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]Department of Computer Science, Durham University, Durham, 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 that is concise, precise and expressive for ensuring the safety of pointer-based programs. Our approach uses *user-definable* predicates to allow programmers to describe a wide range of data structures with their associated *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. 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 [36] and Ishtiaq and O'Hearn [19], new automated reasoning tools based on separation logic, such as [3,16], have gradually appeared. Several recent works, such as [2,13], 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, these annotations are expected to be derived by the reasoning system. Intraprocedural 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

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

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 [32], 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 [17,21] 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 (that describe 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).

- We improve the expressiveness of our automated verification tool by allowing it to capture both the size and bag properties from each predicate that is being used to define some data structure. The numeric 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, such as the preservation of the elements inside a 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 focus on using separation logic for either program analysis or verification. The general framework of separation logic [36,19] is highly expressive but undecidable. In the search for a decidable fragment of separation logic for

automated verification, Berdine *et al.* [2] supports only a limited set of predicates *without* size properties, disjunctions and existential quantifiers. Similarly, Jia and Walker [20] postponed the handling of recursive predicates in their recent work on automated reasoning of pointer programs. Our approach is more pragmatic as we aim for a sound and terminating formulation of automated verification via separation logic, but do not aim for completeness in the expressive fragment that we handle. On the inference front, Lee et al. [30] has conducted an intraprocedural analysis for loop invariants using grammar approximation under separation logic. Their analysis can handle a wide range of shape predicates with local sharing but is restricted to predicates with two parameters and without size properties. Gotsman et al. [16] have also formulated an interprocedural shape inference which is restricted to just the list segment shape predicate. Sims [39] extends separation logic with fixpoint connectives and postponed substitution to express recursively defined formulae to model the analysis of while-loops. However, it is unclear how to check for entailment in their extended separation logic. While our work does not address the inference/analysis challenge, we have succeeded in providing direct support for automated verification via an expressive specification mechanism through user-specified predicates with 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.

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 size and bag properties.

2.1. Language

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

Figure 1. A Core Imperative Language

We provide a simple imperative language in Figure 1. 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 $\texttt{v.f}_1\texttt{.f}_2\ldots$), and we allow only boolean variables (but not expressions) to be used as the test conditions for conditionals. The language supports data type declaration via *datat*, and shape predicate[2] definition via *spred*. The syntax for shape predicates is given in the next subsection.

[2]Shape predicates are predicates specifying data structure shapes. Our shape predicates can also specify certain numerical properties of data structures, such as size and reachability.

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 $\{\mathtt{x}, \mathtt{y}\}$ 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.

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; return vn }
  else if (x.next=null) then
    { x.next:=vn; vn.next:=null; return x }
  else { x.next:=insert(x.next, vn); return x }}
```

```
node insertion_sort(node y)
  where ···
{ if (y.next=null) then return y
  else {
   y.next:=insertion_sort(y.next);
   return 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. $\mathtt{e}_2$ from $\mathtt{e}_1;\mathtt{e}_2$) denotes the result of an expression. The missing method specifications, denoted by $mspec$, are described in the next section.

2.2. The Specification Language

Separation logic [36,19] extends Hoare logic to support reasoning about shared mutable data structures. It adds two more connectives to classical logic, namely separation conjunction $*$ and separation implication $-\!\!*$. The formula $p_1 * p_2$ asserts that two heaps described by p_1 and p_2 are domain-disjoint. The formula $p_1 -\!\!* p_2$ asserts that if the current heap is extended with a disjoint heap described by p_1, then p_2 holds in the extended heap. In this paper we use only separation conjunction, as we focus on only forward reasoning. This extension can help support more precise and concise reasoning for heap memory, as it can easily support must-aliasing and local reasoning. For example, when we specify that $\mathtt{x}::\mathtt{node}\langle 3, \mathtt{y}\rangle * \mathtt{y}::\mathtt{node}\langle 5, \mathtt{x}\rangle$ [3]

[3]Note that our syntax is slightly different from the separation logic [36,19], in which the formula should be written as $\mathtt{x}\mapsto[\mathtt{val}:3, \mathtt{next}:\mathtt{y}] * \mathtt{y}\mapsto[\mathtt{val}:5, \mathtt{next}:\mathtt{x}]$ denoting the separation conjunction of two singleton heaps.

to be a precondition of some method, we can immediately determine that $\texttt{x}, \texttt{y}$ are non-aliased, namely $\texttt{x}{\neq}\texttt{y}$ due to the use of the separation conjunction, while $\texttt{x.next} = \texttt{y}$ and $\texttt{y.next} = \texttt{x}$ are must-aliases for the two fields from the heap formula. In contrast, if we had used the formula $\texttt{x::node}\langle 3, \texttt{y}\rangle{\wedge}\texttt{y::node}\langle 5, \texttt{x}\rangle$, we may not be able to determine if $\texttt{x}, \texttt{y}$ are aliased with each other, or not, Furthermore, due to the use of local reasoning, we can assume that *only* the heap memory specified in the precondition of each method is ever possibly modified by its 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 [29] or Spec$^\sharp$[1].

We propose an intuitive 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 [36,19] uses the notation $\mapsto$ to denote singleton heaps, e.g. the formula $\texttt{p}{\mapsto}[\texttt{val} : 3, \texttt{next} : \texttt{l}]$ represents a singleton heap referred to by p, where $[\texttt{val} : 3, \texttt{next} : \texttt{l}]$ 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. $\texttt{lseg}(\texttt{p}, \texttt{q})$ represents list segments from p to q. In our system, we unify these two different representations into one form: $\texttt{p::c}\langle \texttt{v}^* \rangle$. When c is a data type name, $\texttt{p::c}\langle \texttt{v}^* \rangle$ stands for a singleton heap $\texttt{p}{\mapsto}[(\texttt{f:v})^*]$ where $\texttt{f}^*$ are fields of data declaration c. When c is a predicate name, $\texttt{p::c}\langle \texttt{v}^* \rangle$ stands for the predicate formula $\texttt{c}(\texttt{p}, \texttt{v}^*)$. The reason we distinguish the first parameter from the rest is that each predicate has an implicit parameter 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:

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

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 $\texttt{root}{=}\texttt{null}$) or consists of a head data node (specified by $\texttt{root::node}\langle \texttt{i}, \texttt{q} \rangle$) and a separate tail data structure which is also an ll list ($\texttt{q::ll}\langle \texttt{m} \rangle$). The $*$ connector ensures that the head node and the tail reside in disjoint heaps. We also specify a default invariant $\texttt{n}{\geq}0$ that holds for all 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 $\texttt{n} = 0$ implies the given invariant $\texttt{n}{\geq}0$. Similarly, the $\texttt{n} = \texttt{m} + 1$ branch together with $\texttt{m}{\geq}0$ from the invariant of $\texttt{q::ll}\langle \texttt{m} \rangle$ also implies the given invariant $\texttt{n}{\geq}0$.) Our predicate uses existential quantifiers for local values and pointers, such as $\texttt{i}, \texttt{m}, \texttt{q}$. The syntax for inductive shape predicates is given in Figure 2. For each shape definition *spred*, the heap-independent invariant π over the parameters $\{\texttt{root}, v^*\}$ 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 $\texttt{m}, \texttt{n}, \texttt{i}$ are of int type, while $\texttt{root}, \texttt{q}$ are of the node type. As the construction of type inference algorithm is quite standard for a language without polymorphism, its description

is omitted in the current paper.

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

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

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. Non-parameter variables (including anonymous variables) in the RHS of the shape definition, 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 [2,4] 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.

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

The constraint min≤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) :

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

Note that we use the notation $\Phi_{pr} *{\mapsto} \Phi_{po}$ to capture a precondition Φ_{pr} and a postcondition Φ_{po} of a method. A special identifier res is used in the postcondition to denote the result of a method. The postcondition of insertion_sort shows that the output list is sorted and has the same number of nodes as the input list.

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 ([34]) 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] 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

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

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

Figure 2. The Specifications

constraint domains in the future, with the adoption of new constraint solvers/provers in our system.

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 set of heap nodes in κ that are reachable from a set of pointers* `S` *can be computed by the following function.*

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

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

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,* `S`*. That is, we have $\forall i \cdot \kappa_i = \texttt{reach}(\kappa_i, \gamma_i, \texttt{S})$, modulo associativity and commutativity of the separation conjunction $*$.*

We also ensure that `root` can appear only in predicate bodies, `res` in postconditions. 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* $p = \texttt{root}$.

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 :

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

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

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

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:

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

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. In the rest of the paper, we will use the following bag operators: bag union $\sqcup$, bag intersection $\sqcap$, bag subsumption $\sqsubset$, and bag cardinality $|\texttt{B}|$. 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 :

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

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:

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

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:

$$
\begin{array}{l}
\texttt{data pair}\{\texttt{node v1; node v2}\} \\
\texttt{pair partition(node x, int p) where} \\
\texttt{x::ll3}\langle \texttt{n}, \texttt{A}\rangle \ *\!\!\rightarrow \texttt{res::pair}\langle \texttt{r}_1, \texttt{r}_2\rangle * \texttt{r}_1\texttt{::ll3}\langle \texttt{n}_1, \texttt{B}_1\rangle * \texttt{r}_2\texttt{::ll3}\langle \texttt{n}_2, \texttt{B}_2\rangle \\
\wedge \texttt{A}{=}\texttt{B}_1 \sqcup \texttt{B}_2 \wedge \texttt{n}{=}\texttt{n}_1 + \texttt{n}_2 \wedge (\forall \texttt{a}{\in}\texttt{B}_1{\cdot}\texttt{a}{\leq}\texttt{p}) \wedge (\forall \texttt{a}{\in}\texttt{B}_2{\cdot}\texttt{a}{>}\texttt{p}) \\
\{ \ \texttt{if } (\texttt{x}{=}\texttt{null}) \texttt{ then new pair}(\texttt{null}, \texttt{null}) \\
\quad \texttt{else } \{ \texttt{ pair t; t:=partition}(\texttt{x.next}, \texttt{p}); \\
\qquad \texttt{if } (\texttt{x.val}{\leq}\texttt{p}) \texttt{ then } \{ \texttt{ x.next:=t.v1; t.v1:=x } \} \\
\qquad\qquad \texttt{else } \{ \texttt{ x.next:=t.v2; t.v2:=x } \}; \\
\qquad \texttt{t } \} \ \}
\end{array}
$$

$$
\begin{array}{l}
\texttt{bool allPos(node x) where} \\
\quad \texttt{x::ll3}\langle \texttt{n}, \texttt{B}\rangle \ *\!\!\rightarrow \texttt{x::ll3}\langle \texttt{n}, \texttt{B}\rangle \wedge ((\forall \texttt{a}{\in}\texttt{B}{\cdot}\texttt{a}{\geq}0) \wedge \texttt{res} \ \vee \ (\exists \texttt{a}{\in}\texttt{B}{\cdot}\texttt{a}{<}0) \wedge \neg\texttt{res}) \\
\{ \ \texttt{if } (\texttt{x}{=}\texttt{null}) \texttt{ then true} \\
\quad \texttt{else if } (\texttt{x.val}{<}0) \texttt{ then false else allPos}(\texttt{x.next}) \ \}
\end{array}
$$

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. These specifications are somewhat expressive, but can be easily handled by our separation logic prover in conjunction with relevant classical provers, such as MONA [33] and Isabelle [23].

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 other program at locations that are possibly reachable, as otherwise such other 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.

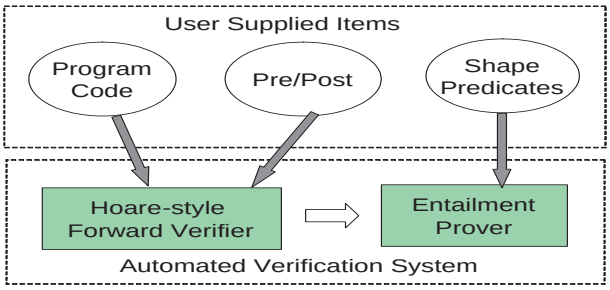

Figure 3: Our Approach to Verification

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. The full set of rules are given in Figure 4. They are used to track heap states as precisely as possible using a range of known techniques. For example, path-sensitivity is captured by the [FV–IF] rule, flow-sensitivity is captured by the [FV–SEQ] rule, while context sensitivity is captured by the [FV–CALL] rule. In our rules, we refer to the latest value of a variable v by v′, while the original value at the start of the method is referenced in the unprimed form, v. Therefore, we can actually view an abstract state, e.g. Δ_1, Δ_2 from the Hoare triple $\vdash \{\Delta_1\}\, e\, \{\Delta_2\}$, as a binary relation over unprimed and primed variables (as far as only program variables are concerned). This is also why the precondition Φ_{pr} is 'lifted' to a binary relation via the *nochange* function in the [FV–METH] rule, where *nochange*(V) returns a formula asserting that the unprimed and primed versions of each variable in V are equal. This is also why we use primed variables in many verification rules which refer to the current values of those variables.

For each call site, [FV–CALL] ensures that the callee's precondition is satisfied. For each method definition, [FV–METH] checks that its postcondition holds for the method body assuming its precondition. A method postcondition may capture only part of the heap at the end of the method, leaving some leaked heap nodes in Δ_2. For a programming language with garbage collector, these leaked memory nodes do not pose any problem, as they can be automatically recovered at runtime. For each shape definition, [FV–PRED] checks that its given invariant is a consequence of the well-founded heap formula. The approximation function *XPure*$_i$ (used in [FV–PRED]) translate 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*$_n$, as described in Sec 4. The soundness of the forward verification is formulated in Sec 5.

We now explain the operators/functions used in our verification rules. The operator $\wedge_{\{v\}}$ used in [FV–ASSIGN] and the operator $*_{V \cup W}$ used in [FV–CALL] are instances of *composition with update* operators. Given a state Δ_1, a state change Δ_2, and a set of variables to be updated

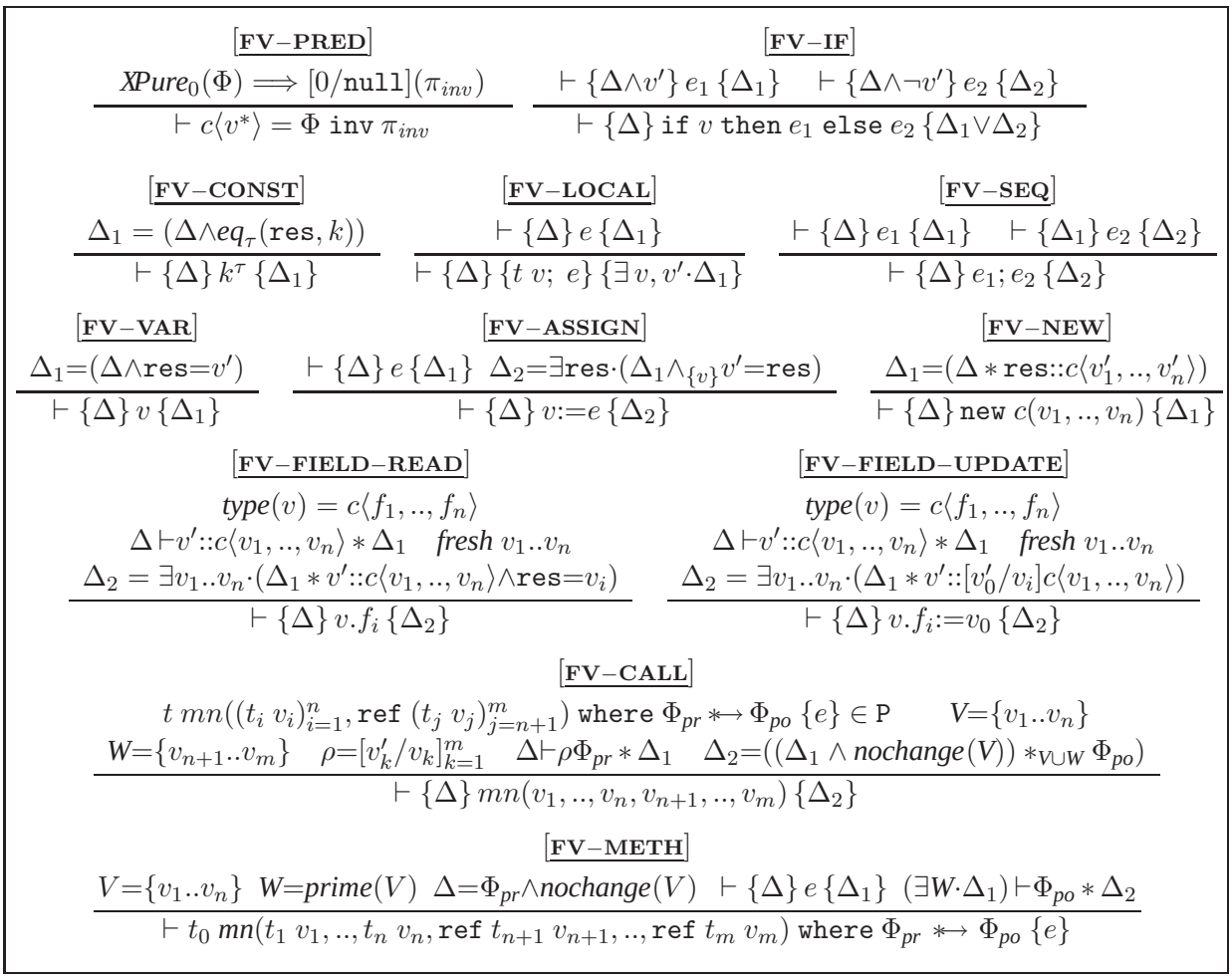

Figure 4. Forward Verification Rules

$X=\{x_1, \ldots, x_n\}$, 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 $*$. The function $\mathtt{eq}_\tau$ is an equality operator that converts boolean constants and `null` to its corresponding integer values, but ignores floating point constants. The function *prime*(V) returns the primed form of all variables in V. $[e^*/v^*]$ represents substitutions of v^* by e^*. A special case is $[0/\mathtt{null}]$, which denotes replacement of `null` by 0. The variable `P` in [FV–CALL] denotes the entire program and is used primarily to retrieve method declarations. 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., $\Delta * \pi = \Delta \wedge \pi$. 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

3.2. Forward Verification Example

We present the detailed verification of the first branch of the `insert` method from Sec 2. Note that program variables appear primed in formulae to denote the latest values, whereas logical variables are always unprimed. The verification is straightforward, except for the last step, where a folding operation is invoked when we check an obtained disjunctive formula establishes the method's postcondition. The procedure to perform the unfolding/folding operations is presented in Sec 4.

$\{\mathtt{x}'{::}\mathtt{sortl}\langle \mathtt{n}, \mathtt{mi}, \mathtt{ma}\rangle * \mathtt{vn}'{::}\mathtt{node}\langle \mathtt{v}, \_\rangle\}$ // [FV–METH](*initialize precondition*)
  if $(\mathbf{vn.val} \leq \mathbf{x.val})$ **then** $\{$
$\{(\mathtt{x}'{::}\mathtt{node}\langle \mathtt{mi}, \mathtt{null}\rangle * \mathtt{vn}'{::}\mathtt{node}\langle \mathtt{v}, \_\rangle \wedge \mathtt{mi}{=}\mathtt{ma} \wedge \mathtt{n}{=}1 \wedge \mathtt{v}{\leq}\mathtt{mi})$
 $\vee\ (\exists \mathtt{q}, \mathtt{k} \cdot \mathtt{x}'{::}\mathtt{node}\langle \mathtt{mi}, \mathtt{q}\rangle * \mathtt{q}{::}\mathtt{sortl}\langle \mathtt{n}{-}1, \mathtt{k}, \mathtt{ma}\rangle * \mathtt{vn}'{::}\mathtt{node}\langle \mathtt{v}, \_\rangle$
$\wedge \mathtt{mi}{\leq}\mathtt{k} \wedge \mathtt{mi}{\leq}\mathtt{ma} \wedge \mathtt{n}{\geq}2 \wedge \mathtt{v}{\leq}\mathtt{mi})\}$ // [FV–IF], [UNFOLDING](*Sec 4*)[5]
    $\mathbf{vn.next} := \mathbf{x}$;
$\{(\mathtt{x}'{::}\mathtt{node}\langle \mathtt{mi}, \mathtt{null}\rangle * \mathtt{vn}'{::}\mathtt{node}\langle \mathtt{v}, \mathtt{x}'\rangle \wedge \mathtt{mi}{=}\mathtt{ma} \wedge \mathtt{n}{=}1 \wedge \mathtt{v}{\leq}\mathtt{mi})$
 $\vee\ (\exists \mathtt{q}, \mathtt{k} \cdot \mathtt{x}'{::}\mathtt{node}\langle \mathtt{mi}, \mathtt{q}\rangle * \mathtt{q}{::}\mathtt{sortl}\langle \mathtt{n}{-}1, \mathtt{k}, \mathtt{ma}\rangle * \mathtt{vn}'{::}\mathtt{node}\langle \mathtt{v}, \mathtt{x}'\rangle$
$\wedge \mathtt{mi}{\leq}\mathtt{k} \wedge \mathtt{mi}{\leq}\mathtt{ma} \wedge \mathtt{n}{\geq}2 \wedge \mathtt{v}{\leq}\mathtt{mi})\}$ // [FV–FIELD–UPDATE]
    $\mathbf{vn}$
$\{(\mathtt{x}'{::}\mathtt{node}\langle \mathtt{mi}, \mathtt{null}\rangle * \mathtt{vn}'{::}\mathtt{node}\langle \mathtt{v}, \mathtt{x}'\rangle \wedge \mathtt{mi}{=}\mathtt{ma} \wedge \mathtt{n}{=}1 \wedge \mathtt{v}{\leq}\mathtt{mi} \wedge \mathtt{res}{=}\mathtt{vn}')$
 $\vee\ (\exists \mathtt{q}, \mathtt{k} \cdot \mathtt{x}'{::}\mathtt{node}\langle \mathtt{mi}, \mathtt{q}\rangle * \mathtt{q}{::}\mathtt{sortl}\langle \mathtt{n}{-}1, \mathtt{k}, \mathtt{ma}\rangle * \mathtt{vn}'{::}\mathtt{node}\langle \mathtt{v}, \mathtt{x}'\rangle$
$\wedge \mathtt{mi}{\leq}\mathtt{k} \wedge \mathtt{mi}{\leq}\mathtt{ma} \wedge \mathtt{n}{\geq}2 \wedge \mathtt{v}{\leq}\mathtt{mi} \wedge \mathtt{res}{=}\mathtt{vn}')\}$ // [FV–VAR]
  $\}$
$\{\mathtt{res}{::}\mathtt{sortl}\langle \mathtt{n}{+}1, \mathtt{min}(\mathtt{v}, \mathtt{mi}), \mathtt{max}(\mathtt{v}, \mathtt{ma})\rangle\}$
    // [FV–METH](*checking postcondition*), [FOLDING](*Sec 4*)

4. Entailment Prover

Our prover for the entailment relation of separation formulas, abbreviated as *heap entailment*, is of the form $\Delta_A \vdash^{\kappa}_{V} \Delta_C * \Delta_R$ which denotes $\kappa * \Delta_A \vdash \exists V \cdot (\kappa * \Delta_C) * \Delta_R$.

The purpose of heap entailment is to check that heap nodes in the antecedent Δ_A are sufficiently precise to cover all nodes from the consequent Δ_C, and to compute a residual heap state Δ_R. κ is the history of nodes from the antecedent that have been used to match nodes from the consequent, V is the list of existentially quantified variables from the consequent. Note that κ and V are derived. The entailment checking procedure is initially invoked with $\kappa = \mathtt{emp}$ and

[5]The rule [UNFOLDING] is to replace the predicate $\mathtt{x}'{::}\mathtt{sortl}\langle \mathtt{n}, \mathtt{mi}, \mathtt{ma}\rangle$ by its definition. The rule [FOLDING] used in the last step is to fold a formula which matches with a predicate's definition back to the predicate. Both rules will be discussed in detail in Sec 4.

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

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

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

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

$$\frac{\Delta_1\vdash^{\kappa}_{V}\Delta_3*\Delta_4\quad \Delta_2\vdash^{\kappa}_{V}\Delta_3*\Delta_5}{\Delta_1\vee\Delta_2\vdash^{\kappa}_{V}\Delta_3*(\Delta_4\vee\Delta_5)}\ [\textbf{ENT-LHS-OR}]$$

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

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

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

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

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

Figure 6. Separation Constraint Entailment

$V=\emptyset$. The entailment proving rules are given in Fig 6. We now briefly discuss the key steps that we may use in such an entailment proof.

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

$$\frac{\frac{\frac{(((\mathtt{p}{=}\mathtt{null}\wedge\mathtt{n}{=}0)\vee(\mathtt{p}{\neq}\mathtt{null}\wedge\mathtt{n}{>}0))\wedge\mathtt{n}{>}0\wedge\mathtt{m}{=}\mathtt{n})\implies\mathtt{p}{\neq}\mathtt{null}}{(\mathit{XPure}_1(\mathtt{p}{::}\mathtt{ll}\langle\mathtt{n}\rangle)\wedge\mathtt{n}{>}0\wedge\mathtt{m}{=}\mathtt{n}\implies\mathtt{p}{\neq}\mathtt{null})\qquad \Delta_{\mathtt{R}}=(\mathtt{n}{>}0\wedge\mathtt{m}{=}\mathtt{n})}}{\mathtt{n}{>}0\wedge\mathtt{m}{=}\mathtt{n}\vdash^{\mathtt{p}::\mathtt{ll}\langle\mathtt{n}\rangle}\mathtt{p}{\neq}\mathtt{null}*\Delta_{\mathtt{R}}}}{\mathtt{p}{::}\mathtt{ll}\langle\mathtt{n}\rangle\wedge\mathtt{n}{>}0\vdash\mathtt{p}{::}\mathtt{ll}\langle\mathtt{m}\rangle\wedge\mathtt{p}{\neq}\mathtt{null}*\Delta_{\mathtt{R}}}$$

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

Another feature of the entailment procedure is exemplified by the transfer of $\mathtt{m}{=}\mathtt{n}$ to the antecedent (and subsequently to the residue). In general, when a match occurs (rule [ENT−MATCH])

and an argument of the heap node coming from the consequent is free, the entailment procedure binds the argument to the corresponding variable from the antecedent and moves the equality to the antecedent. In our system, free variables in consequent are variables from method preconditions. 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) to be aware of their instantiated values. This process is formalized by the function *freeEqn* below, where V is the set of existentially quantified variables:

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

For soundness, we perform a preprocessing step to ensure that variables appearing as arguments of heap nodes and predicates are i) distinct and ii) if they are free, they do not appear in the antecedent by adding (existentially quantified) fresh variables and equalities. This guarantees that the 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.

Apart from the matching operation, another two 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. Unfold/fold operations can be used to handle well-founded inductive predicates in a deductive manner. They are normally invoked before a matching operation is invoked. In particular, we can unfold a predicate that appears in the antecedent if it co-relates (via aliasing) with a data node in the consequent. Correspondingly, if a predicate that appears in the consequent co-relates (via aliasing) with a data node in the antecedent, then we can fold the data node (perhaps together with other nodes) in the antecedent back to a shape predicate so that it can match with the predicate in the consequent. The well-founded condition is sufficient to ensure termination. We shall now use some examples to illustrate these two key steps.

Unfolding a shape predicate in the antecedent. Consider:

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

where $\Delta_{\texttt{R}}$ captures the residue of entailment. Note that a predicate $\texttt{x::ll3}\langle \texttt{n}, \texttt{B}\rangle$ from the antecedent and a data node $\texttt{x::node}\langle \texttt{r}, \texttt{y}\rangle$ from the consequent are co-related via the same variable x. For the entailment to succeed, we would first unfold the $\texttt{ll3}\langle \texttt{n}, \texttt{B}\rangle$ predicate in the antecedent:

$$\begin{array}{l}\exists \texttt{q}_1, \texttt{v} \cdot \texttt{x::node}\langle \texttt{v}, \texttt{q}_1\rangle{*}\texttt{q}_1\texttt{::ll3}\langle \texttt{n}{-}1, \texttt{B}_1\rangle{\wedge}\texttt{n}{>}2{\wedge}\texttt{B}{=}\texttt{B}_1 \cup \{\texttt{v}\} \\ \quad \vdash (\exists \texttt{r}{\cdot}\texttt{x::node}\langle \texttt{r}, \texttt{y}\rangle{\wedge}\texttt{y}{\neq}\texttt{null} \wedge \texttt{r} \in \texttt{B}) * \Delta_{\texttt{R}}\end{array}$$

After removing the existential quantifiers, we obtain:

$$\begin{array}{l}\texttt{x::node}\langle \texttt{v}, \texttt{q}_1\rangle{*}\texttt{q}_1\texttt{::ll3}\langle \texttt{n}{-}1, \texttt{B}_1\rangle{\wedge}\texttt{n}{>}2 \;\wedge\; \texttt{B}{=}\texttt{B}_1 \cup \{\texttt{v}\} \\ \quad \vdash (\texttt{x::node}\langle \texttt{r}, \texttt{y}\rangle{\wedge}\texttt{y}{\neq}\texttt{null} \wedge \texttt{r} \in \texttt{B}) * \Delta_{\texttt{R}}\end{array}$$

The data node in the consequent is then matched up, giving:

$$\texttt{q}_1\texttt{::ll3}\langle \texttt{n}{-}1, \texttt{B}_1\rangle{\wedge}\texttt{n}{>}2{\wedge}\texttt{B}{=}\texttt{B}_1 \cup \{\texttt{v}\}{\wedge}\texttt{q}_1{=}\texttt{y} \;\vdash\; (\texttt{q}_1{\neq}\texttt{null} \wedge \texttt{v} \in \texttt{B}) * \Delta_{\texttt{R}}$$

Due to the well-founded condition, each unfolding exposes a data node that matches with the data node in the consequent. Thus a reduction of the consequent immediately follows, which

contributes to the termination of the entailment proving. A formal definition of the unfolding operation is given by the [UNFOLDING] rule in Figure 6.

Folding against a shape predicate in the consequent. Consider:

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

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 :

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

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

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.

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

A formal definition of folding is specified by the rule [FOLDING] in Figure 6. 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.

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

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

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

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

Figure 7. *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 function $\mathit{XPure}_n(\Phi)$, whose definition is given in Fig 7, returns a sound approximation of Φ as a formula of the form: $\beta ::= \mathbf{ex}\ i \cdot \beta \mid \bigvee(\exists v^*{\cdot}\pi)^*$ where $\mathbf{ex}\ i$ construct is being used to capture a distinct symbolic address i that has been abstracted from a heap node or predicate Φ. The function $\mathit{IsData}(c)$ returns `true` if c is a data node, while $\mathit{IsPred}(c)$ returns `true` if c is a shape predicate.

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

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

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

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

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

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 $\mathtt{ll\langle n\rangle}$, 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 $\mathbf{ex}\ \mathtt{i} \cdot (\mathtt{root{=}0 \wedge n{=}0} \vee \mathtt{root{=}i \wedge i{>}0 \wedge n{-}1{\geq}0})$, 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 $\mathtt{x::node\langle \_, y\rangle} * \mathtt{y::ll\langle m\rangle} \vdash \mathtt{x::ll\langle n\rangle} \wedge \mathtt{n{\geq}1}$ 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 $\mathtt{x::ll\langle n\rangle} \wedge \mathtt{n{>}0} \vdash \mathtt{x{\neq}null}$ 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:

```
data pair { int x; 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.

```
int foo(pair p) where
  ∃a · p::pair⟨a, b⟩ ∧ a+b>0 ⇝ res = b
{if (p.x + p.y)≤0 then error() else p.y}
```

If the expected precondition does not hold, the above method raises an error by calling a special `error()` primitive. Furthermore, we shall assume that the pair object is leaked (or garbage collected) 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 b is *free* since it is used in both precondition and postcondition. In contrast, logical variable 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 residue heap state. Let us re-visit our earlier implicit instantiation technique by examining the following entailment proof.

$$\frac{(\mathtt{c}=2\wedge\mathtt{b}=3 \implies \exists \mathtt{a}\cdot \mathtt{a}=\mathtt{c}\wedge \mathtt{a}{+}\mathtt{b}{>}0) \quad \Delta_{\mathtt{R}}=(\mathtt{c}=2\wedge\mathtt{b}=3)}{\mathtt{c}=2\wedge\mathtt{b}=3 \vdash^{\mathtt{p::pair}\langle 2,3\rangle}_{\{\mathtt{a}\}} (\mathtt{a}=\mathtt{c}\wedge \mathtt{a}{+}\mathtt{b}{>}0)*\Delta_{\mathtt{R}}}$$
$$\overline{\mathtt{p::pair}\langle \mathtt{c},3\rangle\wedge\mathtt{c}=2 \vdash \exists \mathtt{a}\cdot \mathtt{p::pair}\langle \mathtt{a},\mathtt{b}\rangle\wedge \mathtt{a}{+}\mathtt{b}{>}0*\Delta_{\mathtt{R}}}$$

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

We shall now propose an alternative technique for the instantiation of free variables. To do that, we introduce a new notation $(\exists \mathtt{v}{:}\mathcal{I}\cdot\Delta)$ that explicitly marks 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 b, as follows: $(\exists \mathtt{a}\exists \mathtt{b}{:}\mathcal{I}\cdot \mathtt{p::pair}\langle \mathtt{a},\mathtt{b}\rangle\wedge \mathtt{a}{+}\mathtt{b}{>}0)$.

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 emp consequent of entailment prover to:

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

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:

$$\frac{\begin{array}{c}(\mathtt{c}=2 \implies \exists \mathtt{a},\mathtt{b}\cdot \mathtt{a}=\mathtt{c}\wedge\mathtt{b}=3\wedge\mathtt{a}{+}\mathtt{b}{>}0) \quad \pi_{\mathcal{I}}=((\exists \mathtt{a}\cdot(\mathtt{a}=\mathtt{c}\wedge\mathtt{b}=3\wedge\mathtt{a}{+}\mathtt{b}{>}0))\\ \Delta_{\mathtt{R}}=(\mathtt{c}=2\wedge(\mathtt{b}=3\wedge\mathtt{c}{>}{-}3))\end{array}}{\mathtt{c}=2 \vdash^{\mathtt{p::pair}\langle 2,3\rangle}_{\{\mathtt{a},(\mathtt{b}:\mathcal{I})\}} (\mathtt{a}=\mathtt{c}\wedge\mathtt{b}=3\wedge\mathtt{a}{+}\mathtt{b}{>}0)*\Delta_{\mathtt{R}}}$$
$$\overline{\mathtt{p::pair}\langle \mathtt{c},3\rangle\wedge\mathtt{c}=2 \vdash \exists \mathtt{a}\exists \mathtt{b}{:}\mathcal{I}\cdot \mathtt{p::pair}\langle \mathtt{a},\mathtt{b}\rangle\wedge \mathtt{a}{+}\mathtt{b}{>}0*\Delta_{\mathtt{R}}}$$

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.

5. Soundness

In this section we present 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 [36], 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:

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

Note that each heap h is a finite partial mapping while each stack s is a total mapping, as in the classical separation logic [36,19]. 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)

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

Note that $h_1 \perp h_2$ indicates h_1 and h_2 are domain-disjoint, $h_1 \cdot h_2$ denotes the union of disjoint heaps h_1 and h_2. 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 the least fixpoint, i.e., for the set of states (s, h) satisfying the predicate. 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 $\beta = \mathbf{ex}\ i \cdot \beta \mid \bigvee(\exists v^* \cdot \pi)^*$ given in last section has the following model $s, h \models \beta$, namely :

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. We also need to extract the post-state of a heap constraint by:

Definition 5.3 (Poststate) *Given a constraint Δ, Post(Δ) captures the relation between primed variables of Δ. That is :*

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

Theorem 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 without free variables 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 $\langle [], [], e\rangle \not\hookrightarrow^*$.*

Proof Sketch: If the evaluation of `e` does not diverge, it will terminate in a finite number of steps (say n): $\langle [], [], e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle \hookrightarrow \cdots \hookrightarrow \langle s_n, h_n, e_n\rangle$. By Theorem 5.1, there exist $\Delta_1, .., \Delta_n$ such that, $s_i, h_i \models \mathit{Post}(\Delta_i)$, 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). Note that the stack s_i (initially empty) may change if there are local variables declared in e but the final stack s_n will be empty. This can be formally proved by a structural induction over e.

5.3. Soundness of Entailment

The following theorems state that our entailment proving procedure (given in Fig. 6) 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 sketch: A well-founded measure exists for heap entailment. Matching and unfolding decrease nodes from the consequent. The fold operation has bounded recursive depth as each recursive fold operation always decreases the antecedent since the shape predicate has the well-founded property. The size of antecedent is bounded despite unfolding since each unfold is always followed by a decrease of a data node from the consequent. At the end of a fold operation, a heap node from the consequent is also removed. A detailed proof is 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* $c\langle v^* \rangle \equiv \Phi$ **inv** π*, 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* $\Phi \models$ *XPure*$_n(\Phi)$.
Proof *: By structural induction on* Φ.

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*$_n$).

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 [34], Isabelle [33], and MONA [23].

Figure 8 summarizes a suite of programs tested. Tests were performed on an Intel Pentium D 3.00 GHz. For each example we report:

- the timings for verifying pointer safety, where only separation/shape information is taken into account, but not size or bag properties (the second 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 third column), Isabelle (the fourth column), or MONA (the fifth column). Verification time of a function includes the time to verify all functions that it calls.

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

Figure 8. Verification Times (in seconds) for Data Structures with Arithmetic and Bag/Set Constraints

- 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 sixth column), or MONA (the seventh 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):

$$\begin{aligned}&\texttt{sortl3}\langle \texttt{B}\rangle \equiv (\texttt{root}{=}\texttt{null}\wedge \texttt{B}{=}\{\})\\ &\quad \vee (\texttt{root}{::}\texttt{node}\langle \texttt{v},\texttt{q}\rangle{*}\texttt{q}{::}\texttt{sortl3}\langle \texttt{B}_1\rangle \wedge \texttt{B}{=}\texttt{B}_1 \sqcup \{\texttt{v}\} \wedge \ \forall\, \texttt{x} : (\texttt{x} \notin \texttt{B}_1 \vee \texttt{v} \leq \texttt{x})\end{aligned}$$

- **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 subtrees 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.

- **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 third 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 fourth column) and MONA (the fifth 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 ([14]). Therefore, first-order variables can be compared and be subjected only to addition with constants. As Presburger arithmetic ([35])

allows the addition of arbitrary linear arithmetic terms, we converted its formulas into WS1S by encoding naturals as Base-2 bit strings. MONA translates WS1S and WS2S formulas into minimum DFAs (Deterministic Finite Automata) and GTAs (Guided Tree Automata), respectively.

- **Bag/Set Constraints**. Bag constraints were solved using the multiset theory of Isabelle (the sixth column), while weak monadic second-order theory of 1 successor WS1S from MONA was used to handle set constraints (the seventh 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. Regarding the MONA prover, though it provides satisfactory timings when handling set constraints, it might happen for the translation from WS1S and WS2S formulas into DFAs and GTAs to cause a state-space explosion. In our case, we confronted such a problem when verifying the method which deletes the root of a priority heap, `delete_max`, when the size of the corresponding automaton exceeded the available memory space.

From our experiments, we conclude that the verification process is dominated by entailment proving of pure formulas that are fast with specialised solvers, such as Omega for Presburger constraints and MONA for set constraints. The timings for verifying shapes only (without size/bag proving) are benign, as reflected in the second column. Moreover, 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, etc., and show that our approach is general enough to handle such interesting data structures in an uniform way.

We have undertaken some performance engineering in order to speed up the verification process and have 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 had to speed up the calls to the external solvers. One technique of simplifying these calls, was 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 rule `LHS-OR` in Figure 6 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 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.

7. Related Work

Shape Checking/Analysis. Many formalisms for shape analysis have been proposed for checking user programs' intricate manipulations of shapely data structures. One well-known work is the Pointer Assertion Logic [31] by Moeller and Schwartzbach, where shape specifications in monadic second-order logic are given by programmers for loop invariants and method pre/post

conditions, and checked by their MONA tool. For shape inference, Sagiv et al. [38] presented a parameterised framework, called TVLA, using 3-valued logic formulae and abstract interpretation. Based on the properties expected of data structures, programmers must supply a set of predicates to the framework which are then used to analyse that certain shape invariants are maintained. However, most of these techniques were focused on analysing shape invariants, and did not attempt to track the size and bag properties of complex data structures. An exception is the quantitative shape analysis of Rugina [37] where a data flow analysis was proposed to compute quantitative information for programs with destructive updates. By tracking unique points-to reference and its height property, their algorithm is able to handle AVL-like tree structures. Even then, the author acknowledged the lack of a general specification mechanism for handling arbitrary shape/size properties.

Size Properties. In another direction of research, size properties have been most explored for declarative languages [18,41,8] as the immutability property makes their data structures easier to analyse statically. Size analysis was later extended to object-based programs [9] but was restricted to tracking either size-immutable objects that can be aliased and size-mutable objects that are unaliased, with no support for complex shapes. The Applied Type System (ATS) [7] was proposed for combining programs with proofs. In ATS, dependent types for capturing program invariants are extremely expressive and can capture many program properties with the help of accompanying proofs. Using linear logic, ATS may also handle mutable data structures with sharing 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.

Set/Bag Properties. Set-based analysis has been proposed to verify data structure consistency properties in [26], 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 [27] reported 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.

Unfold/Fold Mechanism. Unfold/fold techniques were originally used for program transformation [5] on purely functional programs. A similar technique called unroll/roll was later used in alias types [40] 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 was given by Berdine et al. [2], but it was 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.

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 [15] 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, but it forgoes soundness for the potential benefits of more automation and faster verification time. The ESC/Java effort is continued with ESC/Java2 [11], which

adds support for current versions of Java, and also verifies more JML [29] constructs. One significant addition is the support for model fields and method calls within annotations [10]. 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 [12] to discharge its proof obligations. Spec$^\sharp$ also supports runtime assertion checking. Jahob's [24,25] main focus 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 [33] language as its specification language, and works on instantiatable data structures, as opposed to global data structures used in its predecessor, Hob [28]. 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. As a comparison, we shall discuss some features in our current verification system that differs from those used in traditional verifiers. Firstly, separation logic employs local reasoning via a frame rule. Hence, our approach does not require a separate `modifies` clause to be prescribed. Secondly, we make use of user-defined predicates that allow constraints and parameters to capture properties that we are interested in analysing. This removes the need for model fields and having object invariants tied to class/type declarations. Furthermore, ghost specification variables are not required since we provide support for automatically instantiating the predicates' parameters. Lastly, 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.

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 [2,13,16,30], 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, but also various numerical 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 separation and numerical 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 [42,6]. However, it is open how to systematically infer pre/post-conditions and loop invariants for the domain combined with separation and numerical 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. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *the Asian Symposium on Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, November 2005.
3. 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.
4. J. Bingham and Z. Rakamaric. A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 207–221, Charleston, U.S.A, January 2006. Springer.
5. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, January 1977.
6. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–300, 2009.
7. C. Chen and H. Xi. Combining Programming with Theorem Proving. In *the ACM SIGPLAN International Conference on Functional Programming*, pages 66–77, Tallinn, Estonia, September 2005.
8. W.N. Chin and S.C. Khoo. Calculating sized types. In *the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–72, Boston, United States, January 2000.
9. 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 *International Conference on Software Engineering*, pages 186–195, St. Louis, Missouri, May 2005.
10. D. R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005.
11. D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Int'l 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.
12. L. M. de Moura and N. Bjørner. Efficient E-Matching for SMT Solvers. In *International Conference on Automated Deduction*, pages 183–198, 2007.
13. D. Distefano, P. W. O'Hearn, and H. Yang. A Local Shape Analysis based on Separation Logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, March 2006.

14. J. Elgaard, N. Klarlund, and A. Møller. MONA 1.x: new techniques for WS1S and WS2S. In *Proc. 10th International Conference on Computer-Aided Verification, CAV '98*, volume 1427 of *LNCS*, pages 516–520. Springer-Verlag, June/July 1998.
15. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 234–245, June 2002.
16. 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, Seoul, Korea, August 2006. Springer.
17. C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of ACM*, 50(1):63–69, January 2003.
18. 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 Press, January 1996.
19. 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, London, January 2001.
20. L. Jia and D. Walker. ILC: A foundation for automated reasoning about pointer programs. In *The 15th European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 131–145, March 2006.
21. C. Jones, P. O'Hearn, and J. Woodcock. Verified Software: A Grand Challenge. *IEEE Computer*, 39(4):93–95, April 2006.
22. D. König. Theorie der endlichen und unendlichen Graphen (in German). *Akademische Verlagsgesellschaft*, 1936.
23. N. Klarlund and A. Moller. MONA Version 1.4 - User Manual. BRICS Notes Series, January 2001.
24. V. Kuncak. *Modular Data Structure Verification*. PhD thesis, Massachusetts Institute of Technology, 2007.
25. 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, December 2006.
26. V. Kuncak, H. H. Nguyen, and M. Rinard. An algorithm for deciding bapa: Boolean algebra with presburger arithmetic. In *20th International Conference on Automated Deduction (CADE'05)*, volume 3632 of *Lecture Notes in Computer Science*, pages 260–277, Tallinn, Estonia, July 2005. Springer.
27. S. Lahiri and S. Qadeer. Verifying Properties of Well-Founded Linked Lists. In *ACM Symposium on Principles of Programming Languages*, pages 115–126, South Carolina, January 2006.
28. P. Lam. *The Hob System for Verifying Software Design Properties*. PhD thesis, Massachusetts Institute of Technology, February 2007.
29. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.
30. 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, April 2005.
31. A. Moeller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 221–231, June 2001.
32. G. Necula. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages*, pages 106–119, January 1997.
33. 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.
34. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis.

Communications of the ACM, 8:102–114, 1992.
35. C. R. Reddy and D. W. Loveland. Presburger arithmetic with bounded quantifier alternation. In *the Tenth Annual ACM Symposium on Theory of Computing*, pages 320–325. ACM, 1978.
36. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, July 2002.
37. R. Rugina. Quantitative Shape Analysis. In *Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*, pages 228–245, Verona, Italy, August 2004. Springer.
38. S. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. on Programming Languages and Systems*, 24(3):217 – 298, May 2002.
39. É-J. Sims. Extending separation logic with fixpoints and postponed substitution. *Theoretical Computer Science*, 351(2):258–275, 2006.
40. 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.
41. H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.
42. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *20th 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. 9. 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:

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

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:

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

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.[6] The operation $[v{\mapsto}\nu]{+}s$ "pops in" 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.

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

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

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

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

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

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

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

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

Figure 9. Small-Step Operational Semantics

B. Proofs

B.1. Theorem 5.1 – Preservation

Proof: By structural induction on e.

- 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.

[6] 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.

- Case $v_1.f := v_2$. Take $\Delta_1 = \Delta$. It concludes from rule [FV–FIELD–UPDATE] and the dynamic rule.

- Case $\texttt{new}\ c(v^*)$. From verification rule [FV–NEW], we have $\vdash \{\Delta\}\texttt{new}\ c(v^*)\{\Delta_2\}$, where $\Delta_2 = \Delta * \texttt{res}::c\langle v_1', .., v_n'\rangle$. Let $\Delta_1 = \Delta_2$. From the dynamic semantics, we have $\langle s, h, \texttt{new}\ c(v^*)\rangle \hookrightarrow \langle s, h + [\iota \mapsto r], \iota\rangle$, 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 $\texttt{new}\ c(v^*)$. 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. 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 in Figure 6 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:*

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

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

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

$$\Delta_1{\vee}\Delta_2 \vdash^{\kappa}_{V} \Delta_3 * (\Delta_4{\vee}\Delta_5) \;\rightarrow\; \Delta_1 \vdash^{\kappa}_{V} \Delta_3 * \Delta_4$$

$$\Delta_1{\vee}\Delta_2 \vdash^{\kappa}_{V} \Delta_3 * (\Delta_4{\vee}\Delta_5) \;\rightarrow\; \Delta_2 \vdash^{\kappa}_{V} \Delta_3 * \Delta_5$$

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

$$\Delta_1 \vdash^{\kappa}_{V} (\Delta_2{\vee}\Delta_3) * \Delta_i^R \;\rightarrow\; \Delta_1 \vdash^{\kappa}_{V} \Delta_2 * \Delta_2^R$$

$$\Delta_1 \vdash^{\kappa}_{V} (\Delta_2{\vee}\Delta_3) * \Delta_i^R \;\rightarrow\; \Delta_1 \vdash^{\kappa}_{V} \Delta_3 * \Delta_3^R$$

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

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

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

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

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

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

Definition B.3 (Entailment Search Tree) *An entailment search tree for $\mathcal{E} \equiv \Delta_1 \vdash^{\kappa}_{V} \Delta_2 * \Delta$ 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.*

We now prove the soundness theorem 5.4 (repeated below):

Theorem B.1 (Soundness) *If the entailment* $\Delta_1 \vdash \Delta_2 * \Delta$ *succeeds, then for all* s, h*: if* $s, h \models \Delta_1$ *then* $s, h \models \Delta_2 * \Delta$.

Proof: We need to show that if $\mathcal{E}_0 \equiv \Delta_1 \vdash^{\kappa}_{V} \Delta_2 * \Delta_3$ succeeds and $s, h \models \Delta_1$, then $s, h \models \Delta_2 * \Delta_3$. Note that the entailment rule [ENT–MATCH] in Fig. 6 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 $\mathcal{E}_n \equiv \Delta_a \vdash^{\kappa}_{V} \Delta_c * \Delta_r$ is $p{::}c\langle v^* \rangle$, and $\mathcal{E}_n$ becomes $\Delta'_a \vdash^{\kappa * p::c\langle v^* \rangle}_{V} \Delta'_c * \Delta_r$ 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 $\mathcal{E}'_0$:

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

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. □

We can now prove the termination theorem 5.5 (repeated below):

Theorem B.4 (Termination) *The entailment check* $\Delta_1 \vdash \Delta_2 * \Delta$ *always terminates.*

Proof: By Koenig's lemma [22] and Lemma B.3, all paths are finite. This concludes that the entailment checking terminates. □