Specifying Compatible Sharing in Data Structures

Asankhaya Sharma Aquinas Hobor Wei-Ngan Chin

{asankhs, hobor, chinwn} @comp.nus.edu.sg
Department of Computer Science, National University of Singapore

Abstract. Automated verification of programs that utilize data structures with intrinsic sharing is a challenging problem. Verifying such programs is of practical importance because they occur in many device drivers, runtime systems, and operating system kernels. We develop an extension to separation logic that can reason about aliasing in heaps using a notion of *compatible sharing*. Compatible sharing can model a variety of fine grained sharing and aliasing scenarios with concise specifications. Given these specifications, our entailment procedure enables fully automated verification of a number of challenging programs manipulating data structures with intrinsic sharing. We benchmarked our prototype with a number of examples derived from practical algorithms found in systems code, such as those using threaded trees and overlaid data structures.

1 Introduction

Systems software frequently employs data structures with intrinsic sharing, such as threaded trees (a data structure which can be treated simultaneously as a list and a tree). Sharing enables more efficient use of memory and better performance (*e.g.*, $O(\log\, n)$ lookup for arbitrary elements combined with $O(1)$ lookup for the minimal element). Unfortunately, sharing prevents easy formal reasoning because it precludes simple reasoning about the different parts of the data structure in isolation.

Separation logic [17,11] extends Hoare logic to enable compositional reasoning of heap-manipulating programs and has frequently been applied to automatically verify such programs [4,2,9]. In addition to the usual logical operators $\neg$, $\wedge$, and $\vee$, separation logic utilizes the spatial (or separating) conjunction $*$ to represent assertions valid on disjoint portions of the heap. The separating conjunction enables natural descriptions of many kinds of data structures without intrinsic sharing, such as trees. The disjoint parts of such structures can be reasoned about in isolation—a process called local reasoning—because updates on disjoint heaps do not affect each other. Local reasoning [16] is the key to scalable verification with separation logic.

Local reasoning is encapsulated in the FRAME rule of separation logic:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * F\}\ c\ \{Q * F\}}\ [\mathbf{FRAME}] \qquad \begin{array}{l}\text{(variables modified in } c\\ \text{cannot occur free in } F)\end{array}$$

That is, if a command c can be run in a heap satisfying precondition P, and the resulting state satisfies postcondition Q, then c can also be executed in any larger heap satisfying precondition $P * F$ (for any F) to yield the postcondition $Q * F$. Conversely, reading the rule bottom to top, it is sound to "frame out" any unused portion of the heap F.

Many common data structures, such as linked lists and binary trees, can be naturally represented with $*$ because their constituent subparts occupy disjoint parts of the heap.

However, many more sophisticated data structures, such as threaded trees and graphs, cannot easily be so naturally represented due to the sharing intrinsic to such structures.

We extend the notion of separation to enable local reasoning for such intrinsically-shared data structures by introducing a notion of *compatibility*. In brief, two predicates are compatible when updates to one will not affect the other, despite potential spatial overlap. Consider the following example. The Linux IO scheduler maintains a structure which overlays a doubly-linked list, which maintains insertion order, and a red-black tree, which provides efficient indexing to arbitrary nodes. If the data fields of the nodes are not updated, then the linked list and tree structures do not affect each other despite sharing the entire heap, and we can "frame out" one while working on the other. Thus, our notion of compatibility relies on restricting access to parts of the described structure in a way that is similar to fractional permissions but without the attendant bookkeeping. Prior work on program analysis for overlaid data structures [15] only verifies the shape of the data structure and cannot handle functional properties like order and height balance. In contrast, our proposal is based on user defined inductive predicates with shape, size and bag properties that allows us to express and verify functional correctness of data structures with compatible sharing. In addition, we have certified the correctness proof of *compatibility* checking in Coq.

Our main contribution is an automated procedure to check compatibility and verify programs using compatible data structures. In particular, we describe:

- a specification mechanism that can model sharing and aliasing scenarios,
- an entailment procedure to reason about sharing,
- how to automate the verification of compatible sharing in data structures, and
- our implementation and benchmark its performance. Our prototype, together with a web-based GUI for easy experimentation and machine checked proofs in Coq, is available at:
  `http://loris-7.ddns.comp.nus.edu.sg/~project/HIPComp/`

The rest of the paper is as follows. In section 2 we give some motivating examples. In section 3 we formalize our specification language. In section 4 we describe our entailment procedure. In section 5 we discuss the verification of compatible sharing. In section 6 we discuss our implementation and experiments. In section 7, we review some related work and conclude. Due to lack of space we present the semantic model and soundness proofs in the Appendix (Coq files are available online).

2 Motivating Examples

2.1 From Separation to Sharing

Separation logic provides a natural way to represent disjoint heaps using the separating conjunction $*$. However, if two assertions both require some shared portion of memory, then $*$ cannot easily combine them. Consider the following simple example:

$$\texttt{data pair} \{ \texttt{int fst}; \texttt{int snd} \}$$

Here `pair` is a data structure consisting of two fields, `fst` and `snd`. The following assertion[1] indicates that `x` points to such a structure with field values `f` and `s`:

$$\mathtt{x} \mapsto \mathtt{pair}\langle \mathtt{f}, \mathtt{s} \rangle$$

We denote two disjoint pairs `x` and `y` with the *separating* conjunction $*$, which ensures that `x` and `y` cannot be aliased:

$$\mathtt{x} \mapsto \mathtt{pair}\langle \mathtt{f}_1, \mathtt{s}_1 \rangle * \mathtt{y} \mapsto \mathtt{pair}\langle \mathtt{f}_2, \mathtt{s}_2 \rangle$$

In contrast, to capture aliased pairs we use *classical* conjunction $\wedge$ as follows:

$$\mathtt{x} \mapsto \mathtt{pair}\langle \mathtt{f}_1, \mathtt{s}_1 \rangle \wedge \mathtt{y} \mapsto \mathtt{pair}\langle \mathtt{f}_2, \mathtt{s}_2 \rangle$$

The $\wedge$ operator specifies "must aliasing", that is, $\wedge$ ensures that the pointers `x` and `y` are the equal and that the object field values are identical (*i.e.*, $\mathtt{f}_1 = \mathtt{f}_2$ and $\mathtt{s}_1 = \mathtt{s}_2$).

The basic separating and classical conjunctions are sufficient for "tree-like" data structures. However, to represent and reason about more sophisticated structures we need more subtle specification techniques. Consider a program that is manipulating a threaded tree (a data structure which overlays a list and an ordered tree). The nodes of a common type of threaded tree have six fields: a `data` field; two "list" fields, `next` and `previous`; and three "tree" fields, `left`, `right`, and `parent`. To verify "list operations" such as enqueue and dequeue, we wish to frame out `left`, `right`, and `parent`; conversely, to verify "tree operations" such as insert and lookup, we wish to frame out `next` and `previous`. Tree operations also need access to the `data` field, to support $O(\log\, n)$ access. All of the above means we wish to support field-level framing.

To do so, we add annotations to fields; when the field of an object is absent we mark it with `@A`, whereas when it is present we mark it with `@M`. Consider the following:

$$\mathtt{x} \mapsto \mathtt{pair}\langle \mathtt{f}_1@\mathtt{M}, \mathtt{s}_1@\mathtt{A} \rangle * \mathtt{y} \mapsto \mathtt{pair}\langle \mathtt{f}_2@\mathtt{A}, \mathtt{s}_2@\mathtt{M} \rangle$$

This formula asserts that the heap can be split into two disjoint parts, the first of which contains a first-half-pair pointed to by `x`, and the second of which contains a second-half-pair pointed to by `y`. Since by default fields are mutable `@M`, and when a field is absent `@A` we need not bind a variable to its value, the formula can also be written as:

$$\mathtt{x} \mapsto \mathtt{pair}\langle \mathtt{f}_1, @\mathtt{A} \rangle * \mathtt{y} \mapsto \mathtt{pair}\langle @\mathtt{A}, \mathtt{s}_2 \rangle$$

All this seems simple enough, but there is a subtle wrinkle: notice that `x` and `y` may be aliased (if the combined heap contains a single pair that has been split in half fieldwise), but need not be (if the combined heap contains two distinct half pairs). This ambiguity is inconvenient. We introduce a new operator, the *overlaid* conjunction $\mathbb{A}$ to indicate that the locations are the same although the fields are disjoint. Thus, when we write

$$\mathtt{x} \mapsto \mathtt{pair}\langle \mathtt{f}_1, @\mathtt{A} \rangle \mathbb{A} \mathtt{y} \mapsto \mathtt{pair}\langle @\mathtt{A}, \mathtt{s}_2 \rangle$$

[1] Our separation logic is both "Java-like" and "C-like". Our logic is "Java-like" in the sense that heap locations (pointers) contain (point to) indivisible objects rather than individual memory cells, avoiding the possibility of pointers pointing into the middle of a structure (*i.e.*, skewing). On the other hand, our logic is "C-like" because our formulae are given a classical rather than intuitionistic semantics, *i.e.*, $\mathtt{x} \mapsto \mathtt{pair}\langle \mathtt{f}, \mathtt{s} \rangle$ means that the heap contains **exactly** a single `pair` object at the location pointed to by `x` rather than **at least** a single `pair` object at `x`.

we unambiguously mean that x and y are aliased and have been split fieldwise. On the other hand, hearafter when we use $*$, then x and y are **not** aliased, just as was the case before we added fieldwise separation. We do not use the ambiguous version of $*$.

Unfortunately, separating fields by @A and @M is not enough. For example, a field may be required by "both halves" of the separation; alternatively, we may want to restrict how some fields are used more precisely. For example, in the threaded tree example discussed above, we noted that the "tree" operations also needed access to the data field to support log-time access. However, clients often want to know that even though data may be accessed by the tree operations, it is never modified. To support these use cases we can also mark a field immutable @I along the lines of David *et al.* [5]. The same field can be present (*i.e.*, not absent @A) on both sides of an overlaid conjunction $\mathbb{A}$ as long as both sides are @I. In addition, any mutable field can be "downgraded" into an immutable field. Our annotations are thus a kind of "poor man's fractional permissions [3]", in which @A is analogous to the empty permission, @M is analogous to the full permission, and @I is analogous to an existentialized permission. Although less precise than fractional permissions, these annotations are sufficient for a number of interesting examples and by using them we avoid some of the hassles of integrating fractional permissions into a verification tool [14].

We are now ready to give an intuition for our notion of *compatible sharing*: essentially, a conjunction ($\wedge$, $\mathbb{A}$, and $*$) expresses compatible sharing when one side can be safely framed away. As the simplest example, the following pairs are compatible because the separating conjunction guarantees that they exist on disjoint heaps:

$$\mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_1, \mathtt{s}_1\rangle * \mathtt{y}{\mapsto}\mathtt{pair}\langle \mathtt{f}_2, \mathtt{s}_2\rangle$$

Consider next the following two uses of classical conjunction $\wedge$:

$$\begin{array}{c} \mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_1, @\mathtt{A}\rangle \ \wedge\ \mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_2, @\mathtt{A}\rangle \\ \mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_1@\mathtt{I}, @\mathtt{A}\rangle \ \wedge\ \mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_2@\mathtt{I}, @\mathtt{A}\rangle \end{array}$$

The difference between the two formulae is that in the second example we have marked the field fst as immutable @I. Because fst is mutable @M in the first example, we are not able to frame away half of the conjunction, since we need to maintain the fact that $\mathtt{f}_1 = \mathtt{f}_2$. On the other hand, in the second example, since fst is immutable on both sides of the conjunction, we are able to frame away either side. Therefore, we deem the first example incompatible while we consider the second compatible.

Checking for compatibility is useful not only for the $\wedge$ operator but also for $\mathbb{A}$ operator in the presence of aliasing as shown in the following examples:

$$\begin{array}{ll} \mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_1, @\mathtt{A}\rangle \ \mathbb{A}\ \mathtt{y}{\mapsto}\mathtt{pair}\langle \mathtt{f}_2, \mathtt{s}_2\rangle & (Incompatible) \\ \mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_1, @\mathtt{A}\rangle \ \mathbb{A}\ \mathtt{y}{\mapsto}\mathtt{pair}\langle @\mathtt{A}, \mathtt{s}_2\rangle & (Compatible) \end{array}$$

Our examples so far are for simple pairs. As we will see, our operators are especially useful in the context of more complex objects, such as user defined inductive predicates.

2.2 Shared Process Scheduler

Consider the implementation of a process scheduler, a key data structure for such an implementation is the list of processes currently in the system. Assume that for simplicity, a process may be in only two states, either running or sleeping. In order to efficiently

traverse the list of processes we maintain pointers to the next running or sleeping process as well. The data structure can be represented by the following declaration.

$$\texttt{data node}\ \{\ \texttt{int id};\ \texttt{node next};\ \texttt{node rnext};\ \texttt{node snext}\ \}$$

The node object consists of an integer field denoting the process identifier (`id`). The `next` field points to the next process in the list of all processes. The `rnext` field points to the next running process and `snext` field points to the next sleeping process. In the list of running processes we can mark the `snext` field as absent (`@A`) while in the list of sleeping processes we can mark the `rnext` field as absent. Thus the same set of nodes have multiple views (lists) representing the processes in the system. We use the following three predicates to describe list of all processes (`al`), running processes (`rl`) and sleeping processes (`sl`).

$$\begin{array}{l}
\texttt{al}\langle \texttt{root}, \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\{\}) \\
\quad \vee \exists\, \texttt{d}, \texttt{q}, \texttt{S}_\texttt{q} \cdot (\texttt{root} \mapsto \texttt{node}\langle \texttt{d@I}, \texttt{q}, \texttt{@A}, \texttt{@A}\rangle * \texttt{al}\langle \texttt{q}, \texttt{S}_\texttt{q}\rangle \wedge \texttt{S}{=}\texttt{S}_\texttt{q} \cup \{\texttt{root}\}) \\
\texttt{rl}\langle \texttt{root}, \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\{\}) \\
\quad \vee \exists\, \texttt{d}, \texttt{q}, \texttt{S}_\texttt{q} \cdot (\texttt{root} \mapsto \texttt{node}\langle \texttt{d@I}, \texttt{@A}, \texttt{q}, \texttt{@A}\rangle * \texttt{rl}\langle \texttt{q}, \texttt{S}_\texttt{q}\rangle \wedge \texttt{S}{=}\texttt{S}_\texttt{q} \cup \{\texttt{root}\}) \\
\texttt{sl}\langle \texttt{root}, \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\{\}) \\
\quad \vee \exists\, \texttt{d}, \texttt{q}, \texttt{S}_\texttt{q} \cdot (\texttt{root} \mapsto \texttt{node}\langle \texttt{d@I}, \texttt{@A}, \texttt{@A}, \texttt{q}\rangle * \texttt{sl}\langle \texttt{q}, \texttt{S}_\texttt{q}\rangle \wedge \texttt{S}{=}\texttt{S}_\texttt{q} \cup \{\texttt{root}\})
\end{array}$$

A key safety property for this process scheduler is that all processes in the list `al` should also be in either the list `rl` or the list `sl`. Note that the use of field annotations inside the definition of the predicates ensures that `rl` can only access the running processes as the other fields are marked `@A`. Also the `id` field of node in `al`, `rl` and `sl` is marked immutable with `@I`. We also specify the set of addresses by using the predicate parameter `S`.

$$\texttt{al}\langle \texttt{x}, \texttt{S}_\texttt{x}\rangle * \texttt{rl}\langle \texttt{y}, \texttt{S}_\texttt{y}\rangle * \texttt{sl}\langle \texttt{z}, \texttt{S}_\texttt{z}\rangle \wedge \texttt{S}_\texttt{x}{=}\texttt{S}_\texttt{y} \cup \texttt{S}_\texttt{z} \wedge \texttt{S}_\texttt{y} \cap \texttt{S}_\texttt{z}{=}\{\}$$

Even though this formula uses compatible sharing of heaps, it is non-trivial to prove that automatically. Since the field annotations are hidden inside the predicate definition they cannot be exposed without doing an unfolding of the predicate. In order to expose the information about the fields inside the predicate we introduce the notion of *memory specifications*. We allow the user to specify the memory footprint of the predicate, in particular we support two constructs **mem** and **memE**. A memory specification with **memE** denotes exact footprint while **mem** is used for an under approximation of the footprint. The enhanced predicate definitions for the process scheduler are shown below.

$$\begin{array}{l}
\texttt{al}\langle \texttt{root}, \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\{\}) \\
\quad \vee \exists\, \texttt{d}, \texttt{q}, \texttt{S}_\texttt{q} \cdot (\texttt{root} \mapsto \texttt{node}\langle \texttt{d@I}, \texttt{q}, \texttt{@A}, \texttt{@A}\rangle * \texttt{al}\langle \texttt{q}, \texttt{S}_\texttt{q}\rangle \wedge \texttt{S}{=}\texttt{S}_\texttt{q} \cup \{\texttt{root}\}) \\
\mathbf{memE}\ \texttt{S} \hookrightarrow (\texttt{node}\langle \texttt{@I}, \texttt{@M}, \texttt{@A}, \texttt{@A}\rangle) \\
\texttt{rl}\langle \texttt{root}, \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\{\}) \\
\quad \vee \exists\, \texttt{d}, \texttt{q}, \texttt{S}_\texttt{q} \cdot (\texttt{root} \mapsto \texttt{node}\langle \texttt{d@I}, \texttt{@A}, \texttt{q}, \texttt{@A}\rangle * \texttt{rl}\langle \texttt{q}, \texttt{S}_\texttt{q}\rangle \wedge \texttt{S}{=}\texttt{S}_\texttt{q} \cup \{\texttt{root}\}) \\
\mathbf{memE}\ \texttt{S} \hookrightarrow (\texttt{node}\langle \texttt{@I}, \texttt{@A}, \texttt{@M}, \texttt{@A}\rangle) \\
\texttt{sl}\langle \texttt{root}, \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\{\}) \\
\quad \vee \exists\, \texttt{d}, \texttt{q}, \texttt{S}_\texttt{q} \cdot (\texttt{root} \mapsto \texttt{node}\langle \texttt{d@I}, \texttt{@A}, \texttt{@A}, \texttt{q}\rangle * \texttt{sl}\langle \texttt{q}, \texttt{S}_\texttt{q}\rangle \wedge \texttt{S}{=}\texttt{S}_\texttt{q} \cup \{\texttt{root}\}) \\
\mathbf{memE}\ \texttt{S} \hookrightarrow (\texttt{node}\langle \texttt{@I}, \texttt{@A}, \texttt{@A}, \texttt{@M}\rangle)
\end{array}$$

The **memE** construct consists of a memory region along with a list of possible field annotations that the predicate unfolding would generate. It allows us to syntactically

check if two predicates that share memory region have compatible field annotations. Looking at the memory specification of `al` and `rl` it is easy to see that `al` does not affect (or is compatible with) `rl`. The `id` field is immutable in `rl` and the only field which is mutable in `al` is absent in `rl`. Thus any updates made to the nodes in memory region `S` using predicate `al` will not have any effect when accessing the same memory region using predicate `rl`.

To avoid writing such verbose predicates with set of addresses and to make the specifications more concise we use the overlaid conjunction operator ($\mathbb{A}$). Formulas using the $\mathbb{A}$ operator are translated automatically to those that use the $*$ operator with memory specifications. For the shared process scheduler the memory region shared by the lists `al` is same as the one shared by `rl` and `sl`. The $\mathbb{A}$ operator provides the hint to the system to force the memory on both sides to be the same. Hence the key invariant of the data structure is captured more concisely as:

$$\mathtt{al}\langle \mathtt{x}\rangle \;\mathbb{A}\; (\mathtt{rl}\langle \mathtt{y}\rangle * \mathtt{sl}\langle \mathtt{z}\rangle)$$

This formula is automatically translated by first enhancing the predicate definitions with memory specifications by using the `XMem` function from figure 2. And then forcing the memory region on both sides of $\mathbb{A}$ to be same. As the final translated formula is exactly the same as given before, the use of $\mathbb{A}$ provides a specification mechanism to precisely describe the user intention.

$\mathtt{al}\langle \mathtt{x}\rangle \;\mathbb{A}\; (\mathtt{rl}\langle \mathtt{y}\rangle * \mathtt{sl}\langle \mathtt{z}\rangle)$ //*Provided by User*
$\mathtt{al}\langle \mathtt{x}, \mathtt{S_x}\rangle \;\mathbb{A}\; (\mathtt{rl}\langle \mathtt{y}, \mathtt{S_y}\rangle * \mathtt{sl}\langle \mathtt{z}, \mathtt{S_z}\rangle)$ //*Predicate extension with mem[E]*
$\mathtt{al}\langle \mathtt{x}, \mathtt{S_x}\rangle * \mathtt{rl}\langle \mathtt{y}, \mathtt{S_y}\rangle * \mathtt{sl}\langle \mathtt{z}, \mathtt{S_z}\rangle \wedge \mathtt{S_x} = \mathtt{S_y} \cup \mathtt{S_z} \wedge \mathtt{S_y} \cap \mathtt{S_z} = \{\}$ //*Translated form*

Using the $\mathbb{A}$ operator makes the specification of methods utilizing overlaid structures less verbose. Consider the following `insert` method which is called while scheduling a new process in the system. The new process has to be inserted into `al`, and depending on the status flag, also in `rl` or `sl`. The precondition of the method uses the $\mathbb{A}$ operator to specify the key safety property. The use of overlaid sharing operator allows the user to express the precondition in a concise form. Compatible sharing is used to verify this method as the inserts made to different lists can be shown to not interfere.

```
void insert(int id, int status, node x, node y, node z)
requires  al⟨x⟩ 𝔸 (rl⟨y⟩ * sl⟨z⟩) ∧ status=1
ensures  al⟨x′⟩ 𝔸 (rl⟨y′⟩ * sl⟨z⟩)
requires  al⟨x⟩ 𝔸 (rl⟨y⟩ * sl⟨z⟩) ∧ status=0
ensures  al⟨x′⟩ 𝔸 (rl⟨y⟩ * sl⟨z′⟩)
  { node tmp = new node(id, null, null, null);
    tmp.next = x; x = tmp;
    if(status == 1) y = rlinsert(y, tmp);
    else z = slinsert(z, tmp); }
```

3 Specification Language

We extend the specification language of separation logic with *memory enhanced* predicate definitions. The specification language is as described in Fig. 1 (we use the superscript * to denote a list of elements). $\Phi_{pr} \mathrel{*\!\!\rightarrow} \Phi_{po}$ captures a precondition Φ_{pr} and a

postcondition Φ_{po} of a method or a loop. They are abbreviated from the standard representation `requires` Φ_{pr} and `ensures` Φ_{po}, and formalized by separation logic formula Φ. In turn, the separation logic formula is a disjunction of a heap formula and a pure formula ($\kappa \wedge \pi$). The pure part π captures a rich constraint from the domains of Presburger arithmetic, monadic set constraint or polynomial real arithmetic. We use the set constraints for representing memory regions as shown in Fig. 1. The memory specification allows for two kinds of regions **mem** and **memE**. The difference between them is that while **mem** represents an under approximation of the memory, **memE** captures the exact memory. **memE** is useful in cases like the overlaid data structures where it is important to be able to specify that the memory regions of both overlaying structures are exactly the same.

$$
\begin{array}{lll}
\mathit{pred} & ::= p(v^*) \equiv \Phi \;\; [\mathit{inv}\;\; \pi][\mathit{mem}\;\; \mathcal{S} \hookrightarrow ([c(@u^*)]^*)] \\
 & \quad\quad\quad\quad\quad\quad\quad\quad \mid \Phi \;\; [\mathit{inv}\;\; \pi][\mathit{memE}\;\; \mathcal{S} \hookrightarrow ([c(@u^*)]^*)] \\
\mathit{mspec} & ::= \Phi_{pr} \ast\!\!\rightarrow \Phi_{po} \\
\Phi & ::= \bigvee\; (\exists w^* \cdot \kappa \wedge \pi)^* \\
\kappa & ::= \mathtt{emp} \mid v \mapsto c(v[@u]^*) \mid p(v^*) \mid \kappa_1 \sharp \kappa_2 \quad (\sharp \in \{*, \wedge, \mathbb{A}\}) \\
\pi & ::= \alpha \mid \pi \wedge \varphi \quad \alpha ::= \beta \mid \neg\beta \\
\beta & ::= v_1 = v_2 \mid v = \mathtt{null} \mid a \leq 0 \mid a = 0 \\
a & ::= k \mid k \times v \mid a_1 + a_2 \mid \mathit{max}(a_1, a_2) \mid \mathit{min}(a_1, a_2) \\
\varphi & ::= v \in \mathcal{S} \mid \mathcal{S}_1 = \mathcal{S}_2 \mid \mathcal{S}_1 \subset \mathcal{S}_2 \mid \forall v \in \mathcal{S} \cdot \pi \mid \exists v \in \mathcal{S} \cdot \pi \\
\mathcal{S} & ::= \mathcal{S}_1 \cup \mathcal{S}_2 \mid \mathcal{S}_1 \cap \mathcal{S}_2 \mid \mathcal{S}_1 - \mathcal{S}_2 \mid \{\} \mid \{v\} \\
u & ::= M \mid I \mid A \quad (M \;<:\; I \;<:\; A) \\
\mathit{where} & \; p \text{ is a } \mathit{predicate\ name};\; v, w \text{ are } \mathit{variable\ names}; \\
 & \; c \text{ is a } \mathit{data\ type\ name};\; u \text{ is a } \mathit{field\ annotation};
\end{array}
$$

Fig. 1. Specification Language

For predicate definition we also declare a pure invariant ($\mathit{inv}\;\; \pi$) that is valid for each instance of the predicate. Since we are interested in only under approximation (**mem**) or exactness (**memE**) of memory we do not allow the set of addresses $\mathcal{S}$ to contain any null pointers. Hence, whenever the predicate has a memory specification we strengthen the invariant by automatically adding a constraint using the $\mathtt{addMemInv}(\pi, \mathcal{S})$ function as shown below.

$$addMemInv(\pi, \mathcal{S}) =_{df} \quad \pi \wedge (\forall x \in \mathcal{S} \cdot x \neq \mathtt{null})$$

Before we can use the memory specification in the entailment we need to check whether the predicate definition implies the memory specified by the user. In order to do that we take help of the $\mathtt{XMem}(\kappa)$ function. The $\mathtt{XMem}(\kappa)$ function, whose definition is given in Fig. 2, returns a sound approximation of the memory footprint of heap κ as a tuple of the form: $(\mathcal{S}, [c(@u^*)]^*)$ which corresponds to the set of addresses and the list of field annotations used in memory specifications. The function $\mathtt{isData}(\mathtt{c})$ returns `true` if `c` is a data node, while $\mathtt{isPred}(\mathtt{c})$ returns true if `c` is a heap predicate. We use lists $\mathtt{L_1}$ and $\mathtt{L_2}$ to represent the field annotations. The function $\mathtt{union}(\mathtt{L_1}, \mathtt{L_2})$ returns the union of lists $\mathtt{L_1}$ and $\mathtt{L_2}$. We do not need to consider the pure formula π in `XMem` as it doesn't correspond to any heap. In general, Φ can be disjunctive, so we can have a number of possible approximations of memory for a predicate, each corresponding to a particular disjunct.

$$XMem(\texttt{emp}) =_{df} (\{\}, [])$$

$$\frac{isData(c)}{XMem(c\langle p, v@u^*\rangle) =_{df} (\{p\}, [c\langle @u^*\rangle])}$$

$$\frac{isPred(c) \quad c\langle p, S, v^*\rangle \equiv \Phi[inv\ \ \pi][mem[\texttt{E}]\ \ \mathcal{S}\hookrightarrow L]}{XMem(c\langle p, S, v^*\rangle) =_{df} (S, L)}$$

$$\frac{XMem(\kappa_1)=(S_1, L_1) \quad XMem(\kappa_2)=(S_2, L_2)}{XMem(\kappa_1 \sharp \kappa_2) =_{df} (S_1 \cup S_2, union(L_1, L_2))}$$

Fig. 2. XMem: Translating to Memory Form

We illustrate how the approximation function works by using the example of a linked list.

$$\begin{array}{l}\texttt{data node \{ int val; node next \}}\\ \texttt{ll}\langle\texttt{root}, \texttt{S}\rangle \equiv (\texttt{root}=\texttt{null} \wedge \texttt{S}=\{\})\\ \quad \vee \exists\, \texttt{d}, \texttt{q}, \texttt{S}_\texttt{q} \cdot (\texttt{root} \mapsto \texttt{node}\langle\texttt{d}, \texttt{q}\rangle * \texttt{ll}\langle\texttt{q}, \texttt{S}_\texttt{q}\rangle \wedge \texttt{S}=\texttt{S}_\texttt{q} \cup \{\texttt{root}\})\\ \textbf{memE}\ \texttt{S} \hookrightarrow (\texttt{node}\langle @\texttt{M}, @\texttt{M}\rangle)\end{array}$$

As an example consider the memory approximation of the following predicate.

$$XMem(\texttt{x} \mapsto \texttt{node}\langle\texttt{d}, \texttt{p}\rangle * ll\langle y, S_y\rangle)$$

We proceed by using the rules from Fig. 2 for the data node `x` and predicate `ll`.

$$XMem(\texttt{x} \mapsto \texttt{node}\langle\texttt{d}, \texttt{p}\rangle) = (\{x\}, [node\langle @M, @M\rangle])$$

$$XMem(ll\langle y, S_y\rangle) = (S_y, [node\langle @M, @M\rangle])$$

$$XMem(\texttt{x} \mapsto \texttt{node}\langle\texttt{d}, \texttt{p}\rangle * ll\langle y, S_y\rangle) = (\{x\} \cup S_y, [node\langle @M, @M\rangle])$$

As a consistency check on the memory specification we use the predicate definition to validate the user supplied memory specification. In case the user doesn't provide a memory specification (e.g. when using the $\mathbb{A}$ operator) we automatically extend the predicate definition with set of addresses returned by the `XMem` function. We use an existing underlying (from [5]) entailment procedure to discharge the entailment during validation of memory specifications. The rules for checking the memory specification are given in Fig. 3. In the following discussion for brevity we represent a list of field annotations used in memory specification $(c(@u^*)^*)$ with `L`. We define a $\texttt{subtype}(\texttt{L}_1, \texttt{L}_2)$ function on lists of field annotations. The function returns true if all the field annotations of data nodes in $\texttt{L}_1$ have a corresponding node in $\texttt{L}_2$ and their field annotations are in the subtyping relation (as defined in Fig. 1).

$$subtype(L_1, L_2) \quad =_{df} \quad \forall\, c(@u_1^*)\ in\ L_1, \quad \exists\, c(@u_2^*)\ in\ L_2\ s.t.\ u_2 <: u_1$$

The `subtype` function is used to check the validity of the memory specification by ensuring that the field annotations defined inside the predicate are really subtype of those given by the memory specification. For a predicate $p(v^*) \equiv \Phi\ \ mem[\texttt{E}]\ \ \mathcal{S}\hookrightarrow L$ the following judgment defines the validity of the memory specification.

$$\Phi \vdash_{mem[E]} \mathcal{S}\hookrightarrow L$$

Rules [CHECK−MEM] and [CHECK−MEME] are used when the Φ formula doesn't contain a disjunction, while the rules [CHECK−OR−MEM] and [CHECK−OR−MEME] are used for the disjunctive case. The main difference in the disjunctive case is in the handling of list of field annotations. For the set of addresses ($\mathcal{S}$) we can approximate the heap in each disjunctive formula. However, the field annotations have to be computed for the entire predicate as the annotations may differ in different disjuncts. Since memory specifications are essential to check compatibility in data structures, we have machine checked the soundness proof of these rules and the `XMem` function in Coq.

$$\frac{\Phi = \exists w^* \cdot \kappa \wedge \pi \quad XMem(\kappa) = (S_x, L_x) \quad \Phi \vdash^{\kappa}_{V,I} (S \subset S_x) * \Delta \quad subtype(L, L_x)}{\Phi \vdash_{mem} \mathcal{S} \hookrightarrow L} \; [\textbf{CHECK-MEM}]$$

$$\frac{\Phi = \exists w^* \cdot \kappa \wedge \pi \quad XMem(\kappa) = (S_x, L_x) \quad \Phi \vdash^{\kappa}_{V,I} (S = S_x) * \Delta \quad subtype(L, L_x) \wedge subtype(L_x, L)}{\Phi \vdash_{memE} \mathcal{S} \hookrightarrow L} \; [\textbf{CHECK-MEME}]$$

$$\frac{\Phi_1 = \exists w_1^* \cdot \kappa_1 \wedge \pi_1 \quad \Phi_2 = \exists w_2^* \cdot \kappa_2 \wedge \pi_2 \quad XMem(\kappa_1) = (S_1, L_1) \quad XMem(\kappa) = (S_2, L_2) \quad \Phi_1 \vdash^{\kappa}_{V,I} (S \subset S_1) * \Delta \quad \Phi_2 \vdash^{\kappa}_{V,I} (S \subset S_2) * \Delta \quad subtype(L, union(L_1, L_2))}{\Phi_1 \vee \Phi_2 \vdash_{mem} \mathcal{S} \hookrightarrow L} \; [\textbf{CHECK-OR-MEM}]$$

$$\frac{\Phi_1 = \exists w_1^* \cdot \kappa_1 \wedge \pi_1 \quad \Phi_2 = \exists w_2^* \cdot \kappa_2 \wedge \pi_2 \quad XMem(\kappa_1) = (S_1, L_1) \quad XMem(\kappa_2) = (S_2, L_2) \quad \Phi_1 \vdash^{\kappa}_{V,I} (S = S_1)) * \Delta \quad \Phi_2 \vdash^{\kappa}_{V,I} (S = S_2)) * \Delta \quad subtype(L, union(L_1, L_2)) \wedge subtype(union(L_1, L_2), L)}{\Phi_1 \vee \Phi_2 \vdash_{memE} \mathcal{S} \hookrightarrow L} \; [\textbf{CHECK-OR-MEME}]$$

Fig. 3. Validating the Memory Specification

4 Entailment Proving with Memory Specifications

The starting point of this work is an entailment prover and forward verifier for separation logic [5]. Given formulas Δ_1 and Δ_2, our entailment prover checks if Δ_1 entails Δ_2, that is if in any heap satisfying Δ_1, we can find a sub-heap satisfying Δ_2: $\Delta_1 \vdash^{\kappa}_{V,I} \Delta_2 * \Delta_R$. 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, and I keeps track of history of nodes from the antecedent that are temporarily removed from the antecedent. Note that κ, V and I are derived. The entailment checking procedure is initially invoked with $\kappa = \texttt{emp}, V = [], I = \texttt{emp}$. Besides determining if the entailment relation holds, our entailment prover also infers the residual heap of the entailment, that is a formula Δ_R such that $\Delta_1 = \Delta_2 * \Delta_R$

To support proof search the entailment checking can be generalized to a set of residues.

$$\Delta_1 \vdash^{\kappa}_{V,I} \Delta_2 * \Psi$$

The entailment succeeds when Ψ is non-empty, otherwise it is deemed to have failed. The multiple residual states captured in Ψ signify different search outcomes during proving. Our aim in this section is to adopt the entailment to work with memory specifications so that we can verify programs using different operators ($\wedge, *$ and $\mathbb{A}$). We make use of the $\texttt{Entail}_{\texttt{FA}}(\texttt{u}_\texttt{a}, \texttt{u}_\texttt{c})$ function (given in Fig. 4) which takes as input the annotation from the matched node on antecedent ($\texttt{u}_\texttt{a}$) and the annotation from the consequent($\texttt{u}_\texttt{c}$) to calculate the resultant left over annotation in the antecedent after matching. Henceforth we refer to the antecedent as LHS and the consequent as RHS.

During the entailment we check the subtyping between field annotations ($\texttt{u}_\texttt{a} <: \texttt{u}_\texttt{c}$). In case the subtyping does not hold we mark it as an error and the entailment fails. In addition, each pair of aliased nodes from LHS and RHS are matched up, whenever they are proved identical. In order to handle memory specifications and field annotation we extend the matching to fields using the $\texttt{Entail}_{\texttt{FA}}$ function. The modified entailment rule

[ENT−MATCH−FA] is presented next. The XPure_n function in the rule is from [5], it computes a sound approximate pure formula (π) from a heap formula (κ). As a side note we would like to mention that, we have also checked the soundness of the XPure_n function in Coq. We discovered a bug in the previous paper and pen proof given in [5] (a missing extra condition, $p \neq 0$). Interestingly, this condition is also omitted from the proof in [4]

$$\frac{\begin{array}{c}[\textbf{ENT}-\textbf{MATCH}-\textbf{FA}]\\ \mathit{XPure}_n(\mathtt{p_1}{\mapsto}\mathtt{c}\langle\mathtt{v_1}@\mathtt{u_1^*}\rangle\sharp\kappa_1{\wedge}\pi_1){\Longrightarrow}\mathtt{p_1}{=}\mathtt{p_2}\quad \rho{=}[v_1^*/v_2^*]\\ u_1^* <: u_2^*\quad w^*{=}Entail_{FA}(u_1^*,u_2^*)\\ \kappa_1{\wedge}\pi_1{\wedge}\mathit{freeEqn}(\rho,V){\vdash}^{\kappa\sharp\mathtt{p_1}\mapsto\mathtt{c}\langle\mathtt{v_1}@\mathtt{u_1^*}\rangle}_{V-\{v_2^*\},I\sharp\mathtt{p_1}\mapsto\mathtt{c}\langle\mathtt{v_1}@\mathtt{w^*}\rangle}\rho(\kappa_2{\wedge}\pi_2)*\Delta\end{array}}{\mathtt{p_1}{\mapsto}\mathtt{c}\langle\mathtt{v_1}@\mathtt{u_1^*}\rangle\sharp\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V,I}(\mathtt{p_2}{\mapsto}\mathtt{c}\langle\mathtt{v_2}@\mathtt{u_2^*}\rangle\sharp\kappa_2{\wedge}\pi_2)*\Delta}$$

We check subtyping of each field annotation for the matched node and only then calculate the new annotation for LHS using the $\mathtt{Entail}_{\mathtt{FA}}$ function. The entailment keeps track of nodes with new field annotations in I. These nodes are added back at the end of the entailment using the rule [ENT−EMP−FA].

$$\frac{\begin{array}{c}[\textbf{ENT}-\textbf{EMP}-\textbf{FA}]\\ \rho{=}[0/\mathtt{null}]\\ b{=}(\mathit{XPure}_n(\kappa_1\sharp\kappa{\wedge}\pi_1)\implies\exists V{\cdot}\rho\pi_2)\end{array}}{\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V,I}(\pi_2)*\{\kappa_1\sharp I{\wedge}\pi_1\mid b\}}$$

When a match occurs (rule [ENT−MATCH−FA]) and an argument of the heap predicate 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. This process is formalized by the function *freeEqn* below, where V is the set of existentially quantified variables:

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

The matching rule matches only a single node between LHS and RHS. However due to the presence of sharing operators we may have multiple shared views on the node. After entailment, adding the nodes back with new field annotations may leave multiple nodes representing the same memory. Hence we need a mechanism to combine the shared nodes using field annotations. During the entailment we combine partial fields using field annotations, into a single node so that the [ENT−MATCH−FA] can match them properly. The combination function is called `Compact`, it takes two inputs a formula $\Phi{\equiv}\kappa{\wedge}\pi$ and a set of aliases (generated from the pure formula π). The function compacts the aliased nodes in the heap formula (κ) and adds a pure formula π_{eq} to ($\kappa{\wedge}\pi$). The formula π_{eq} captures the equalities generated due to the compaction process. The compaction helps to bring common nodes from shared heaps and make them into a single node when possible. To compact two field annotations we use the $\mathtt{Join}_{\mathtt{FA}}$ function from Fig. 4. The nodes that cannot be combined are handled by the splitting of entailments as the field annotations suggest they can affect each other. The $\mathtt{Join}_{\mathtt{FA}}$ can now be used to define the `Compact` function as follows.

$$Compact(\kappa \wedge \pi, alias(\pi)) \quad =_{df} \quad split\ \kappa\ into\ \kappa_1 \sharp \kappa_2$$
$$forall\ c_1(v_1@u_1^*)\ from\ \kappa_1\ and\ c_2(v_2@u_2^*)\ from\ \kappa_2$$
$$if\ \sharp\ is\ \wedge\ then\ \pi_{eq} = (c1{=}c2)\ else\ \pi_{eq} = \texttt{true}$$
$$with\ \{c1, c2\}\ in\ alias(\pi)\ do\ the\ following$$
$$w = Join_{FA}(u1, u2) \qquad \pi_{eq} = \pi_{eq} \wedge (v1{=}v2)$$
$$\kappa = replace\ c_1(v_1@u_1^*)\ and\ c_2(v_2@u_2^*)\ in\ \kappa\ with\ \ c_1(v_1@w^*)$$
$$\pi = \pi \wedge \pi_{eq}$$

The following examples illustrate the use of `Compact` function. We denote the alias set with `R` for brevity.

$$Compact(\texttt{x}{\mapsto}\texttt{node}\langle@\texttt{A}, \texttt{p}\rangle * \texttt{x}{\mapsto}\texttt{node}\langle\texttt{d}, @\texttt{A}\rangle, R){=}\texttt{x}{\mapsto}\texttt{node}\langle\texttt{d}, \texttt{p}\rangle$$
$$Compact(\texttt{x}{\mapsto}\texttt{node}\langle\texttt{d}@\texttt{I}, \texttt{p}\rangle{\wedge}\texttt{y}{\mapsto}\texttt{node}\langle\texttt{e}@\texttt{I}, @\texttt{A}\rangle, R){=}\texttt{x}{\mapsto}\texttt{node}\langle\texttt{d}@\texttt{I}, \texttt{p}\rangle{\wedge}d{=}e{\wedge}x{=}y$$

The `Compact` function has the property that for any formula Φ, $\Phi \vdash \texttt{Compact}(\Phi)$. We use $\texttt{Compact}(\Phi)$ as it is much easier and simpler to verify using the underlying entailment procedure. Since we are interested in verifying compatible sharing, we now present a way to detect incompatible formulas using memory specifications. We mark a formula as compatible if the field annotations are disjoint or they use immutability ($@\texttt{I}$) for sharing. All the other combinations of annotations allow exclusive writing and reading on the fields and hence can affect with each other when present in multiple views. The $\texttt{Compatible}_{\texttt{FA}}$ function checks for pairs of annotations that are compatible.

u_a	u_c	$Entail_{FA}$	u_1	u_2	$Compatible_{FA}$	u_1	u_2	$Join_{FA}$
$@M$	$@M$	$@A$	$@M$	$@M$	`false`	$@M$	$@A$	$@M$
$@M$	$@I$	$@A$	$@M$	$@I$	`false`	$@I$	$@I$	$@I$
$@M$	$@A$	$@M$	$@M$	$@A$	`true`	$@I$	$@A$	$@I$
$@I$	$@I$	$@I$	$@I$	$@I$	`true`	$@A$	$@A$	$@A$
$@I$	$@A$	$@I$	$@I$	$@A$	`true`			
$@A$	$@A$	$@A$	$@A$	$@A$	`true`			

Fig. 4. Field Annotations - Entailment, Compatiblity and Join

The $\texttt{Compatible}_{\texttt{FA}}$ function (given in Fig. 4) doesn't depend on the sharing operator as regardless of the sharing of heaps we want to restrict ourselves to shared heaps that do not have exclusive read and write annotations. During the entailment we need to ensure that the memory represented by the heap formula is compatible. For this we use another function ($\texttt{Compatible}(\kappa)$) which takes a κ formula and returns `true` if the formula is compatible. Using the compatibility of field annotations and `XMem` memory approximation we define the $\texttt{Compatible}(\kappa)$ function as follows.

$$Compatible(\kappa) \quad =_{df} \quad split\ \kappa\ into\ \kappa_1 \sharp \kappa_2$$
$$(S_1, L_1){=}XMem(\kappa_1) \quad (S_2, L_2){=}XMem(\kappa_2)$$
$$\forall\ c(@u_1^*)\ in\ L_1, \quad \exists\ c(@u_2^*)\ in\ L_2\ s.t.\ Compatible_{FA}(u_1, u_2)$$
$$\forall\ c(@u_2^*)\ in\ L_2, \quad \exists\ c(@u_1^*)\ in\ L_1\ s.t.\ Compatible_{FA}(u_2, u_1)$$

We are now in position to give the rules for entailment using memory specifications after the matching of nodes is done. The $\texttt{isExact}(\kappa)$ function returns true if the heap κ consists of only data nodes and predicates with exact memory (declared by **memE**). The entailment rules are presented in Fig. 5 and are based on splitting the RHS and LHS.For

the case of the [ENT−SPLIT−RHS] rule, the splitting is performed when encountering a sharing operator ($\mathbb{A}$ and $\wedge$). At that point the heap entailment is divided into three entailments. We entail the first heap to obtain a residue Δ and then the second heap is entailed with the help of the residue. Lastly the pure information from RHS is entailed generating the final residue Δ_2. The rest of the rules for entailment are same as given in [4]. The only difference is that, we now have the operators $\mathbb{A}$ and $\wedge$ which we need to eliminate by splitting.

$$\frac{\begin{array}{c}[\textsc{ent}-\textsc{split}-\textsc{rhs}]\\ \kappa_a{\wedge}\pi_a{\vdash}^{\kappa}_{V,I}\kappa_1 * \Delta\\ \Delta{\vdash}^{\kappa}_{V,I}\kappa_2 * \Delta_1\\ \Delta_1{\vdash}^{\kappa}_{V,I}\pi_c * \Delta_2\end{array}}{\kappa_a{\wedge}\pi_a{\vdash}^{\kappa}_{V,I}\kappa_1\sharp\kappa_2{\wedge}\pi_c * \Delta_2} \qquad \frac{\begin{array}{c}[\textsc{ent}-\textsc{split}-\textsc{lhs}]\\ \texttt{isExact}(\kappa_1{\wedge}\kappa_2)\\ \kappa_a{\wedge}\kappa_b{\wedge}\pi_b{=}\\ \texttt{Compact}(\kappa_1{\wedge}\kappa_2{\wedge}\pi_{\texttt{a}},\texttt{alias}(\pi_{\texttt{a}}))\\ \texttt{Compatible}(\kappa_{\texttt{a}}{\wedge}\kappa_{\texttt{b}})\\ (S_a,L_a){=}\texttt{XMem}(\kappa_{\texttt{a}}) \quad (S_b,L_b){=}\texttt{XMem}(\kappa_{\texttt{b}})\\ \pi_b{\vdash}^{\kappa}_{V,I}S_a{=}S_b * \Delta\\ \kappa_a{\wedge}\pi_b{\vdash}^{\kappa}_{V,I}\kappa{\wedge}\pi_c * \Psi_1\\ \kappa_b{\wedge}\pi_b{\vdash}^{\kappa}_{V,I}\kappa{\wedge}\pi_c * \Psi_2\end{array}}{\kappa_1{\wedge}\kappa_2{\wedge}\pi_a{\vdash}^{\kappa}_{V,I}\kappa{\wedge}\pi_c * (\Psi_1 \cup \Psi_2)}$$

Fig. 5. Entailment with Memory Specifications

For the sharing operators we utilize the proof search capability of our system by trying to entail the heap κ from RHS using both κ_1 and κ_2. The set of residues from proof search is represented by Ψ. The entailment is valid if there is at least one residue in set Ψ that succeeds. In the rule [ENT−SPLIT−LHS] the entailment succeeds if either one of the split entailments with κ_a and κ_b gives rise to a successful entailment, that is if at least one of the residual states Ψ_1 and Ψ_2 is valid. The check for compatibility ensures that after splitting the entailment if either of the entailment succeeds we can say that the original entailment succeeds. For $\wedge$ operator we also check if the memory on both sides of the operator is exact and same. Thus using these rules we can eliminate the sharing operators and reduce the entailment to one which can be handled by the system using only $*$ operator.

5 Forward Verification with Compatible Sharing

Our core imperative language is presented in Fig. 6. A program `P` comprises of a list of data structure declarations `tdecl`* and a list of method declarations `meth`*. Data structure declaration can be a simple node `datat` or a recursive shape predicate declaration `spred` as described in Section 3. A method is declared with a prototype, its body `e`, and multiple specification `mspec`*. The prototype comprises a method return type, method name and method's formal parameters. The parameters can be passed by *value* or by *reference* with keyword `ref` and their types can be primitive τ or user-defined `c`. With the pre/post conditions declared for each method in P, we can now apply modular verification to its body using Hoare-style triples $\vdash \{\Delta_1\}e\{\Delta_2\}$. We expect Δ_1 to be given before computing Δ_2 since the rules are based on a forward verifier. To capture proof search, we generalize the forward rule to the form $\vdash \{\Delta_1\}e\{\Psi\}$ where Ψ is a set of heap states, discovered by a search-based verification process. When Ψ is empty, the forward verification is said to have failed for Δ_1 as prestate.

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

Fig. 6. A Core Imperative Language

We now present the modifications needed to do forward verification with compatible sharing. As most of the forward verification rules are standard [4], we only provide those for field read and field update. Verification of a method starts with each precondition, and proves that the postcondition is guaranteed at the end. We need to modify the rules related reading and updating of fields to the following.

$$
\frac{\begin{array}{c}[\underline{\mathbf{FV-[FIELD-READ]}}] \\ \Delta \vdash^{\kappa}_{V,I} \mathtt{v}' \mapsto \mathtt{c}\langle \mathtt{v_1@A}, \mathtt{v_2@A}, .., \mathtt{v_i@I}, .., \mathtt{v_n@A}\rangle * \Psi_1 \quad fresh\ v_1..v_n \quad \Psi_1 \neq \{\} \\ \Psi_2 = \exists v_1..v_n \cdot (\Delta \wedge \mathtt{res}{=}v_i)\end{array}}{\vdash \{\Delta\} v.f_i \{\Psi_2\}}
$$

$$
\frac{\begin{array}{c}[\underline{\mathbf{FV-[FIELD-UPDATE]}}] \\ \Delta \vdash^{\kappa}_{V,I} \mathtt{v}' \mapsto \mathtt{c}\langle \mathtt{v_1@A}, \mathtt{v_2@A}, .., \mathtt{v_i@M}, .., \mathtt{v_n@A}\rangle * \Psi_1 \quad fresh\ v_1..v_n \quad \Psi_1 \neq \{\} \\ \Psi_2 = \exists v_1..v_n \cdot (\Delta * [v_0'/v_i]\mathtt{v}' \mapsto \mathtt{c}\langle \mathtt{v_1@A}, \mathtt{v_2@A}, .., \mathtt{v_i@M}, .., \mathtt{v_n@A}\rangle)\end{array}}{\vdash \{\Delta\} v.f_i{:=}v_0 \{\Psi_2\}}
$$

Whenever there is a field access (read or update), the current state, Δ, must contain the node to be dereferenced. For $[\underline{\mathbf{FV-[FIELD-READ]}}]$ only the field that is been read is marked with @I annotation. In case of $[\underline{\mathbf{FV-[FIELD-UPDATE]}}]$ the field that is updated is marked with the mutable annotation (@M). As shown in the $\texttt{Entail}_{\texttt{FA}}$ rule from section 4 entailing a @I or @M field from RHS with a corresponding node on LHS will consume the field from LHS. Hence, we discard the residue from the entailment (Ψ_1) and instead keep the original field annotation on LHS (Δ) so as prevent it from getting consumed.

6 Experiments

We have built a prototype system using Objective Caml called HIPComp[2]. The web interface of HIPComp allows testing the examples without downloading or installing the system. The proof obligations generated by HIPComp are discharged using off-the-shelf constraint solvers (Omega Calculator [12] and Mona [13]). In addition to the examples presented in this paper we can do automated verification of a number of challenging data structures with complex sharing. The examples are hard to reason with separation logic due to inherit sharing and aliasing in heap. We use overlaid conjunction ($\mathbb{A}$) to concisely capture safety properties of programs, as seen by the following invariants verified in our experiments.

$\texttt{al}\langle \texttt{x}\rangle\ \mathbb{A}\ (\texttt{rl}\langle \texttt{y}\rangle * \texttt{sl}\langle \texttt{z}\rangle)$ //*Process Scheduler*
$\texttt{llnext}\langle \texttt{x}\rangle\ \mathbb{A}\ \texttt{lldown}\langle \texttt{y}\rangle$ //*Doubly Circular List*
$\texttt{ll}\langle \texttt{x}\rangle\ \mathbb{A}\ \texttt{tree}\langle \texttt{t}\rangle$ //*LL (Linked List) and Tree*
$\texttt{ll}\langle \texttt{x}\rangle\ \mathbb{A}\ \texttt{sll}\langle \texttt{y}\rangle$ //*LL and SortedLL*
$(\texttt{ll}\langle \texttt{x}\rangle\ \mathbb{A}\ \texttt{tree}\langle \texttt{t}\rangle) * \texttt{ll}\langle \texttt{y}\rangle$ //*Disk IO Scheduler*

[2] Available at http://loris-7.ddns.comp.nus.edu.sg/~project/HIPComp/

The following table summarizes the suite of examples verified by HIPComp. All experiments were done on a 3.20GHz Intel Core i7-960 processor with 16GB memory running Ubuntu Linux 10.04. The first column gives the name of the program, second column lists the lines of code (including specifications) in the program. In the third column we show the sharing degree, it is defined as the percentage of specifications that use compatible sharing using field annotations. As is clear from our benchmark programs, the ability to specify sharing is important to verify these data structures. The last column is the percentage of total entailments generated that are split using the compatible entailment rules. The compatibility check is essential to verify sharing in these programs.

Program	*LOC*	*Timings [secs]*	*Sharing [%]*	*Compatibility [%]*
Parameterized List	30	0.28	0	0
Compatible Pairs	12	0.09	100	25
LL and SortedLL	175	0.61	22	22
LL and Tree	70	0.24	16	7
Doubly Circular List	50	0.41	50	32
Process Scheduler	70	0.47	33	23
Disk IO Scheduler	88	1.3	16	27

7 Related Work and Conclusions

Our sharing and aliasing logic is most closely related to Hobor and Villard [10]. They present the RAMIFY rule of separation logic and show how to mechanically reason with graphs, dags and overlaid structures. Our work can be seen as a specific instance where we seek to automatically verify programs with compatible sharing. The problem of sharing has also been explored in the context of concurrent data structures and objects [6,18]. Our work is influenced by them but for a sequential setting, indeed the notion of self-stable concurrent abstract predicates is analogous to our condition for compatibility. However since we are focused on sequential programs, we avoid the use of environment actions and instead focus on checking compatibility between shared predicates. Regional logic [1] also uses a notion of set of addresses as footprint of formulas. These regions are used with dynamic frames to enable local reasoning of programs. Memory layouts [8] were used by Gast, as a way to formally specify the structure of individual memory blocks.

In the area of program analysis the work most closely related to ours is by Lee et al. [15] on overlaid data structures. They show how to use two complementary static analysis over different shapes and combine them to reason about overlaid structures. Their shape analysis uses the $\wedge$ operator in the abstract state to capture the sharing of heaps in overlaid structures, but they do not provide a general way to reason with shared heaps. In contrast, we verify that the shared heaps used by the predicates are compatible with each other. This enables us to check that the updates to shared regions are restricted to corresponding predicates. Similarly, the recent work of Dragoi et al. [7] considers only the shape analysis of overlaid lists. We extend these separation logic based techniques by going beyond shape properties and handling arbitrary data structures. Our proposal is built on top of user defined predicates with shape, size and bag properties

that can express functional properties (order, sorting, height balance etc.) of overlaid data structures.

We have proposed a specification mechanism to express different kinds of sharing and aliasing in data structures. The specifications can capture correctness properties of various kinds of programs using compatible sharing. We present an automated verification system which can be used to reason about sharing in data structures. We have implemented a prototype based on our approach. An initial set of experiments with small but challenging programs have confirmed the usefulness of our method.

References

1. Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, pages 387–411, 2008.
2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, Springer LNCS 4111, pages 115–137, 2006.
3. John Tang Boyland. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.*, 32(6), 2010.
4. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.
5. Cristina David and Wei-Ngan Chin. Immutable specifications for more concise and precise verification. In *OOPSLA*, pages 359–374, 2011.
6. Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.
7. Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Local shape analysis for overlaid data structures. In *SAS*, pages 150–171, 2013.
8. Holger Gast. Reasoning about memory layouts. In *FM*, pages 628–643, 2009.
9. A. Gotsman, J. Berdine, and B. Cook. Interprocedural Shape Analysis with Separated Heap Abstractions. In *SAS*, Springer LNCS, pages 240–260, Seoul, Korea, August 2006.
10. Aquinas Hobor and Jules Villard. The Ramifications of Sharing in Data Structures. In *POPL*, 2013.
11. S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM POPL*, pages 14–26, London, January 2001.
12. P. Kelly, V. Maslov, W. Pugh, and et al. *The Omega Library Version 1.1.0 Interface Guide*, November 1996.
13. N. Klarlund and A. Moller. MONA Version 1.4 - User Manual. BRICS Notes Series, January 2001.
14. Xuan Bach Le, Cristian Gherghina, and Aquinas Hobor. Decision procedures over sophisticated fractional permissions. In *APLAS*, pages 368–385, 2012.
15. Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program analysis for overlaid data structures. In *CAV*, pages 592–608, 2011.
16. P. W. O'Hearn, J. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In *CSL*, September 2001.
17. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, pages 55–74, 2002.
18. Aaron Joseph Turon and Mitchell Wand. A separation logic for refining concurrent objects. In *POPL*, pages 247–258, 2011.

A Appendix

In this section we present the soundness properties for both the forward verifier with compatible sharing and the entailment prover with memory specifications.

A.1 Storage Model

The storage model is similar to classical separation logic [17], with the difference that we support field annotations, memory specifications and sharing operators. Accordingly, we define our storage model by making use of a domain of heaps, which is equipped with a partial operator for gluing together disjoint heaps. $\mathtt{h}_0 \cdot \mathtt{h}_1$ takes the union of partial functions when $\mathtt{h}_0$ and $\mathtt{h}_1$ have disjoint domains of definition, and is undefined when $\mathtt{h}_0(\mathtt{l})$ and $\mathtt{h}_1(\mathtt{l})$ are both defined for at least one location $\mathtt{l} \in \mathit{Loc}$.

To define the model we assume sets *Loc* of locations (positive integer values), *Val* of primitive values, with $0 \in \mathit{Val}$ denoting `null`, *Var* of variables (program 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 $f_1, .., f_n$. Each field has an attached annotation from $\{M, I, A\}$. I means that the corresponding field value cannot be modified, while M allows its mutation, and A denotes no access.

$$h \in \mathit{Heaps} =_{df} \mathit{Loc} \rightharpoonup_{fin} \mathit{ObjVal} \times \{M, I, A\}$$
$$s \in \mathit{Stacks} =_{df} \mathit{Var} \rightarrow \mathit{Val} \cup \mathit{Loc}$$

Note that each heap h is a finite partial mapping while each stack s is a total mapping, as in the classical separation logic [17,11].

A.2 Semantic Model of the Specification Formula

The semantics of our separation heap formula is similar to the model given for separation logic [17], except that we have extensions to handle our user-defined heap predicates together with the field annotations and new sharing operators. Let $s, h \models \Phi$ denote the model relation, i.e. the stack s and heap h satisfy the constraint Φ. Function $\mathit{dom}(f)$ returns the domain of function f. Now we use $\mapsto$ to denote mappings, not the points-to assertion in separation logic. The model relation for separation heap formulae is given in Def 1. The model relation for pure formula $s \models \pi$ denotes that the formula π evaluates to `true` in s.

The last case in Def 1 is split into two cases: (1) c is a data node defined in the program *P*; (2) c is a heap predicate defined in the program *P*. In the first case, h has to be a singleton heap. In the second case, the heap predicate c may be inductively defined. Note that the semantics for an inductively defined heap predicate denotes the least fixpoint, i.e., for the set of states (s, h) satisfying the predicate. The monotonic nature of our heap predicate definition guarantees the existence of the descending chain of unfoldings, thus the existence of the least solution.

The soundness of our verification rules is defined with respect to the small-step operational semantics. We need to extract the post-state of a heap constraint by function $Post(\Delta)$ defined in Def 2.

Definition 1 (Model for Specification Formula)

$$
\begin{array}{ll}
s,h \models \Phi_1 \vee \Phi_2 & \texttt{iff } s,h \models \Phi_1 \textit{ or } s,h \models \Phi_2 \\
s,h \models \exists v_{1..n} \cdot \kappa \wedge \pi & \texttt{iff } \exists \nu_{1..n} \cdot s[v_1 \mapsto \nu_1, .., v_n \mapsto \nu_n], h \models \kappa \textit{ and} \\
 & \quad s[v_1 \mapsto \nu_1, .., v_n \mapsto \nu_n] \models \pi \\
s,h \models \kappa_1 * \kappa_2 & \texttt{iff } \exists h_1, h_2 \cdot h_1 \perp h_2 \textit{ and } h = h_1 \cdot h_2 \textit{ and} \\
 & \quad s,h_1 \models \kappa_1 \textit{ and } s,h_2 \models \kappa_2 \\
s,h \models \kappa_1 \wedge \kappa_2 & \texttt{iff } s,h \models \kappa_1 \textit{ and } s,h \models \kappa_2 \\
 & \\
s,h \models \texttt{emp} & \texttt{iff } \mathit{dom}(h) = \emptyset \\
s,h \models \texttt{c(p, v}_{1..n}\texttt{@u}_{1..n}\texttt{)} & \texttt{iff data } c\ \{t_1\ f_1, .., t_n\ f_n\} \in P,\ h = [s(p) \mapsto r], \mathit{dom}(h) = \{x\} \\
 & \quad \textit{and } r = c[f_1 \mapsto_{w_1} s(v_1), .., f_n \mapsto_{w_n} s(v_n)] \textit{ and } u_i <: w_i \\
 & \quad \texttt{or } (c\langle v_{1..n}\rangle \equiv \Phi\ \texttt{inv}\ \pi) \in P \textit{ and } s,h \models [p/\texttt{root}]\Phi
\end{array}
$$

Definition 2 (Poststate) *Given a constraint Δ, Post(Δ) captures the relation between primed variables of Δ. That is:*

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

A.3 Dynamic Semantics

This section presents a small-step operational semantics for our language. The rules are given in Fig. 7. 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. Note that the operational semantics must consider the field annotations of assertions recorded in the heap h. Each reduction step is formalized as a transition of the form: $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$. 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.

We now explain the notations used in the operational semantics. We make use of k to denote a constant, $\perp$ to denote an undefined value, and $()$ to denote the empty expression (program). The operation $[v \mapsto \nu] + s$ "pushs in" the variable v to s with the value ν, such that $([v \mapsto \nu] + s)(v) = \nu$. 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 operation $h(l)[v_n \mapsto_u t_n]$ updates the field v_n of the object stored at location l in the heap h with the value t_n, with u as the attached field annotation, $u \in \{M, I, L, A\}$. If no field annotation is present, then the mapping $h(l)[v_n \mapsto t_n]$ maintains the previous field annotation. We also make use of the function $type(v)$ to get the run-time type of the variable v.

In order to relate the logic with the storage model, we introduce an auxiliary construct, $\texttt{assert}\ \Phi$, which checks whether the model relation $s,h \models \Phi$ holds. The forward verification rule for this intermediate construct is given below.

$$
\frac{[\mathbf{FV}\text{-}[\mathbf{ASSERT}]] \quad \Delta \vdash^{\kappa}_{V,I} \Phi_1 * \Psi_1 \quad \Psi_1 \neq \emptyset}{\vdash \{\Delta\}\ \texttt{assert}\ \Phi_1\ \{\Delta\}}
$$

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

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

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

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

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

$$\langle s,h,\texttt{ret}(v^*,k)\rangle\hookrightarrow\langle s{-}\{v^*\},h,k\rangle$$

$$\frac{\langle s,h,e\rangle\hookrightarrow\langle s_1,h_1,e_1\rangle}{\langle s,h,\texttt{ret}(v^*,e)\rangle\hookrightarrow\langle s_1,h_1,\texttt{ret}(v^*,e_1)\rangle}$$

$$\frac{\begin{array}{c}type(v_1)=c\langle w_1,..,w_n\rangle\\ r{=}h(s(v_1))[f{\mapsto}s(v_2)] \quad h_1{=}h[s(v_1){\mapsto}r]\end{array}}{\langle s,h,v_1.f_i:=v_2\rangle\hookrightarrow\langle s,h_1,()\rangle}$$

$$\frac{s,h\models\Phi}{\langle s,h,\texttt{assert } \Phi\rangle\hookrightarrow\langle s,h,()\rangle}$$

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

$$\frac{\begin{array}{c}t_0\ mn((\texttt{ref}\ t_j\ w_j)_{j=1}^{m-1},(t_j\ w_j)_{j=m}^{n})\ \{\texttt{requires}\ \Phi_{pr}^i\ \texttt{ensures}\ \Phi_{po}^i\}_{i=1}^{p}\ \{e\}\\ s_1{=}[w_j{\mapsto}s(v_j)]_{j=m}^{n}{+}s \qquad \langle s_1,h,(\texttt{assert}\ \Phi_{pr}^i)_{i=1}^{p}\rangle\hookrightarrow\langle s_1,h,()\rangle\\ \langle s_1,h,\texttt{ret}(\{w_j\}_{j=m}^{n},[v_j/w_j]_{j=1}^{m-1}e;(\texttt{assert}\ \Phi_{po}^i)_{i=1}^{p})\rangle\hookrightarrow\langle s,h_1,e_1\rangle\end{array}}{\langle s,h,mn(v^*)\rangle\hookrightarrow\langle s,h_1,e_1\rangle}$$

Fig. 7. Small-Step Operational Semantics

The connection between the field annotation information from the specification and the storage model is established at the beginning and end of each method execution, when the `assert` construct is used to check that the stack s and the heap h model the method's precondition and postcondition, respectively. Note that in the rule for `new` $c(v^*)$ we assume that the fields of the newly created object are mutable and attach the M annotation.

Theorem 1 (Preservation) *If*

$$\vdash\{\Delta\}\ e\ \{\Psi_2\} \quad \Psi_2{\neq}\{\} \quad s,h\models Post(\Delta) \quad \langle s,h,e\rangle\hookrightarrow\langle s_1,h_1,e_1\rangle$$

Then there exists $\Psi_1{\neq}\{\}$*, such that* $\forall\Delta_1\in\Psi_1\cdot s_1,h_1\models Post(\Delta_1)$ *and* $\vdash\{\Delta_1\}\ e_1\ \{\Psi_3\} \quad \Psi_3{\subseteq}\Psi_2$.

Proof: By structural induction on e.

- Case $v := e_3$. There are two cases according to the dynamic semantics:
  - e_3 is not a value. Then, from [[FV–ASSIGN]] in [4],
    if $\vdash\{\Delta\}e_3\{\Psi_3\}$, then $\Psi_2{=}\exists\texttt{res}\cdot\Psi_3\wedge_{\{v\}}v'{=}\texttt{res}$.
    From dynamic rules, if $\langle s,h,e_3\rangle\hookrightarrow\langle s_4,h_4,e_4\rangle$, then
    $\langle s,h,v{:=}e_3\rangle\hookrightarrow\langle s_4,h_4,v{:=}e_4\rangle$.
    From $\langle s,h,e_3\rangle\hookrightarrow\langle s_4,h_4,e_4\rangle$ and $\vdash\{\Delta\}e_3\{\Psi_3\}$, by induction hypothesis, we have that there exists S_4 such that $\forall\Delta_4\in S_4\cdot s_4,h_4\models Post(\Delta_4)$ and $\vdash\{\Delta_4\}e_1\{\Psi_3\}$.
    Then, from verification rule [[FV–ASSIGN]] in [4], $\forall\Delta_4\in\Psi_4\cdot\vdash\{\Delta_4\}v{:=}e_1\{\Psi_2\}$.

Hence, the proof is complete for $h_1{=}h_4$, $s_1{=}s_4$, and $S_1{=}S_4$.
- e_3 is a value. Let $e_1{=}()$, $\Psi_1{=}rn_2{=}\{(\Delta \wedge v'{=}e_3)\}$, $h_1{=}h_2{=}h$, and $s_1{=}s_2{=}s[v \mapsto e_3]$. Straightforward.

– Case $v_1.f := v_2$. Take $e_1{=}()$, $\Psi_1{=}\Psi_2$, $h_1{=}h_2$, and $s_1{=}s_2{=}s$. Hence, it is straightforward that $\forall \Delta_1 \in \Psi_1 \cdot \{\Delta_1\}\ e_1\ \{\Psi_2\}$. Now, we must show that $\forall \Delta_1 \in \Psi_1 \cdot s_1, h_1 \models \mathit{Post}(\Delta_1)$.
We have $s{=}s_1$ and $h_1{=}h[s(v_1){\mapsto}r]$, where $r{=}h(s(v_1))[f{\mapsto}s(v_2)]$.
From the rule [FV–FIELD–UPDATE] in Sec 5,
we have $\Delta \vdash^{\kappa}_{V} \mathtt{v}'_1{::}\mathtt{c}\langle \mathtt{w_1@A}, .., \mathtt{w_f@M}, .., \mathtt{w_n@A}\rangle * \Psi_3$, $\Psi_3{\neq}\emptyset$
and $\Psi_1 = \exists w_1..w_n \cdot (S_3 * [v'_2/w_f]\mathtt{v}'_1{::}\mathtt{c}\langle \mathtt{w_1@A}, .., \mathtt{w_f@M}, .., \mathtt{w_n@A}\rangle)$ (*).
From the hypothesis, we know that $s, h \models \mathit{Post}(\Delta)$ (**).
From (*) and (**), by Theorem 3, $\forall\ \Delta_3 \in \Psi_3.s, h \models \mathit{Post}(\Delta_3)$.
From the dynamic rules for $\mathtt{assert}\ \mathtt{v_1{::}c}\langle \mathtt{w_1@A}, .., \mathtt{w_f@M}, .., \mathtt{w_n@A}\rangle$,
we have $s, h \models \mathtt{v_1{::}c}\langle \mathtt{w_1@A}, .., \mathtt{w_f@M}, .., \mathtt{w_n@A}\rangle$.
Hence, the conclusion follows from $s{=}s_1$ and $h_1{=}h[s(v_1){\mapsto}r]$, where $r{=}h(s(v_1))[f{\mapsto}s(v_2)]$.

– Case $\mathtt{new}\ c(v^*)$. From verification rule [FV–NEW] in [4], we have $\vdash \{\Delta\}\mathtt{new}\ c(v^*)\{\Psi_2\}$, where $S_2 = \{\Delta * \mathtt{res}{::}c\langle v'_1, .., v'_n\rangle\}$. Let $\Psi_1 = \Psi_2$. From the dynamic semantics, we have
$\langle s, h, \mathtt{new}\ c(v^*)\rangle \hookrightarrow \langle s, h{+}[\iota \mapsto r], \iota\rangle$, where $\iota \notin \mathit{dom}(h)$.
From $s, h \models \mathit{Post}(\Delta)$, we have $\forall \Delta_1 \in \Psi_1.s, h{+}[\iota \mapsto r] \models \mathit{Post}(\Delta_1)$. Moreover, $\vdash \{\Psi_1\}\iota\{\Psi_2\}$.

– 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] in [4], we have $\vdash \{\Delta\}e_1\{\Psi_3\}$. By induction hypothesis, there exists Ψ_1 s.t. $\forall \Delta_1 \in \Psi_1.s_1, h_1 \models \mathit{Post}(\Delta_1)$ and $\vdash \{\Psi_1\}e_3\{\Psi_3\}$. By rule [FV–SEQ] , we have $\vdash \{\Psi_1\}e_3; e_2\{\Psi_2\}$.

– Case $\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$. There are two possibilities in the dynamic semantics:
- $s(v){=}\mathtt{true}$. We have $\langle s, h, \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rangle \hookrightarrow \langle s, h, e_1\rangle$. Let $\Psi_1 = \{\Delta{\wedge}v'\}$. It is obvious that $\forall\ \Delta_1 \in S_1.s, h \models \mathit{Post}(\Delta_1)$. By the rule [FV–IF] in [4], we have $\vdash \{\Delta{\wedge}v'\}\ e_1\ \{\Psi^1\}$. From the same rule $\Psi_2 = \Psi^1 \vee \Psi^2$, where $\vdash \{\Delta{\wedge}{\neg}v'\}\ e_2\ \{\Psi^2\}$. By weakening the postcondition (rule [FV–POST–WEAKENING] in [4], we get $\vdash \{\Delta{\wedge}{\neg}v'\}\ e_1\ \{\Psi_2\}$, which is $\vdash \{\Psi_1\}\ e_1\ \{\Psi_2\}$.
- $s(v) = \mathtt{false}$. Analogous to the above.

– Case $t\ v;\ e$. Let $\Psi_1 = \{\Delta\}$. From the dynamic rules in fig 7, we have $e_1 = \mathtt{ret}(v, e)$, $s_1 = [v{\mapsto}\bot]{+}s$, $h_1 = h$. We conclude immediately from the assumption and the rules [FV–LOCAL] and [FV–RET] in [4].

– Case $mn(v_{1..n})$. From the dynamic rule for method call in Fig 7, $s_1{=}[w_j{\mapsto}s(v_j)]^n_{j=m}{+}s$. h_1 is the same as h, with the difference that it has been decorated with the immutability annotations by the instruction $\mathtt{assume}\ \mathit{heap}(\Phi^i_{pr})$ (the operational semantics must consider the mutability assertions recorded in the heap h). From rule [FV–CALL] in Sec 5, we know $\Delta{\vdash}\rho\,\Phi^i_{pr} * \Psi_i$. Take $\Psi_1 = \rho\,\Phi^i_{pr}{*}\Psi_i$. From the hypothesis and Theorem 3, we have $\forall \Delta_1 \in \Psi_1.s_1, h_1 \models \mathit{Post}(\Delta_1)$. From rule [FV–CALL], we have $\Psi_2 = \bigcup^p_{i=1} \Psi_i * \Phi^i_{po}$, while from the dynamic semantics we have
$e_1 = \mathtt{ret}(\{w_j\}^n_{j=m}, [v_j/w_j]^{m-1}_{j=1}e)$.
From rule [FV–METH], we have $\vdash \{\rho\,\Phi_{pr}{*}\Psi_i\}\ e_1\ \{\Psi_i{*}\Phi_{po}\}$ which concludes proof for this case.

– Case $\mathtt{ret}(v^*, e)$. There are two cases:
- e is a value k. Let $\Psi_1 = \{\exists v'^* \cdot \Delta\}$. 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 Ψ_1 s.t. $\forall\Delta_1\in\Psi_1.s_1,h_1\models \mathit{Post}(\Delta_1)$ and $\vdash\{\Psi_1\}\,e_1\,\{\Psi_3\}$, and $\Psi_2=\exists v'^*.\Psi_3$. By rule [FV−RET] again, we have $\vdash\{\Psi_1\}\,\texttt{ret}(v^*,e_1)\,\{\Psi_2\}$.

– Case `assume` κ. Take $e_1{=}()$, $\Psi_1{=}\Psi_2{=}\{assume\ FA(\Delta,\mathit{XPure}_n(\Delta),\kappa)\}$, $s_1{=}s_2{=}s$, and h_1 the same as h, with the difference that it has been decorated with the field annotations from κ according to the dynamic semantics. Hence, it is straightforward that $\forall\Delta_1\in\Psi_1{\cdot}\ \{\Delta_1\}\,e_1\,\{\Psi_2\}$, and that $\forall\Delta_1\in\Psi_1{\cdot}s_1,h_1\models \mathit{Post}(\Delta_1)$.
– Case `null` | k | v | $v.f$ | `assert` κ. Straightforward.

Theorem 2 (Progress) *If*

$$\vdash\{\Delta\}\ e\ \{\Psi_1\}\quad \Psi_1{\neq}\{\}\quad s,h\models \mathit{Post}(\Delta)$$

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.

– Case $v:=e$. There are two cases:
- e is a value k. We conclude.
- e is not a value. By [FV−ASSIGN] in [4], we have $\vdash\{\Delta\}\,e\,\{\Psi_2\}$. 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 $v_1.f:=v_2$. From [FV−FIELD−UPDATE] in Sec 5, we know
$\Delta\vdash^{\kappa}_{V}\mathtt{v}'{::}\mathtt{c}\langle\mathtt{v_1}@\mathtt{A},..,\mathtt{v_i}@\mathtt{M},..,\mathtt{v_n}@\mathtt{A}\rangle*\Psi_1,\ ,\ S_1{\neq}\emptyset$, for fresh $v_1,..,v_n$.
Hence, `assert` $\mathtt{v_1}{::}\mathtt{c}\langle\mathtt{w_1}@\mathtt{A},..,\mathtt{w_i}@\mathtt{M},..,\mathtt{w_n}@\mathtt{A}\rangle$ in the dynamic semantics in Fig 7 will go through. Furthermore, 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 `new` $c(v_{1\cdots n})$. Let ι be a fresh location, r denotes the object value $c[f_1{\mapsto}s(v_1),..,f_n{\mapsto}s(v_n)]$. Take $s_1=s$, $h_1=h{+}[\iota{\mapsto}r]$, and $e_1=\iota$. We conclude.
– Case $e_1;e_2$. If e_1 is a value $()$, we conclude immediately by taking $s_1=s,h_1=h$. Otherwise, by induction hypothesis, there exist s_1,h_1,e_3 s.t. $\langle s,h,e_1\rangle{\hookrightarrow}\langle s_1,h_1,e_3\rangle$. We then have
$\langle s,h,e_1;e_2\rangle{\hookrightarrow}\langle s_1,h_1,e_3;e_2\rangle$ from the dynamic semantics.
– Case `if` v `then` e_1 `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 $s_1=[w_j{\mapsto}s(v_j)]^n_{j=m}{+}s$, h_1 is equal to h decorated with the field annotations from the precondition Φ^i_{pr}, and $e_1=\texttt{ret}(\{w_j\}^n_{j=m},[v_j/w_j]^{m-1}_{j=1}e)$, where w_j are from method specification
$t_0\ mn((\texttt{ref}\ t_j\ w_j)^{m-1}_{j=1},(t_j\ w_j)^n_{j=m})$ `where` $\{\Phi^i_{pr}\mathrel{*}{\mapsto}\Phi^i_{po}\}^p_{i=1}\ \{e\}$.
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 $\texttt{assume}\ \kappa$. If $\kappa = \kappa_1 \sharp \kappa_2$, where $\sharp \in \{\wedge, \mathbb{A}, *, \uplus\}$, then take $s_1 = s$, $h_1 = h$, and $e_1 = \texttt{assume}\ \kappa_1; \texttt{assume}\ \kappa_2$. We conclude by the dynamic semantics. On the other hand, if $\kappa = \texttt{p::c}\langle \texttt{v}_1@\texttt{u}_1, .., \texttt{v}_\texttt{n}@\texttt{u}_\texttt{n}\rangle$, then we can conclude by taking $h_1{=}h[s(p){\mapsto}r]$, where $r{=}h(s(p))[v_i{\mapsto}_{u_i}\_]$, $s_1 = s$, and $e_1 = ()$.
- Case $\texttt{null} \mid k \mid v \mid v.f \mid \texttt{assert}\ \kappa$. Straightforward.

Theorem 3 (Soundness of Heap Entailment) *If entailment check $\Delta_1 \vdash \Delta_2 * \Psi$ succeeds, we have: for all $s, h,$ and $\Delta \in \Psi$, if $s, h \models \Delta_1$ then $s, h \models \Delta_2 *_i \Delta$.*

Proof: Note that the entailment rules [[ENT−MATCH−FA]] in Sec 4 denote 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, meaning that the consequent consists of only a pure formula. It can only be the case where rule [[ENT−EMP−FA]] 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−FA]]) 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]], [[ENT−RHS−EX]], [[ENT−SPLIT−RHS]], [[ENT−SPLIT−LHS$_1$]], [[ENT−SPLIT−LHS$_2$]], [[ENT−SPLIT−LHS$_3$]] [[ENT−NONINTER−LHS]]. A case analysis on these rules shows that the following properties hold:

$$\begin{array}{l} s,h \models \mathit{LHS}(\mathcal{E}_i) \implies s,h \models \mathit{LHS}(\mathcal{E}_{i+1})\ (LHS\ is\ always\ weakened\ by\ the\ entailment\ rules) \\ s,h \models \mathit{RHS}(\mathcal{E}_{i+1}) \implies s,h \models \mathit{RHS}(\mathcal{E}_i)\ (RHS\ is\ always\ strengthened\ by\ the\ entailment\ rules) \end{array} \quad (\dagger)$$

Now for a match between the antecedent and the consequent, suppose the matched 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 $(\dagger)$.