proofs in the combined domain. Our automated deduction mechanism is based on the unfold/fold reasoning of user-definable predicates and has been proven to be sound and terminating.

While this paper is focused on automated verification, we shall also look into automated inference, in order to allow our system to work for substantial sizable software. Automated inference aims to automatically derive program annotations such as method pre/post-conditions and loop invariants, rather than reply on programmers/users to manually supply. Recently, there has been noticeable advance on automated inference for the separation domain [61,7]. However, it is open how to systematically infer pre/post-conditions and loop invariants for the domain combined with shape, size and bag information and in the presence of user-specified inductive predicates. This remains our main future work.

Acknowledgement
This work was supported by the Singapore-MIT Alliance and the A*STAR grant R-252-000-233-305. Shengchao Qin was supported by the UK Engineering and Physical Sciences Research Council [grant numbers EP/E021948/1, EP/G042322/1].

REFERENCES

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Int'l Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.
2. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *International Workshop on Satisfiability Modulo Theories*. Informal proceedings, 2010.
3. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *Asian Symposium on Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005.
4. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2006.

5. J. Bingham and Z. Rakamaric. A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs. In *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 207–221. Springer, 2006.
6. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, 1977.
7. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–300. ACM, 2009.
8. C. Chen and H. Xi. Combining Programming with Theorem Proving. In *ACM SIGPLAN International Conference on Functional Programming*, pages 66–77. ACM, 2005.
9. W.N. Chin, C. David, H.H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties. In *IEEE International Conference on Engineering Complex Computer Systems*, pages 307–320. IEEE, 2007.
10. W.N. Chin and S.C. Khoo. Calculating sized types. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–72. ACM, 2000.
11. 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 International Conference on Software Engineering*, pages 186–195. ACM, 2005.
12. D. R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005.
13. D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.
14. L. M. de Moura and N. Bjørner. Efficient E-Matching for SMT Solvers. In *International Conference on Automated Deduction*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.
15. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003.
16. D. Distefano, P. W. O'Hearn, and H. Yang. A Local Shape Analysis based on Separation Logic. In *Tools and Algorithms for Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.
17. J. Elgaard, N. Klarlund, and A. Møller. MONA 1.x: new techniques for WS1S and WS2S. In *International Conference on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 516–520. Springer, 1998.
18. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245. ACM, 2002.
19. A. Gotsman, J. Berdine, and B. Cook. Interprocedural Shape Analysis with Separated Heap Abstractions. In *Static Analysis Symposium*, volume 4134 of *Lecture Notes in Computer Science*, pages 240–260. Springer, 2006.
20. W. Harwood, A. Cavalcanti, and J. Woodcock. A Theory of Pointers for the UTP. In *International Colloquium on Theoretical Aspects of Computing*, volume 5160 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2008.
21. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

22. C. A. R. Hoare and J. He. A Trace Model for Pointers and Objects. In *European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1999.
23. C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of ACM*, 50(1):63–69, 2003.
24. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *ACM Symposium on Principles of Programming Languages*, pages 410–423. ACM, 1996.
25. S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM Symposium on Principles of Programming Languages*, pages 14–26. ACM, 2001.
26. L. Jia and D. Walker. ILC: A foundation for automated reasoning about pointer programs. In *European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 131–145. Springer, 2006.
27. C. Jones, P. O'Hearn, and J. Woodcock. Verified Software: A Grand Challenge. *IEEE Computer*, 39(4):93–95, 2006.
28. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006.
29. D. König. Theorie der endlichen und unendlichen Graphen (in German). *Akademische Verlagsgesellschaft*, 1936.
30. N. Klarlund and A. Moller. MONA Version 1.4 - User Manual. BRICS Notes Series, January 2001.
31. N. Klarlund and M. I. Schwartzbach. Graph Types. In *ACM Symposium on Principles of Programming Languages*, pages 196–205. ACM, 1993.
32. V. Kuncak. *Modular Data Structure Verification*. PhD thesis, Massachusetts Institute of Technology, 2007.
33. V. Kuncak, P. Lam, K. Zee, and M. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12):988–1005, 2006.
34. V. Kuncak, H. H. Nguyen, and M. Rinard. An algorithm for deciding bapa: Boolean algebra with presburger arithmetic. In *International Conference on Automated Deduction*, volume 3632 of *Lecture Notes in Computer Science*, pages 260–277. Springer, 2005.
35. S. Lahiri and S. Qadeer. Verifying Properties of Well-Founded Linked Lists. In *ACM Symposium on Principles of Programming Languages*, pages 115–126. ACM, 2006.
36. P. Lam. *The Hob System for Verifying Software Design Properties*. PhD thesis, Massachusetts Institute of Technology, 2007.
37. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
38. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 124–140. Springer, 2005.
39. K. R. M. Leino. Specification and verification of object-oriented software. lecture notes. Marktoberdorf international summer school, 2008.
40. Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
41. A. Moeller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *ACM SIG-*

PLAN Conference on Programming Language Design and Implementation, pages 221–231. ACM, 2001.
42. G. Necula. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM, 1997.
43. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.
44. H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. In *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2007.
45. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
46. P. W. O'Hearn, J. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In *Conference of the European Association for Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.
47. B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
48. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.
49. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.
50. C. R. Reddy and D. W. Loveland. Presburger arithmetic with bounded quantifier alternation. In *ACM Symposium on Theory of Computing*, pages 320–325. ACM, 1978.
51. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.
52. R. Rugina. Quantitative Shape Analysis. In *Proceedings of the International Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2004.
53. S. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217 – 298, 2002.
54. É-J. Sims. Extending separation logic with fixpoints and postponed substitution. *Theoretical Computer Science*, 351(2):258–275, 2006.
55. J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, 1989.
56. A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *International Conference on Computer-Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer, 2002.
57. J. Tschannen. Automatic Verification of Eiffel Programs. Master's thesis, ETH Zurich, 2009.
58. D. Walker and G. Morrisett. Alias Types for Recursive Data Structures. In *International Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206. Springer, 2000.
59. T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2006.
60. H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

61. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *International Conference on Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008.

A. Dynamic Semantics

This section presents a small-step operational semantics for our language given in Fig. 1. The machine configuration is represented by $\langle s, h, e\rangle$, where s denotes the current stack, h denotes the current heap, and e denotes the current program code. Each reduction step is formalized as a transition of the form: $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$. The full set of transitions is given in Fig. 8. We have introduced an intermediate construct $\texttt{ret}(v^*, e)$ to model the outcome of call invocation, where e denotes the residual code of the call. It is also used to handle local blocks. The forward verification rule for this intermediate construct is given as follows:

$$\frac{[\textbf{FV–RET}] \quad \vdash \{\Delta\}\, e\, \{\Delta_2\} \quad \Delta_1 = (\exists v'^* \cdot \Delta_2)}{\vdash \{\Delta\}\, \texttt{ret}(v^*, e)\, \{\Delta_1\}}$$

Note that whenever the evaluation yields a value, we assume this value is stored in a special logical variable `res`, although we do not explicitly put `res` in the stack s.

We also have the following postcondition weakening rule:

$$\frac{[\textbf{FV–POST–WEAKENING}] \quad \vdash \{\Delta\}\, e\, \{\Delta_1\} \quad \Delta_1 \approx\!\gg \Delta_2}{\vdash \{\Delta\}\, e\, \{\Delta_2\}}$$

where $\Delta_1 \approx\!\gg \Delta_2 \ =_{df}\ \forall s, h \cdot s, h \models \mathit{Post}(\Delta_1) \Longrightarrow s, h \models \mathit{Post}(\Delta_2)$. As discussed earlier, we can view Δ_1 and Δ_2 as binary relations (as far as only program variables are concerned). Therefore, we use $\mathit{Post}(\Delta)$ here to refer to the postcondition(i.e. the set of post-states) specified by Δ. Note also that Δ_1 and Δ_2 share the same set of initial states (in which e start to execute).

We now explain the notations used in the operational semantics. We use k to denote a constant, $\perp$ to denote an undefined value, and $()$ to denote the empty expression (program). Note that the runtime stack s is viewed as a 'stackable' mapping, where a variable v may occur several times, and $s(v)$ always refers to the value of the variable v that was popped in most recently.[7] The operation $[v{\mapsto}\nu]{+}s$ "pushes" the variable v to s with the value ν, and $([v{\mapsto}\nu]{+}s)(v) = \nu$. The operation $s{-}\{v^*\}$ "pops out" variables v^* from the stack s. The operation $s[v{\mapsto}\nu]$ changes the value of the most recent v in stack s to ν. The mapping $h[\iota{\mapsto}r]$ is the same as h except that it maps ι to r. The mapping $h{+}[\iota{\mapsto}r]$ extends the domain of h with ι and maps ι to r.

B. Proofs

B.1. Theorem 5.1 – Preservation

Proof: By structural induction on e.

[7] We can give a more formal definition for s, where different occurrences of the same variable can be labeled with different 'frame' numbers. We omit the details here.

$$\langle s,h,v\rangle\hookrightarrow\langle s,h,s(v)\rangle \qquad \langle s,h,k\rangle\hookrightarrow\langle s,h,k\rangle \qquad \langle s,h,v.f\rangle\hookrightarrow\langle s,h,h(s(v))(f)\rangle$$

$$\langle s,h,v{:=}k\rangle\hookrightarrow\langle s[v{\mapsto}k],h,()\rangle \qquad \langle s,h,();e\rangle\hookrightarrow\langle s,h,e\rangle$$

$$\frac{\langle s,h,e_1\rangle\hookrightarrow\langle s_1,h_1,e_3\rangle}{\langle s,h,e_1;e_2\rangle\hookrightarrow\langle s_1,h_1,e_3;e_2\rangle} \qquad \frac{\langle s,h,e\rangle\hookrightarrow\langle s_1,h_1,e_1\rangle}{\langle s,h,v{:=}e\rangle\hookrightarrow\langle s_1,h_1,v{:=}e_1\rangle}$$

$$\frac{s(v){=}\texttt{true}}{\langle s,h,\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2\rangle\hookrightarrow\langle s,h,e_1\rangle} \qquad \frac{s(v){=}\texttt{false}}{\langle s,h,\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2\rangle\hookrightarrow\langle s,h,e_2\rangle}$$

$$\langle s,h,\{t\ v;\ e\}\rangle\hookrightarrow\langle[v{\mapsto}\bot]{+}s,h,\texttt{ret}(v,e)\rangle \qquad \langle s,h,\texttt{ret}(v^*,k)\rangle\hookrightarrow\langle s{-}\{v^*\},h,k\rangle$$

$$\frac{\langle s,h,e\rangle\hookrightarrow\langle s_1,h_1,e_1\rangle}{\langle s,h,\texttt{ret}(v^*,e)\rangle\hookrightarrow\langle s_1,h_1,\texttt{ret}(v^*,e_1)\rangle} \qquad \frac{r{=}h(s(v_1))[f{\mapsto}s(v_2)] \quad h_1{=}h[s(v_1){\mapsto}r]}{\langle s,h,v_1.f:=v_2\rangle\hookrightarrow\langle s,h_1,()\rangle}$$

$$\frac{\texttt{data}\ c\ \{t_1\ f_1,..,t_n\ f_n\}{\in}P \quad \iota{\notin}\mathit{dom}(h) \quad r{=}c[f_1{\mapsto}s(v_1),..,f_n{\mapsto}s(v_n)]}{\langle s,h,\texttt{new}\ c(v_{1\cdots n})\rangle\hookrightarrow\langle s,h{+}[\iota\mapsto r],\iota\rangle}$$

$$\frac{s_1{=}[w_i{\mapsto}s(v_i)]_{i=m}^{n}{+}s \quad t_0\ \mathit{mn}((\texttt{ref}\ t_i\ w_i)_{i=1}^{m-1},(t_i\ w_i)_{i=m}^{n})\ \{e\}}{\langle s,h,\mathit{mn}(v_{1\cdots n})\rangle\hookrightarrow\langle s_1,h,\texttt{ret}(\{w_i\}_{i=m}^{n},[v_i/w_i]_{i=1}^{m-1}e)\rangle}$$

Figure 8. Small-Step Operational Semantics

- Case $v := e$. There are two cases according to the dynamic semantics:
  - e is not a value. From dynamic rules, there is e_1 s.t. $\langle s,h,e\rangle\hookrightarrow\langle s_1,h_1,e_1\rangle$, and $\langle s,h,v{:=}e\rangle\hookrightarrow\langle s_1,h_1,v{:=}e_1\rangle$. From verification rule [FV–ASSIGN], $\vdash\{\Delta\}e\{\Delta_0\}$, and $\Delta_2{=}\exists\texttt{res}{\cdot}\Delta_0{\wedge}_{\{v\}}v'{=}\texttt{res}$. By induction hypothesis, there exists Δ_1, such that $s_1,h_1\models\mathit{Post}(\Delta_1)$ and $\vdash\{\Delta_1\}e_1\{\Delta_2\}$. It concludes from the rule [FV–ASSIGN] that $\vdash\{\Delta_1\}\ v{:=}e_1\ \{\Delta_2\}$.
  - e is a value. Straightforward.
- Case $v_1.f := v_2$. Take $\Delta_1 = \Delta$. It concludes from rule [FV–FIELD–UPDATE] and the dynamic rule.
- Case $\texttt{new}\ c(v_{1\cdots n})$. From verification rule [FV–NEW], we have $\vdash\{\Delta\}\texttt{new}\ c(v_{1\cdots n})\{\Delta_2\}$, where $\Delta_2 = \Delta{*}\texttt{res}{::}c\langle v'_1,..,v'_n\rangle$. Let $\Delta_1 = \Delta_2$. From the dynamic semantics, we have $\langle s,h,\texttt{new}\ c(v_{1\cdots n})\rangle\hookrightarrow\langle s,h{+}[\iota\mapsto r],\iota\rangle$, where $\iota\notin\mathit{dom}(h)$. From $s,h\models\mathit{Post}(\Delta)$, we have $s,h{+}[\iota\mapsto r]\models\mathit{Post}(\Delta_1)$. Moreover, $\vdash\{\Delta_1\}\iota\{\Delta_2\}$.
- Case $e_1;e_2$. We consider the case where e_1 is not a value (otherwise it is straightforward). From the dynamic semantics, we have $\langle s,h,e_1\rangle\hookrightarrow\langle s_1,h_1,e_3\rangle$. From verification rule [FV–SEQ], we have $\vdash\{\Delta\}e_1\{\Delta_3\}$. By induction hypothesis, there exists Δ_1 s.t. $s_1,h_1\models\mathit{Post}(\Delta_1)$, and $\vdash\{\Delta_1\}e_3\{\Delta_3\}$. By rule [FV–SEQ], we have $\vdash\{\Delta_1\}e_3;e_2\{\Delta_2\}$.
- Case $\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. There are two possibilities in the dynamic semantics:
  - $s(v){=}\texttt{true}$. We have $\langle s,h,\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2\rangle\hookrightarrow\langle s,h,e_1\rangle$. Let $\Delta_1 = (\Delta{\wedge}v')$. It is obvious that $s,h\models\mathit{Post}(\Delta_1)$. By the rule [FV–IF], we have $\vdash\{\Delta{\wedge}v'\}\ e_1\ \{\Delta^1\}$.

By the rule [FV–POST–WEAKENING], we have $\vdash \{\Delta \wedge v'\}\, e_1\, \{\Delta^1 \vee \Delta^2\}$. That is, $\vdash \{\Delta_1\}\, e_1\, \{\Delta_2\}$.

– $s(v) = \texttt{false}$. Analogous to the above.

- Case $t\ v;\ e$. Let $\Delta_1 = \Delta$, we conclude immediately from the assumption and the rules [FV–LOCAL] and [FV–RET].

- Case $mn(v_{1..n})$. From rule [FV–CALL], we know $\Delta \vdash \rho\, \Phi_{pr} * \Delta_0$. Take $\Delta_1 = \rho\, \Phi_{pr} * \Delta_0$. From the dynamic rule and the above heap entailment, we have $s_1, h_1 \models \mathit{Post}(\Delta_1)$. From rule [FV–METH], we have $\vdash \{\rho\, \Phi_{pr} * \Delta_0\}\, e_1\, \{\Delta_0 * \Phi_{po}\}$ which concludes.

- Case $\texttt{ret}(v^*, e)$. There are two cases:

  – e is a value k. Let $\Delta_1 = \exists v'^* \cdot \Delta$. It concludes immediately.

  – e is not a value. $\langle s, h, \texttt{ret}(v^*, e)\rangle \hookrightarrow \langle s_1, h_1, \texttt{ret}(v^*, e_1)\rangle$. By [FV–RET] and induction hypothesis, there exists Δ_1 s.t. $s_1, h_1 \models \mathit{Post}(\Delta_1)$ and $\vdash \{\Delta_1\}\, e_1\, \{\Delta_3\}$, and $\Delta_2 = \exists v'^* \cdot \Delta_3$. By rule [FV–RET] again, we have $\vdash \{\Delta_1\}\, \texttt{ret}(v^*, e_1)\, \{\Delta_2\}$.

- Case $\texttt{null} \mid k \mid v \mid v.f$. Straightforward.

B.2. Theorem 5.2 – Progress

Proof: By structural induction on e.

- Case $v := e$. There are two cases:

  – e is a value k. Let $s_1 = s[v \mapsto k]$, $h_1 = h$, and $e_1 = ()$. We conclude.

  – e is not a value. By [FV–ASSIGN], we have $\vdash \{\Delta\}\, e\, \{\Delta_1\}$. By induction hypothesis, there exist s_1, h_1, e_1, such that $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$. We conclude immediately from the dynamic semantics.

- Case $v_1.f := v_2$. Take $e_1 = ()$, $s_1 = s$, and $h_1 = h[s(v_1) \mapsto r]$, where $r = h(s(v_1))[f \mapsto s(v_2)]$. It concludes immediately.

- Case $\texttt{new}\ c(v_{1\cdots n})$. Let ι be a fresh location, r denotes the object value $c[f_1 \mapsto s(v_1), .., f_n \mapsto s(v_n)]$. Take $s_1 = s$, $h_1 = h + [\iota \mapsto r]$, and $e_1 = \iota$. We conclude.

- Case $e_1; e_2$. If e_1 is a value $()$, we conclude immediately by taking $s_1 = s, h_1 = h$. Otherwise, by induction hypothesis, there exist s_1, h_1, e_3 s.t. $\langle s, h, e_1\rangle \hookrightarrow \langle s_1, h_1, e_3\rangle$. We then have $\langle s, h, e_1; e_2\rangle \hookrightarrow \langle s_1, h_1, e_3; e_2\rangle$ from the dynamic semantics.

- Case $\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. It concludes immediately from a case analysis (based on value of v) and the induction hypothesis.

- Case $t\ v;\ e$. Let $s_1 = [v \mapsto \bot] + s$, $h_1 = h$, and $e_1 = \texttt{ret}(v, e)$. We conclude immediately.

- Case $mn(v_{1..n})$. Suppose $v_1, .., v_m$ are pass-by-reference, while others are not. Take $s_1 = [w_i \mapsto s(v_i)]_{i=m}^{n} + s$, $h_1 = h$, and $e_1 = \texttt{ret}(\{w_i\}_{i=m}^{n}, [v_i/w_i]_{i=1}^{m-1} e)$, where w_i are from method specification $t_0\ mn((\texttt{ref}\ t_i\ w_i)_{i=1}^{m-1}, (t_i\ w_i)_{i=m}^{n})\ \{e\}$. We conclude by the dynamic semantics.

- Case $\texttt{ret}(v^*, e)$. If e is a value k, let $s_1 = s - \{v^*\}$, $h_1 = h$, and $e_1 = k$, we conclude. Otherwise, by induction hypothesis, there exist s_1, h_1, e_1 s.t. $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$. We then have $\langle s, h, \texttt{ret}(v^*, e)\rangle \hookrightarrow \langle s_1, h_1, \texttt{ret}(v^*, e_1)\rangle$.

- Case $\texttt{null} \mid k \mid v \mid v.f$. Straightforward.

B.3. Theorem 5.3 – Safety

Before we present the proof for Theorem 5.3, we state and prove the following lemma:

Lemma B.1 *For any s, h, e, if $\langle s, h, e\rangle \hookrightarrow^* \langle \hat{s}, \hat{h}, \nu\rangle$ for some $\hat{s}, \hat{h}, \nu$, where ν is a value, and all free variables of e are already in the domain of the stack s, i.e. free-vars$(e) \subseteq$ dom(s), then dom$(\hat{s}) =$ dom(s).*

Proof: By structural induction over e.

Basic cases: e is $\texttt{null} \mid k \mid v \mid v.f \mid v.f = v_1$. The conclusion is obvious as the stack remains unchanged during the evaluation of e.

Inductive cases:

- e is $v := e_1$. By the operational semantics, we know that $\langle s, h, e_1\rangle \hookrightarrow^* \langle s_1, h_1, \nu_1\rangle$ for some s_1, h_1, ν_1, and $\langle s_1, h_1, v := \nu_1\rangle \hookrightarrow \langle \hat{s}, \hat{h}, \nu\rangle$. Note that free-vars$(e_1) \subseteq$ free-vars$(e) \subseteq$ dom(s), by induction hypothesis, we have dom$(s_1) =$ dom(s). The conclusion follows since dom$(\hat{s}) =$ dom(s_1).

- e is $e_1; e_2$. By the operational semantics, there are s_1, h_1 such that $\langle s, h, e_1\rangle \hookrightarrow^* \langle s_1, h_1, ()\rangle$, $\langle s_1, h_1, (); e_2\rangle \hookrightarrow \langle s_1, h_1, e_2\rangle$, $\langle s_1, h_1, e_2\rangle \hookrightarrow^* \langle \hat{s}, \hat{h}, \nu\rangle$. Note that, for i=1, 2, we have free-vars$(e_i) \subseteq$ free-vars$(e) \subseteq$ dom(s). By induction hypothesis, we have dom$(\hat{s}) =$ dom$(s_1) =$ dom(s).

- e is $t\ v; e_1$. By the operational semantics, we have $\langle s, h, e\rangle \hookrightarrow \langle [v \mapsto \_] + s, h, \texttt{ret}(v, e_1)\rangle$, and $\langle [v \mapsto \_] + s, h, e_1\rangle \hookrightarrow^* \langle s_1, h_1, \nu\rangle$ for some s_1, h_1, and $\langle s_1, h_1, \texttt{ret}(v, \nu)\rangle \hookrightarrow \langle \hat{s}, \hat{h}, \nu\rangle$, where $\hat{s} = s_1 - \{v\}$. Note that free-vars$(e_1) \subseteq$ dom$([v \mapsto _] + s)$, by induction hypothesis, we have dom$(s_1) =$ dom$([v \mapsto _] + s)$. So dom$(\hat{s}) =$ dom$(s_1) - \{v\} =$ dom$([v \mapsto _] + s) - \{v\} =$ dom(s).

- e is $mn(u^*; v^*)$, where v^* are arguments for call-by-value parameters w^*. By the operational semantics, we have (1) $\langle s, h, e\rangle \hookrightarrow \langle [w^* \mapsto v^*] + s, h, \texttt{ret}(w^*, e_{mn})\rangle$, where e_{mn} is the body of the method mn, and (2) $\langle [w^* \mapsto v^*] + s, h, e_{mn}\rangle \hookrightarrow^* \langle s_1, h_1, \nu\rangle$ for some s_1, h_1, and (3) $\langle s_1, h_1, \texttt{ret}(w^*, \nu)\rangle \hookrightarrow \langle \hat{s}, \hat{h}, \nu\rangle$, where $\hat{s} = s_1 - \{w^*\}$. Note also that we have free-vars$(e_{mn}) \subseteq$ dom$([w^* \mapsto v^*] + s)$, by induction hypothesis, we have dom$(s_1) =$ dom$([w^* \mapsto v^*] + s)$. So dom$(\hat{s}) =$ dom$(s_1) - \{w^*\} =$ dom(s). □

Proof of Theorem 5.3: If the evaluation of e does not diverge (is not infinite), it will terminate in a finite number of steps (say n): $\langle [], [], e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle \hookrightarrow \cdots \hookrightarrow \langle s_n, h_n, e_n\rangle$, and there are no further reductions possible. By Theorem 5.1, there exist $\Delta_1, .., \Delta_n$ such that, $s_i, h_i \models$ *Post*(Δ_i), and $\vdash \{\Delta_i\}\ e_i\ \{\Delta\}$. By Theorem 5.2, The final result e_n must be some value v (or it will make another reduction). The conclusion that the stack s_n in the final state is empty is drawn from Lemma B.1 in the above. □

B.4. Soundness and Termination of Heap Entailment

Definition B.1 (length) *We define the* length *of a separation constraint inductively as follows:*

$$\begin{array}{ll}
\mathit{length}(\texttt{emp}) & = 0 \\
\mathit{length}(p{::}c\langle v^*\rangle) & = 1 \\
\mathit{length}(\kappa_1{*}\kappa_2) & = \mathit{length}(\kappa_1){+}\mathit{length}(\kappa_2) \\
\mathit{length}(\exists v^*{\cdot}\kappa{\wedge}\gamma{\wedge}\phi) & = \mathit{length}(\kappa) \\
\mathit{length}(\Phi_1{\vee}\Phi_2) & = \mathit{length}(\Phi_1){+}\mathit{length}(\Phi_2)
\end{array}$$

Definition B.2 (Entailment Transition) *A transition of the form $\mathcal{E}_1 \rightarrow \mathcal{E}_2$ is called an entailment transition where $\mathcal{E}_i$ is either an entailment of the form $\Delta_1{\vdash}^{\kappa}_{V}\Delta_2 * \Delta$ or a fold operation $\mathit{fold}^{\kappa}(\Delta, p{::}c\langle v^*\rangle)$. The set of possible entailment transitions are specified inductively by the entailment rules and the fold operation defined in Section 4.*

- *Rule* [ENT–MATCH]*: There is one possible transition:*

$$\begin{array}{l}
(p_1{::}c\langle v_1^*\rangle{*}\kappa_1){\wedge}\pi_1{\vdash}^{\kappa}_{V}((p_2{::}c\langle v_2^*\rangle{*}\kappa_2){\wedge}\pi_2) * \Delta \\
\rightarrow \\
\kappa_1{\wedge}(\pi_1{\wedge}\mathit{freeEqn}(\rho, V)){\vdash}^{\kappa * p_1::c\langle v_1^*\rangle}_{V-\mathit{dom}(\rho)}\rho(\kappa_2{\wedge}\pi_2) * \Delta
\end{array}$$

- *Rule* [ENT–EMP]*: There is no entailment transition.*

- *Rule* [ENT–UNFOLD]*: There is one transition:*

$$\begin{array}{l}
(p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1){\wedge}\pi_1{\vdash}^{\kappa}_{V}((p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2){\wedge}\pi_2) * \Delta \\
\rightarrow \\
\mathit{unfold}(p_1{::}c_1\langle v_1^*\rangle){*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}((p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2){\wedge}\pi_2) * \Delta
\end{array}$$

- *Rule* [ENT–FOLD]*: There are 2 possible transitions:*

$$\begin{array}{l}
(p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1){\wedge}\pi_1{\vdash}^{\kappa}_{V}((p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2){\wedge}\pi_2) * \Delta \\
\rightarrow \\
\mathit{fold}^{\kappa}((p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1){\wedge}\pi_1, p_1{::}c_2\langle v_2^*\rangle)
\end{array}$$

  and

$$\begin{array}{l}
(p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1){\wedge}\pi_1{\vdash}^{\kappa}_{V}((p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2){\wedge}\pi_2) * \Delta \\
\rightarrow \\
\Delta_i{\wedge}\pi_i^a{\vdash}^{\kappa_i^r}_{V}\kappa_2{\wedge}(\pi_2{\wedge}\pi_i^c) * \Delta \quad \textit{for some } i{\in}1,..,n
\end{array}$$

- *Rule* [ENT–LHS–OR]*: There are two possible transitions:*

$$\begin{array}{lcl}
\Delta_1{\vee}\Delta_2{\vdash}^{\kappa}_{V}\Delta_3 * (\Delta_4{\vee}\Delta_5) & \rightarrow & \Delta_1{\vdash}^{\kappa}_{V}\Delta_3 * \Delta_4 \\
\Delta_1{\vee}\Delta_2{\vdash}^{\kappa}_{V}\Delta_3 * (\Delta_4{\vee}\Delta_5) & \rightarrow & \Delta_2{\vdash}^{\kappa}_{V}\Delta_3 * \Delta_5
\end{array}$$

- *Rule* [ENT–RHS–OR]*: There are two possible transitions:*

$$\begin{array}{lcl}
\Delta_1{\vdash}^{\kappa}_{V}(\Delta_2{\vee}\Delta_3) * \Delta_i^R & \rightarrow & \Delta_1{\vdash}^{\kappa}_{V}\Delta_2 * \Delta_2^R \\
\Delta_1{\vdash}^{\kappa}_{V}(\Delta_2{\vee}\Delta_3) * \Delta_i^R & \rightarrow & \Delta_1{\vdash}^{\kappa}_{V}\Delta_3 * \Delta_3^R
\end{array}$$

- *Rule* [ENT–RHS–EX]*: There is one possible transition:*

$$\Delta_1 \vdash_V^{\kappa} (\exists v \cdot \Delta_2) * \Delta \;\; \rightarrow \;\; \Delta_1 \vdash_{V \cup \{w\}}^{\kappa} ([w/v]\Delta_2) * \Delta_3$$

- *Rule* [ENT–LHS–EX]*: There is one possible transition:*

$$\exists v \cdot \Delta_1 \vdash_V^{\kappa} \Delta_2 * \Delta \;\; \rightarrow \;\; [w/v]\Delta_1 \vdash_V^{\kappa} \Delta_2 * \Delta$$

- *Rule* [FOLDING]*: There is one possible transition:*

$$\mathit{fold}^{\kappa'}(\kappa \wedge \pi, p::c\langle v^* \rangle) \;\; \rightarrow \;\; \kappa \wedge \pi \vdash_{\{v^*\}}^{\kappa'} [p/\mathtt{root}]\Phi * \{(\Delta_i, \kappa_i, V_i, \pi_i)\}_{i=1}^{n}$$

Definition B.3 (Entailment Search Tree) *An entailment search tree for $\mathcal{E} \equiv \Delta_1 \vdash_V^{\kappa} \Delta_2 * \Delta$ is a tree formed as follows:*

- *The nodes of the tree are either entailment relations or fold operations (of the form $\mathit{fold}^{\kappa}(\Delta, p::c\langle v^* \rangle)$).*
- *The root of the tree is $\mathcal{E}$.*
- *The edges from parent nodes to their children nodes are entailment transitions defined in Definition B.2.*

B.5. Theorem 5.4 – Soundness of Heap Entailment

Proof: We need to show that if $\mathcal{E}_0 \equiv \Delta_1 \vdash_V^{\kappa} \Delta_2 * \Delta_3$ succeeds and $s, h \models \Delta_1$, then $s, h \models \Delta_2 * \Delta_3$. Note that the entailment rule [ENT–MATCH] in Sec. 4 denotes a match of two nodes/shape predicates between the antecedent and the consequent. We apply induction on the number of such matches for each path in the entailment search tree for $\mathcal{E}_0$.

Base case. The entailment search succeeds requiring no matches. It can only be the case where rule [ENT–EMP] is applied. It is straightforward to conclude.

Inductive case. Suppose a sequence of transitions $\mathcal{E}_0 \rightarrow \cdots \rightarrow \mathcal{E}_n$ where no match transitions (due to rule [ENT–MATCH]) are involved in this sequence but $\mathcal{E}_n$ will perform a match transition. These transitions can only be generated by the following rules: [ENT–UNFOLD], [ENT–FOLD], [ENT–LHS–OR], [ENT–RHS–OR], [ENT–LHS–EX], and [ENT–RHS–EX]. A case analysis on these rules shows that the following properties hold:

$$\begin{array}{l} s, h \models \mathit{LHS}(\mathcal{E}_i) \implies s, h \models \mathit{LHS}(\mathcal{E}_{i+1}) \\ s, h \models \mathit{RHS}(\mathcal{E}_{i+1}) \implies s, h \models \mathit{RHS}(\mathcal{E}_i) \end{array} \qquad (\dagger)$$

Suppose the match node for $\mathcal{E}_n \equiv \Delta_a \vdash_V^{\kappa} \Delta_c * \Delta_r$ is $p::c\langle v^* \rangle$, and $\mathcal{E}_n$ becomes $\Delta_a' \vdash_V^{\kappa * p::c\langle v^* \rangle} \Delta_c' * \Delta_r$ for some Δ_a', Δ_c'. By induction, we have

$$\forall s, h \cdot s, h \models \Delta_a' \implies s, h \models \Delta_c' * \Delta_r \qquad (\ddagger)$$

From the entailment process, we have $\Delta_a = p::c\langle v^* \rangle * \Delta_a'$, and $\Delta_c = p::c\langle v^* \rangle * \Delta_c'$. Suppose $s, h \models \Delta_a$, then there exist h_0, h_1, such that $h = h_0 * h_1$, $s, h_0 \models p::c\langle v^* \rangle$, and $s, h_1 \models \Delta_a'$. From $(\ddagger)$, we have $s, h_1 \models \Delta_c' * \Delta_r$, which immediately yields $s, h \models \Delta_c * \Delta_r$. We then conclude from $(\dagger)$. □

Before we prove the termination theorem, we state and prove two lemmas.

Lemma B.2 *For any Δ_1 and Δ_2, the entailment search tree for $\mathcal{E} \equiv \Delta_1 \vdash^{\kappa}_{V} \Delta_2 * \Delta$ has only finite number of fold nodes.*

Proof sketch: Suppose the first rule applied in the search tree for $\mathcal{E}$ is [ENT–FOLD] (the only rule that generates fold nodes). By Definition B.2, there are $n{+}1$ children $\mathcal{E}_0, \cdots, \mathcal{E}_n$ for the root node $\mathcal{E}$, for some n, where $\mathcal{E}_0$ is a fold node. Note that the length of the consequent in $\mathcal{E}_i$ is strictly smaller than that in $\mathcal{E}$. On the other hand, node $\mathcal{E}_0$ will perform a transition (due to rule [FOLDING]), yielding a node $\mathcal{E}'$:

$$(p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1){\wedge}\pi_1 \vdash^{\kappa'}_{\{v_2^*\}} [p_1/\mathtt{root}]\Phi * \{\cdots\}$$

This node $\mathcal{E}'$ performs some transitions which do not change the antecedent before it performs a transition due to rule [ENT–MATCH], yielding a new node $\mathcal{E}'_0$:

$$\kappa_1 {\wedge} \pi_1 \vdash^{\kappa' * p_1 :: c_1\langle v_1^*\rangle}_{V-dom(\rho)} \Delta_4 * \Delta'$$

Note that the length of the antecedent in this node is one smaller than that in the root node $\mathcal{E}$. Moreover, it is not possible for $\mathcal{E}'_0$ to perform an unfold operation as the only data node in Φ has been consumed by the match transition. This guarantees the strict decreasing of the length of the antecedent.

In a nutshell, any paths that involve a chain of fold operations will keep the length of the antecedent decreasing, while other paths keep the length of the consequent decreasing. By induction, we can conclude that the number of fold operations is finite. □

Lemma B.3 *For any entailment relation $\mathcal{E} \equiv \Delta_1 \vdash^{\kappa}_{V} \Delta_2 * \Delta$, its entailment search tree is finite branching and has finite depth.*

Proof: Let $l_i = \mathit{length}(\Delta_i)$ for $i = 1, 2$. Obviously we have $l_1{\geq}0,\ l_2{\geq}0$. Let f denote the number of fold operations that have appeared in the entailment search tree.

Due to the well-foundedness of separation constraints, there are finite possible entailment transitions starting from any entailment relation (thus finite possible children for it). This ensures finite branching for each node. What we need to prove is the finite depth property.

To prove finite depth property, we can apply induction on the well-founded measure (f, l_2, l_1) using the following lexicographic order:

$$(f', l_2', l_1') < (f, l_2, l_1) \ =_{df}\ f'{<}f \vee f'{=}f{\wedge}l_2'{<}l_2 \vee f'{=}f{\wedge}(l_2'{=}l_2{\wedge}l_1'{<}l_1)$$

(i). For the base case where the measure at root node is $(f{=}0, l_2{=}0,\ l_1{=}0)$. The only possible transition for the root node is from one of the following rules [ENT–EMP], [ENT–RHS–EX], and [ENT–LHS–EX], as all other rules require $l_1{>}0$ or $l_2{>}0$. If the transition is due to rule [ENT–EMP], the finite depth is obvious due to Definition B.2. If the transition is due to rule [ENT–RHS–EX] or [ENT–LHS–EX], the finite depth is guaranteed as all paths of the tree are formed by finite number of transitions due to rule [ENT–RHS–EX] or [ENT–LHS–EX] and then a transition due to rule [ENT–EMP]. This is because we only have finite number of existential variables.

(ii). For the inductive case $(f{=}m, l_2{=}n, l_1{=}k)$, where $m{+}n{+}k{>}0$. Let us do a case analysis on *the rule* that we apply to the root node $\mathcal{E}$ to generate transitions:

(iia) Rule [ENT–EMP]. The finite depth property is trivial as discussed in (i).

(iib) Rule [ENT–MATCH]. There is only one possible transition $\mathcal{E} \rightarrow \mathcal{E}_1$ (Definition B.2). Let (f', l_2', l_1') denote the measure in the child node $\mathcal{E}_1$, immediately we have $f'{=}f$, $l_2'{=}l_2{-}1$, $l_1'{=}l_1{-}1$. Thus $(f', l_2', l_1') < (f, l_2, l_1)$. By induction hypothesis, the finite depth property holds for the subtree rooted at $\mathcal{E}_1$. So does the whole tree.

(iic) Rule [ENT–UNFOLD]. There is only one possible transition $\mathcal{E} \rightarrow \mathcal{E}_1$ (Definition B.2). Let (f_a, l_{2a}, l_{1a}) denote the measure in the child node $\mathcal{E}_1$.We have $f_a = f$, $l_{2a} = l_2$, and $l_{1a}{\geq}l_1$. The measure does not decrease. However, as a new match is generated after unfolding, the only possible transition from $\mathcal{E}_1$ is the one generated by rule [ENT–MATCH] which we denote as $\mathcal{E}_1 \rightarrow \mathcal{E}_2$. Let (f_b, l_{2b}, l_{1b}) denote the measure in the node $\mathcal{E}_2$. From (iib), we know $f_b{=}f$, $l_{2b}{=}l_{2a}{-}1$, which yields $l_{2b}{=}l_2{-}1{<}l_2$, thus $(f_b, l_{2b}, l_{1b}) < (f, l_2, l_1)$. By induction hypothesis, the finite depth property holds.

(iid) Rule [ENT–FOLD]. There are 2 possible transitions. For the first transition $\mathcal{E} \rightarrow \mathcal{E}_1$ where $\mathcal{E}_1 = \mathit{fold}^{\kappa}(...)$, all nodes in the subtrees of node $\mathcal{E}_1$ have a decreased measure (number of fold operations is decreased!), by induction hypothesis all subtrees of $\mathcal{E}_1$ have finite depth, so is the subtree rooted at $\mathcal{E}_1$. For the other transition $\mathcal{E} \rightarrow \mathcal{E}_i$ (for some $i{\in}2, ..., n{+}1$), we also see the decrease of the measure (the length of the consequence l_2) in nodes $\mathcal{E}_i$. By induction hypothesis again, subtrees rooted at nodes $\mathcal{E}_i$ have finite depth. This concludes the whole tree has finite depth.

(iie) Rule [ENT–LHS–OR] or [ENT–RHS–OR]. The corresponding measure in the only child node is smaller than that in $\mathcal{E}$. It concludes immediately by induction hypothesis.

(iif) Rule [ENT–RHS–EX] or [ENT–LHS–EX]. Starting from $\mathcal{E}$, after finite number of similar transitions (due to [ENT–RHS–EX] or [ENT–LHS–EX]), a different transition (due to rules other than [ENT–RHS–EX] or [ENT–LHS–EX]) will be taken. This then reduces the case to what we have discussed above.

Thus it concludes that the entailment search tree has finite depth. □

B.6. Theorem 5.5 – Termination of Heap Entailment

Proof: By Koenig's lemma [29] and Lemma B.3, all paths are finite. This concludes that the entailment checking terminates. □

B.7. Lemma 5.6 – Sound Abstraction

Before we prove Lemma 5.6, we state and prove the following support lemma.

Lemma B.4 *Given a separation constraint Φ where the invariants of the predicates appearing in Φ are sound, we have $\forall s, h \cdot (\, s, h \models \Phi \implies s \models \mathit{XPure}_0(\Phi))$.*

Proof: By structural induction on Φ.

- $\Phi = \Phi_1 \bigvee \Phi_2$.

$$
\begin{array}{l}
s, h \models \Phi \\
\Longleftrightarrow s, h \models \Phi_1 \bigvee \Phi_2 \\
\Longleftrightarrow s, h \models \Phi_1 \bigvee s, h \models \Phi_2 \quad (\text{model for separation constraint - Def 5.1}) \\
\implies s \models \mathit{XPure}_0(\Phi_1) \bigvee s \models \mathit{XPure}_0(\Phi_2) \quad (\text{induction hypothesis}) \\
\Longleftrightarrow s \models \mathit{XPure}_0(\Phi_1) \bigvee \mathit{XPure}_0(\Phi_2) \quad (\mathit{XPure}_n \text{ definition - Fig 6}) \\
\Longleftrightarrow s \models \mathit{XPure}_0(\Phi)
\end{array}
$$

- $\Phi = \exists v_{1..n} \cdot \kappa \wedge \pi$.

$s, h \models \Phi$
$\iff s, h \models \exists v_{1...n} \cdot \kappa \wedge \pi$
$\iff \exists \nu_{1..n} \cdot s[(v_i \mapsto \nu_i)_{i=1}^n], h \models \kappa \; and \; s[(v_i \mapsto \nu_i)_{i=1}^n] \models \pi$ (model for sep. constraint - Def 5.1)
$\implies \exists \nu_{1..n} \cdot s[(v_i \mapsto \nu_i)_{i=1}^n] \models \mathit{XPure}_0(\kappa) \; and \; s[(v_i \mapsto \nu_i)_{i=1}^n] \models [0/\texttt{null}]\pi$
(induction hypothesis and XPure_0 definition - Fig 6)
$\iff \exists \nu_{1..n} \cdot s[(v_i \mapsto \nu_i)_{i=1}^n] \models \mathit{XPure}_0(\kappa) \wedge [0/\texttt{null}]\pi$ (model for sep. constraint - Def 5.1)
$\iff \exists \nu_{1..n} \cdot s[(v_i \mapsto \nu_i)_{i=1}^n] \models \mathit{XPure}_0(\kappa \wedge \pi)$ (XPure_n definition - Fig 6)
$\iff \exists \nu_{1..n} \cdot s \models \exists \; v_{1..n} \cdot \mathit{XPure}_0(\kappa \wedge \pi)$ (model for separation constraint - Def 5.1)
$\iff \exists \nu_{1..n} \cdot s \models \mathit{XPure}_0(\exists \; v_{1..n} \cdot \kappa \wedge \pi)$ (XPure_n definition - Fig 6)
$\iff s \models \mathit{XPure}_0(\Phi)$

- $\Phi = \kappa_1 * \kappa_2$.

  $s, h \models \Phi$
  $\iff s, h \models \kappa_1 * \kappa_2$
  $\iff s, h_1 \models \kappa_1 \wedge s, h_2 \models \kappa_2 \wedge h = h_1 * h_2$
  Let $\mathit{XPure}_0(\kappa_1) = \exists I \cdot \phi_1$, $\mathit{XPure}_0(\kappa_2) = \exists J \cdot \phi_2$ where
  I and J are composed of fresh symbolic addresses and $I \cap J = \emptyset$
  $\implies s \models \exists I \cdot \phi_1 \wedge s \models \exists J \cdot \phi_2$ and $I \cap J = \emptyset$ (induction hypothesis)
  $\iff s \models (\exists I \cdot \phi_1) \wedge (\exists J \cdot \phi_2)$
  $\iff s \models \mathit{XPure}_0(\kappa_1 * \kappa_2)$ (XPure_n definition - Fig.6)
  $\iff s \models \mathit{XPure}_0(\Phi)$

- $\Phi = \texttt{emp}$. Straightforward.

- $\Phi = p::c\langle v^* \rangle$, and $\mathit{IsData}(c)$.

  $s, h \models \Phi$
  $\iff s, h \models p::c\langle v^* \rangle$
  $\implies \exists \nu \cdot s(p) = \nu \wedge \nu \neq \texttt{null}$ (model for separation constraint - Def 5.1)
  $\iff s \models \exists i \cdot p = i \wedge i \neq 0$
  $\iff s \models \mathit{XPure}_0(\Phi)$ (XPure_n definition - Fig.6)

- $\Phi = p::c\langle v^* \rangle$, and $\mathit{IsView}(c)$.

  $s, h \models \Phi$
  $\iff s, h \models p::c\langle v^* \rangle$
  $\iff s, h \models [p/\texttt{root}]\Phi_c$ (assuming $c\langle v^* \rangle \equiv \Phi_c \; \texttt{inv} \; \pi \; \in \; P$)
  $\implies s \models \mathit{XPure}_0([p/\texttt{root}]\Phi_c)$ (induction hypothesis)
  $\implies s \models [p/\texttt{root}, 0/\texttt{null}]\pi$ (all invariants of predicates in Φ are sound)
  $\iff s \models \mathit{XPure}_0(\Phi)$ (XPure_n definition - Fig.6)

□

Now we present the proof for Lemma 5.6 in what follows.

Proof of Lemma 5.6: Given a separation constraint Φ where the invariants of the predicates appearing in Φ are sound, we show that $\forall s, h \cdot (\, s, h \models \Phi \implies s \models \mathit{XPure}_n(\Phi))$ by induction on n.

Base case: $n = 0$. It follows from Lemma B.4.

Inductive case: We show that for all s, h, $s \models \mathit{XPure}_{n+1}(\Phi)$ if $s, h \models \Phi$ and $s \models \mathit{XPure}_n(\Phi)$. To prove this, we conduct a structural induction on Φ.

- $\Phi = \Phi_1 \bigvee \Phi_2$.

$s, h \models \Phi$
$\iff s, h \models \Phi_1 \bigvee \Phi_2$
$\iff s, h \models \Phi_1 \bigvee s, h \models \Phi_2$ (model for sep. constraint - Def 5.1)
$\implies s \models \mathit{XPure}_{n+1}(\Phi_1) \bigvee s \models \mathit{XPure}_{n+1}(\Phi_2)$ (hypothesis of structural induction)
$\iff s \models \mathit{XPure}_{n+1}(\Phi_1) \bigvee \mathit{XPure}_{n+1}(\Phi_2)$ (XPure_n definition - Fig 6)
$\iff s \models \mathit{XPure}_{n+1}(\Phi)$

- $\Phi = \exists v_{1..m} \cdot \kappa \wedge \pi$.

$s, h \models \Phi$
$\iff s, h \models \exists v_{1...m} \cdot \kappa \wedge \pi$
$\iff \exists \nu_{1..m} \cdot s[(v_i \mapsto \nu_i)_{i=1}^m], h \models \kappa \; and \; s[(v_i \mapsto \nu_i)_{i=1}^m] \models \pi$ (model for sep. constraint - Def 5.1)
$\implies \exists \nu_{1..m} \cdot s[(v_i \mapsto \nu_i)_{i=1}^m] \models \mathit{XPure}_{n+1}(\kappa) \; and \; s[(v_i \mapsto \nu_i)_{i=1}^m] \models [0/\mathtt{null}]\pi$ (hypothesis of structural induction and XPure_n definition - Fig 6)
$\iff \exists \nu_{1..m} \cdot s[(v_i \mapsto \nu_i)_{i=1}^m] \models \mathit{XPure}_{n+1}(\kappa) \wedge [0/\mathtt{null}]\pi$ (model for sep. constraint - Def 5.1)
$\iff \exists \nu_{1..m} \cdot s[(v_i \mapsto \nu_i)_{i=1}^m] \models \mathit{XPure}_{n+1}(\kappa \wedge \pi)$ (XPure_n definition - Fig 6)
$\iff \exists \nu_{1..m} \cdot s \models \exists\, v_{1..m} \cdot \mathit{XPure}_{n+1}(\kappa \wedge \pi)$ (model for sep. constraint - Def 5.1)
$\iff \exists \nu_{1..m} \cdot s \models \mathit{XPure}_{n+1}(\exists\, v_{1..m} \cdot \kappa \wedge \pi)$ (XPure_n definition - Fig 6)
$\iff s \models \mathit{XPure}_{n+1}(\Phi)$

- $\Phi = \kappa_1 * \kappa_2$.

$s, h \models \Phi$
$\iff s, h \models \kappa_1 * \kappa_2$
$\iff s, h_1 \models \kappa_1 \wedge s, h_2 \models \kappa_2 \wedge h = h_1 * h_2$
Let $\mathit{XPure}_{n+1}(\kappa_1) = \exists I \cdot \phi_1$, $\mathit{XPure}_{n+1}(\kappa_2) = \exists J \cdot \phi_2$ where
I and J are composed of fresh symbolic addresses and $I \cap J = \emptyset$
$\implies s \models \exists I \cdot \phi_1 \wedge s \models \exists J \cdot \phi_2$ and $I \cap J = \emptyset$ (hypothesis of structural induction)
$\iff s \models (\exists I \cdot \phi_1) \wedge (\exists J \cdot \phi_2)$
$\iff s \models \mathit{XPure}_{n+1}(\kappa_1 * \kappa_2)$ (XPure_n definition - Fig.6)
$\iff s \models \mathit{XPure}_{n+1}(\Phi)$

- $\Phi = \mathtt{emp}$. Straightforward.

- $\Phi = p{::}c\langle v^* \rangle$, and *IsData*$(c)$.

$$
\begin{array}{l}
s, h \models \Phi \\
\iff s, h \models p{::}c\langle v^* \rangle \\
\implies \exists \nu \cdot s(p) = \nu \wedge \nu \neq \texttt{null} \quad (\text{model for separation constraint - Def 5.1}) \\
\iff s \models \exists i \cdot p = i \wedge i \neq 0 \\
\iff s \models \mathit{XPure}_{n+1}(\Phi) \quad (\mathit{XPure}_n \text{ definition - Fig.6})
\end{array}
$$

- $\Phi = p{::}c\langle v^* \rangle$, and *IsView*$(c)$.

$$
\begin{array}{l}
s, h \models \Phi \\
\iff s, h \models p{::}c\langle v^* \rangle \\
\iff s, h \models [p/\texttt{root}](\Phi_c) \quad (\text{assuming } c\langle v^* \rangle \equiv \Phi_c \ \texttt{inv}\ \pi \ \in \ P) \\
\implies s \models \mathit{XPure}_n([p/\texttt{root}]\Phi_c) \quad (\text{hypothesis of induction over } n) \\
\iff s \models \mathit{XPure}_{n+1}(p{::}c\langle v^* \rangle) \quad (\mathit{XPure}_n \text{ definition - Fig.6})
\end{array}
$$

□