Specifying Compatible Sharing in Data Structures

Asankhaya Sharma[1], Aquinas Hobor[2 1], and Wei-Ngan Chin[1]

[1] School of Computing,
[2] Yale-NUS College,
National University of Singapore
{asankhs,hobor,chinwn}@comp.nus.edu.sg

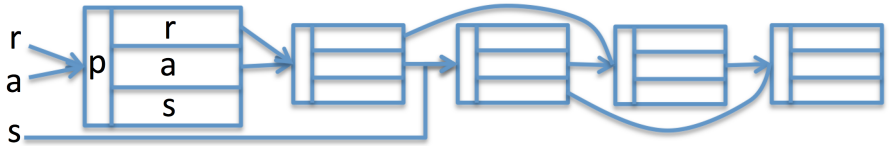
Abstract. Automated verification of programs that utilize data structures with intrinsic sharing is a challenging problem. We develop an extension to separation logic that can reason about aliasing in heaps using a notion of *compatible sharing*. Compatible sharing can model a variety of fine grained 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 non-trivial sharing. We benchmarked our prototype with examples derived from practical algorithms found in systems code, such as those using threaded trees and overlaid data structures.

1 Introduction

Systems software often uses *overlaid data structures* to utilize memory more efficiently and boost performance. Consider maintaining the set of processes in an operating system; some are running while others are sleeping. Sometimes we wish to access every process, whereas other times we only wish to access *e.g.* the running processes. To track this set efficiently we can use a "threaded list" whose nodes are defined as follows:

```
data node { int pid; node anext; node rnext; node snext }
```

Each `node` has four fields. The first two are straightforward: a process id field `pid` and a pointer to the next process (which may be running or sleeping) `anext`. The latter two are a bit trickier. When a process is running, `rnext` points to the next running process in the list, skipping over any sleeping processes in between. When a process is sleeping, `snext` points to the next sleeping process in the list. We maintain three external pointers into the structure: one for the head of the entire list `a`, the second for the head of the running sublist `r`, and the third for the head of the sleeping sublist `s`. Consider this picture:



The efficiency benefits of overlaid structures can be significant: *e.g.*, we avoid representing nodes on multiple spatially-disjoint lists and can visit each running process without needing to step through the sleeping processes. The real drawback, from our perspective, is that programs utilizing overlaid structures are difficult to verify formally!

Separation logic [18] enables compositional reasoning of heap-manipulating programs and has been widely applied to automated verification [5,2,11]. Separation logic uses the separating conjunction $*$ to connect assertions valid on disjoint portions of the heap, enabling natural representations for many data structures such as lists and trees because their constituent subparts are spatially disjoint. Overlaid data structures cannot be specified so naturally because the underlying structures share nodes in memory.

We extend the notion of separation to enable local reasoning for overlaid data structures by introducing a notion of *compatibility*. Two predicates are compatible when updates to one will not affect the other despite spatial overlap. In our threaded list example above, we can imagine splitting the structure into three pseudo-disjoint/compatible lists formed by the `anext`, `rnext`, and `snext` pointer chains. A function that modifies some chains but not others can then "frame away" the part of the structure it does not use. This can happen in several steps: consider switching a process from sleeping to running. First we frame away the `anext` chain. Then we frame away the `rnext` chain, leaving only a straightforward `snext` linked list, on which we do a standard list remove. We then frame `rnext` back in and `snext` away, followed by a standard list add. Finally, we frame `snext` and `anext` back in, restoring the entire structure.

All of the above means we need field-level separation, which we get by adding annotations to fields: when a field is absent (or inaccessible) we mark it with `@A`; when it is present/mutable we mark it with `@M`. Here is how this looks for our threaded list:

$$\begin{aligned}
&\mathtt{al}\langle \mathtt{root}, \mathtt{S}\rangle \equiv (\mathtt{root} = \mathtt{null} \wedge \mathtt{S} = \{\}) \\
&\quad \vee \exists\, \mathtt{p}, \mathtt{a}, \mathtt{S_a} \cdot (\mathtt{root} \mapsto \mathtt{node}\langle \mathtt{p@I}, \mathtt{a@M}, \mathtt{@A}, \mathtt{@A}\rangle * \mathtt{al}\langle \mathtt{a}, \mathtt{S_a}\rangle \wedge \mathtt{S} = \mathtt{S_a} \cup \{\mathtt{root}\}) \\
&\mathtt{rl}\langle \mathtt{root}, \mathtt{S}\rangle \equiv (\mathtt{root} = \mathtt{null} \wedge \mathtt{S} = \{\}) \\
&\quad \vee \exists\, \mathtt{p}, \mathtt{r}, \mathtt{S_r} \cdot (\mathtt{root} \mapsto \mathtt{node}\langle \mathtt{p@I}, \mathtt{@A}, \mathtt{r@M}, \mathtt{@A}\rangle * \mathtt{rl}\langle \mathtt{r}, \mathtt{S_r}\rangle \wedge \mathtt{S} = \mathtt{S_r} \cup \{\mathtt{root}\}) \\
&\mathtt{sl}\langle \mathtt{root}, \mathtt{S}\rangle \equiv (\mathtt{root} = \mathtt{null} \wedge \mathtt{S} = \{\}) \\
&\quad \vee \exists\, \mathtt{p}, \mathtt{s}, \mathtt{S_s} \cdot (\mathtt{root} \mapsto \mathtt{node}\langle \mathtt{p@I}, \mathtt{@A}, \mathtt{@A}, \mathtt{s@M}\rangle * \mathtt{sl}\langle \mathtt{s}, \mathtt{S_s}\rangle \wedge \mathtt{S} = \mathtt{S_s} \cup \{\mathtt{root}\})
\end{aligned}$$

These predicates specify the "all list", "running list", and "sleeping list", respectively. Each list predicate is parameterized by a set of addresses S of nodes on that list. Each points-to predicate ($\mathtt{node}\langle\cdot\rangle$) annotates the ownership of its fields: *e.g.*, the points-to in `al` has full ownership `@M` of the first (`anext`) pointer field. This claim is compatible with the `rl` and `sl` predicates since both of them are absent in that field. An interesting case is the process id field `pid`. All three of the predicates wish to share access to this field; we still consider them to be compatible as long as the field is marked immutable `@I`. 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 some interesting examples and we avoid some of the hassles of integrating fractional permissions into a verification tool [16].

Since we have two compatible predicates we want to use $*$ to combine them:

$$\mathtt{al}\langle \mathtt{root}, \mathtt{S_r} \cup \mathtt{S_l}\rangle * (\mathtt{rl}\langle \mathtt{root}, \mathtt{S_r}\rangle * \mathtt{sl}\langle \mathtt{root}, \mathtt{S_s}\rangle)$$

Actually this is not quite right. Although adding field-level separation is not new [8], the standard way to do so introduces a subtle ambiguity. The issue is that the amount of sharing of the objects is not fully specified: in fact the two $*$ are being used quite

differently. The first $*$, separating $\texttt{al}\langle\cdot\rangle$ from the other two, is actually more like a standard classical separation logic conjunction $\wedge$. That is, every node on the left hand side is also on the right hand side, and vice versa: the fields separate, but the nodes precisely overlay one another. In contrast, the second $*$, separating $\texttt{rl}\langle\cdot\rangle$ from $\texttt{sl}\langle\cdot\rangle$, is much more like the standard classical field-less separating conjunction $*$. That is, the set of nodes are strictly disjoint (no running process is sleeping, and vice versa), so even if $\texttt{rl}\langle\cdot\rangle$ and $\texttt{sl}\langle\cdot\rangle$ had incompatible fields, they would still be separate in memory.

This ambiguity means that the traditional field-level $*$ is a bit too easy to introduce, and unnecessarily painful to eliminate. We resolve it by using two distinct operators: $\mathbb{A}$, when we mean that nodes are identical and fields must be *compatible*; and $*$, which for us means that the nodes themselves are *disjoint*. Thus, we specify our threaded list as:

$$\texttt{al}\langle \texttt{x}, \texttt{S}_\texttt{y} \cup \texttt{S}_\texttt{z} \rangle \;\mathbb{A}\; (\texttt{rl}\langle \texttt{y}, \texttt{S}_\texttt{y} \rangle * \texttt{sl}\langle \texttt{z}, \texttt{S}_\texttt{z} \rangle)$$

We know that this predicate uses only compatible sharing because all of the fields on the lhs and rhs of the $\mathbb{A}$ are disjoint and the $*$ guarantees compatibility on the inner subformula. It may seem that $\mathbb{A}$ is very specific to this example, but that is wrong: in Sec. 6 we mention how we use it to reason about other overlaid data structures.

Contributions. We develop a specification mechanism to capture overlaid sharing, an entailment procedure to reason about such sharing, and integrate our ideas into an existing automated verification system. 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/`

2 Motivating Examples

In this section, we illustrate the difference between the various conjunction operators ($*$, $\wedge$ and $\mathbb{A}$) and give the intuition behind compatible sharing. We also show how to automatically check for compatible sharing in data structures using a notion of *memory specifications*.

2.1 From Separation to Sharing

As discussed earlier, 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[3] 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 $\mathtt{s}_1 = \mathtt{s}_2$).

To support field-level framing we use the field annotations introduced in section 1, to mark fields that are mutable (@M), immutable (@I) and absent (@A). 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

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

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 $*$.

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. Or, in other words it is possible to reason over only one side

[3] 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.

of conjunction and ignore the other since they can be combined together later without conflicts. 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:

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

2.2 Shared Process Scheduler

Recall that from section 1, the formula that we used to specify the threaded lists was as follows:

$$\mathtt{al}\langle \mathtt{x}, \mathtt{S}_\mathtt{y} \cup \mathtt{S}_\mathtt{z}\rangle \mathbb{A} (\mathtt{rl}\langle \mathtt{y}, \mathtt{S}_\mathtt{y}\rangle * \mathtt{sl}\langle \mathtt{z}, \mathtt{S}_\mathtt{z}\rangle)$$

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 using the **mem** construct which is associated with the predicate definition. The enhanced predicate definitions for the process scheduler are shown below.

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

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

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

The **mem** 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. (Predicate definitions also can be enchanced with other pure properties following translation technique described in section 7 of [19]). 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.

$$\begin{array}{l}
\textit{//Provided by User} \\
\mathtt{al}\langle \mathtt{x}\rangle \ \mathbb{A}\ (\mathtt{rl}\langle \mathtt{y}\rangle * \mathtt{sl}\langle \mathtt{z}\rangle) \\
\textit{//Predicate extension with mem} \\
\mathtt{al}\langle \mathtt{x}, \mathtt{S_x}\rangle \ \mathbb{A}\ (\mathtt{rl}\langle \mathtt{y}, \mathtt{S_y}\rangle * \mathtt{sl}\langle \mathtt{z}, \mathtt{S_z}\rangle) \\
\textit{//Translated form} \\
\mathtt{al}\langle \mathtt{x}, \mathtt{S_x}\rangle \wedge (\mathtt{rl}\langle \mathtt{y}, \mathtt{S_y}\rangle * \mathtt{sl}\langle \mathtt{z}, \mathtt{S_z}\rangle) \wedge \mathtt{S_x}{=}\mathtt{S_y} \cup \mathtt{S_z}
\end{array}$$

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 with each other.

```
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);
}
```

2.3 Comparison with Fractional Permissions

In this section, we show the difficulties that arise when using separation logic with fractional permissions (SLfp) to represent overlaid data structures. We avoid these issues by using field annotations and overlaid conjunction operator while specifying compatible sharing in data structures.

Applying fractional permissions (as in SLfp) to fields inside inductive predicates can unintentionally change the meaning of the predicate. E.g consider the following predicate definition of an immutable binary tree in SLfp :

$$\begin{aligned}&\mathtt{tree}\langle\mathtt{root}\rangle \equiv \mathtt{root}=\mathtt{null}\\ &\quad \vee \exists\, \mathtt{d}, \mathtt{l}, \mathtt{r} \cdot (\mathtt{root} \mapsto \mathtt{node}\langle \mathtt{d}@1/2, \mathtt{l}@1/2, \mathtt{r}@1/2\rangle * \mathtt{tree}\langle \mathtt{l}\rangle * \mathtt{tree}\langle \mathtt{r}\rangle)\end{aligned}$$

We restrict the use of fields in the predicate using the fraction 1/2 to give a read-only permission. However, this predicate does not enforce a tree and is in fact a DAG. In standard SLfp the $*$ operator does not enforce strict separation, thus the left and right children can point to the same node and combine using the 1/2 permissions given to each node. A more sophisticated permission system like tree-shares [16] can avoid this problem, but it is not known how to extend a tree-shares like model to fields.

We avoid this problem by using a definition of the $*$ operator that enforces strict object level separation. Also, we use field annotations that provide a simpler way to specify mutable, immutable and absent fields. If we use $*$ for object level separation and

$$
\begin{array}{lll}
\mathit{pred} & ::= p(v^*) \equiv \Phi \;\; [\mathit{inv}\;\; \pi][\mathit{mem}\;\; S \hookleftarrow ([c(@u^*)]^*)] & \\
\mathit{mspec} & ::= \Phi_{pr} \mathrel{*\!\!\rightarrow} \Phi_{po} \quad \Phi ::= \bigvee\; (\exists w^* \cdot \kappa \wedge \pi)^* & \\
\kappa & ::= \mathtt{emp} \mid v \mapsto c(v[@u]^*) \mid p(v^*) \mid \kappa_1 \sharp \kappa_2 \;\; (\sharp \in \{*, \wedge, \mathbb{A}\}) & \\
\pi & ::= \alpha \mid \pi \wedge \varphi \quad \alpha ::= \beta \mid \neg \beta & \\
\beta & ::= v_1 = v_2 \mid v = \mathtt{null} \mid a \leq 0 \mid a = 0 & \\
a & ::= k \mid k \times v \mid a_1 + a_2 \mid \mathit{max}(a_1, a_2) \mid \mathit{min}(a_1, a_2) & \\
\varphi & ::= v \in S \mid S_1 = S_2 \mid S_1 \subset S_2 \mid \forall v \in S \cdot \pi \mid \exists v \in S \cdot \pi & \\
S & ::= S_1 \cup S_2 \mid S_1 \cap S_2 \mid S_1 - S_2 \mid \{\} \mid \{v\} & \\
u & ::= M \mid I \mid A \quad (M <: I <: A) & \\
\multicolumn{3}{l}{\mathit{where}\;\; p \;\mathit{is\ a\ predicate};\; v, w \;\mathit{are\ variables};\; c \;\mathit{is\ a\ data\ type};\; u \;\mathit{is\ a\ field\ annotation};}
\end{array}
$$

Fig. 1. Specification Language

$\wedge$ for object level sharing then it is natural to introduce another operator $\mathbb{A}$ for object level sharing and field level separation. The overlaid conjunction ($\mathbb{A}$) is also practically useful to represent several data structures as shown in section 6.

3 Specification with Compatible Sharing

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 ($\kappa \wedge \pi$). We use the set constraints for representing memory regions as shown in Fig. 1. The predicate definition allows optional **mem** construct to be specified. **mem** 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.

$$
\frac{isData(c)}{XMem(c\langle p, v@u^*\rangle) =_{df} (\{p\}, [c\langle @u^*\rangle])} \qquad \frac{isPred(c) \quad c\langle p, S, v^*\rangle \equiv \Phi[\mathit{inv}\;\; \pi][\mathit{mem}\;\; \mathcal{S} \hookleftarrow L]}{XMem(c\langle p, S, v^*\rangle) =_{df} (S, L)}
$$

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

Fig. 2. XMem: Translating to Memory Form

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

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

Fig. 3. Validating the Memory Specification

In order to check compatible sharing between two predicates we take help of the $XMem(\kappa)$ function. The $XMem(\kappa)$ function, whose definition is given in Figure 2, returns a sound approximation of the memory footprint of heap κ as a tuple of the form $(S, [c(@u^*)]^*)$, which corresponds to the set of addresses and the list of field annotations used in memory specifications. The function $isData(c)$ returns `true` if c is a data node, while $isPred(c)$ returns true if c is a heap predicate. We use lists L_1 and L_2 to represent the field annotations. The function $union(L_1, L_2)$ returns the union of lists L_1 and 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. Since memory specifications are essential to check compatibility in data structures, we have machine checked the soundness proof of the $XMem$ function in Coq. We illustrate how the approximation function works on a linked list.

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

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

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

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

$$\begin{array}{l}
XMem(\texttt{x} \mapsto \texttt{node}\langle \texttt{d}, \texttt{p}\rangle) = (\{\texttt{x}\}, [\texttt{node}\langle @\texttt{M}, @\texttt{M}\rangle]) \\
XMem(\texttt{ll}\langle \texttt{y}, \texttt{S}_\texttt{y}\rangle) = (\texttt{S}_\texttt{y}, [\texttt{node}\langle @\texttt{M}, @\texttt{M}\rangle]) \\
XMem(\texttt{x} \mapsto \texttt{node}\langle \texttt{d}, \texttt{p}\rangle * \texttt{ll}\langle \texttt{y}, \texttt{S}_\texttt{y}\rangle) = (\{\texttt{x}\} \cup \texttt{S}_\texttt{y}, [\texttt{node}\langle @\texttt{M}, @\texttt{M}\rangle])
\end{array}$$

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

$$\frac{Compatible(\kappa_1 \mathbb{A} \kappa_2) \quad (S_1, L_1){=}XMem(\kappa_1) \quad (S_2, L_2){=}XMem(\kappa_2)}{\kappa_1 \mathbb{A} \kappa_2 \rightsquigarrow \kappa_1 \wedge \kappa_2 \wedge S_1 = S_2} \; [\textbf{ELIM-OVER-CONJ}]$$

$$x{\mapsto}c(v[@u]^*) \implies_{u<:w} x{\mapsto}c(v[@w]^*) \; [\textbf{DOWNCAST-FA}]$$

$$x{\mapsto}c(v[@u]^*) \iff x{\mapsto}c(v[@u]^*) \wedge x{\mapsto}c(v[@A]^*) \; [\textbf{SPLIT-COMBINE-FA}]$$

$$x{\mapsto}c(v[@I]^*) \iff x{\mapsto}c(v[@I]^*) \wedge x{\mapsto}c(v[@I]^*) \; [\textbf{SPLIT-READ-FA}]$$

Fig. 4. Rules with Field Annotations

existing underlying [6] entailment procedure (denoted by $\vdash$) 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 $subtype(L_1, L_2)$ function on lists of field annotations. The function returns true if all the field annotations of data nodes in L_1 have a corresponding node in L_2 and their field annotations are in the subtyping relation (as defined in Fig. 1).

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

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 $p(v^*) \equiv \Phi \;\; mem \;\; S{\hookrightarrow}L$, the judgment $\Phi \vdash_{mem} S{\hookrightarrow}L$ in Fig. 3 checks the validity of the memory specification.

Rule [CHECK-MEM] is used when the Φ formula does not contain a disjunction, while [CHECK-OR-MEM] is used when it does. The main difference in the disjunctive case is how we handle of list of field annotations. For the set of addresses (S) we 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.

4 Verification with Compatible Sharing

To verify programs with compatible sharing we make use of an existing entailment procedure for separation logic (denoted by $\vdash$ [5]). The only additional operator we have is the *overlaid* conjunction. We first describe the automatic translation used to eliminate $\mathbb{A}$ operator. For *overlaid* conjunction operator ($\mathbb{A}$), we must first identify the pairs of field annotations that are compatible. The following table can be used to look up compatible field annotations. The $\mathbb{A}$ operator is similar to $\wedge$, except that the shared

heaps must be compatible, which can be checked using the $Compatible$ function.

u_1	u_2	$Compatible_{FA}$
$@M$	$@M$	false
$@M$	$@I$	false
$@M$	$@A$	true
$@I$	$@I$	true
$@I$	$@A$	true
$@A$	$@A$	true

$$
\begin{array}{c}
Compatible(\kappa_1 \mathbb{A} \kappa_2) \quad =_{df} \\
(S_1, L_1){=}XMem(\kappa_1) \quad (S_2, L_2){=}XMem(\kappa_2) \\
\forall\, c(@u_1^*)\ in\ L_1,\ \exists\, c(@u_2^*)\ in\ L_2\ s.t.\ Compatible_{FA}(u_1, u_2) \\
\forall\, c(@u_2^*)\ in\ L_2,\ \exists\, c(@u_1^*)\ in\ L_1\ s.t.\ Compatible_{FA}(u_2, u_1)
\end{array}
$$

As shown in Fig. 4, the [ELIM–OVER–CONJ] rule first checks for compatible sharing of heaps and then uses the $XMem$ function to get the set of addresses S_1 and S_2 which are added to the formula when $\mathbb{A}$ operator is replaced with $\wedge$. Thus for the process scheduler example from Sec. 2 we get the following.

$$
\begin{array}{l}
\mathtt{al}\langle \mathtt{x}, \mathtt{S_x} \rangle \ \mathbb{A}\ (\mathtt{rl}\langle \mathtt{y}, \mathtt{S_y} \rangle * \mathtt{sl}\langle \mathtt{z}, \mathtt{S_z} \rangle) \quad \leadsto \\
\mathtt{al}\langle \mathtt{x}, \mathtt{S_x} \rangle \wedge (\mathtt{rl}\langle \mathtt{y}, \mathtt{S_y} \rangle * \mathtt{sl}\langle \mathtt{z}, \mathtt{S_z} \rangle) \wedge \mathtt{S_x} = \mathtt{S_y} \cup \mathtt{S_z}
\end{array}
$$

Fig. 4 also lists the rules required during entailment with field annotations. These rules are based on the definition of field annotations and the semantic model of the specification formula (details are in appendix 5.2). Rule [DOWNCAST–FA] says that we can always downcast a field annotation. This means that a *write* ($@M$) annotation can be downcast to *read* ($@I$) and a *read* annotation to *absent* ($@A$). The following examples use the [DOWNCAST–FA] rule to check validity of entailments with field annotations.

$$
\begin{array}{ll}
\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@M}, \mathtt{p@I}) \vdash \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@I}, \mathtt{p@A}) & (\mathit{Valid}) \\
\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@I}, \mathtt{p@I}) \vdash \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@I}, \mathtt{p@A}) & (\mathit{Valid}) \\
\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@I}, \mathtt{p@I}) \vdash \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@M}, \mathtt{p@A}) & (\mathit{Invalid})
\end{array}
$$

The *absent* annotation can always be split off (or combined with) any other annotation as shown in rule [SPLIT–COMBINE–FA]. Finally, as given in rule [SPLIT–READ–FA] the *read* annotation can be split into two *read* annotations. Together, these three set of rules allow exclusive write access and shared read access to fields. Entailments showing the use of [SPLIT–COMBINE–FA] rule are given below.

$$
\begin{array}{l}
\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@M}, \mathtt{p@I}) \vdash \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@I}, \mathtt{p@I}) \wedge \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@I}, \mathtt{p@A}) \\
\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@M}, \mathtt{p@M}) \vdash \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@M}, \mathtt{p@A}) \wedge \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@A}, \mathtt{p@M}) \\
\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@I}, \mathtt{p@I}) \vdash \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@I}, \mathtt{p@I}) \wedge \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v@I}, \mathtt{p@A})
\end{array}
$$

5 Semantics and Soundness

5.1 Storage Model

The storage model is similar to classical separation logic [18], 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.

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

Note that each heap h is a finite partial mapping while each stack s is a total mapping, as in the classical separation logic [18,13].

5.2 Semantic Model of the Specification Formula

The semantics of our separation heap formula is similar to the model given for separation logic [18], 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 Defn 1. The model relation for pure formula $s \models \pi$ denotes that the formula π evaluates to `true` in s.

Definition 1 (Model for Specification Formula)

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

The last case in Defn 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.

5.3 Soundness

The soundness of rules given in figure 4 can be established using the semantic model and the definition of field annotations. We now present the proof of soundness of these rules, we start first with the rules for field annotations.

Rule [**DOWNCAST−FA**]:

$$\begin{array}{ll}
& s,h \models x{\mapsto}c(v[@u]^*) \\
\iff & h{=}[s(x){\mapsto}r]{\wedge}r{=}c[f{\mapsto}_w s(v)]^*{\wedge}u{<:}w \quad (\textit{definition } 1) \\
\implies & h'{=}[s(x){\mapsto}r]{\wedge}r{=}c[f{\mapsto}_w s(v)]^*{\wedge}h'{\subset}h \quad (\textit{weakening}) \\
\iff & s,h' \models x{\mapsto}c(v[@w]^*){\wedge}h'{\subset}h \quad (\textit{definition } 1) \\
\iff & s,h \models x{\mapsto}c(v[@w]^*)
\end{array}$$

$$Thus, x{\mapsto}c(v[@u]^*) \Longrightarrow_{u<:w} x{\mapsto}c(v[@w]^*) \qquad \square$$

Rule [**SPLIT−COMBINE−FA**]:

$$\begin{array}{ll}
& s,h \models x{\mapsto}c(v[@u]^*) \\
\iff & h{=}[s(x){\mapsto}r]{\wedge}r{=}c[f{\mapsto}_w s(v)]^*{\wedge}u{<:}w \quad (\textit{definition } 1) \\
\iff & h'{=}[s(x){\mapsto}r]{\wedge}r{=}c[f{\mapsto}_{@A} s(v)]^*{\wedge}h'{\subset}h \ (\forall u \cdot u{<:}@A) \\
\iff & s,h' \models x{\mapsto}c(v[@A]^*){\wedge}h'{\subset}h \quad (\textit{definition } 1) \\
\iff & s,h' \models x{\mapsto}c(v[@A]^*){\wedge}h'{\subset}h \\
& \quad {\wedge}s,h \models x{\mapsto}c(v[@u]^*) \\
\iff & s,h \models x{\mapsto}c(v[@A]^*){\wedge}x{\mapsto}c(v[@u]^*) \quad (\textit{definition } 1)
\end{array}$$

$$\begin{array}{l}
Thus, x{\mapsto}c(v[@u]^*) \iff \\
\quad x{\mapsto}c(v[@u]^*) \wedge x{\mapsto}c(v[@A]^*) \qquad \square
\end{array}$$

Rule [**SPLIT−READ−FA**]:

$$\begin{array}{ll}
& s,h \models x{\mapsto}c(v[@I]^*) \\
\iff & h{=}[s(x){\mapsto}r]{\wedge}r{=}c[f{\mapsto}_w s(v)]^*{\wedge}I{<:}w \quad (\textit{definition } 1) \\
\iff & h'{=}[s(x){\mapsto}r]{\wedge}r{=}c[f{\mapsto}_{@I} s(v)]^*{\wedge}h'{\subset}h \quad (@I{<:}@I) \\
\iff & s,h' \models x{\mapsto}c(v[@I]^*){\wedge}h'{\subset}h \quad (\textit{definition } 1) \\
\iff & s,h' \models x{\mapsto}c(v[@I]^*){\wedge}h'{\subset}h \\
& \quad {\wedge}s,h \models x{\mapsto}c(v[@I]^*) \\
\iff & s,h \models x{\mapsto}c(v[@I]^*){\wedge}x{\mapsto}c(v[@I]^*) \quad (\textit{definition } 1)
\end{array}$$

$$\begin{array}{l}
Thus, x{\mapsto}c(v[@I]^*) \iff \\
\quad x{\mapsto}c(v[@I]^*) \wedge x{\mapsto}c(v[@I]^*) \qquad \square
\end{array}$$

Using the rules for field annotations we prove the soundness of the elimination rule as follows.

Rule [**ELIM−OVER−CONJ**]:

$$
\begin{array}{l}
s,h \models \kappa_1 \mathbb{\&} \kappa_2 \wedge (S_1, L_1){=}XMem(\kappa_1) \\
\quad \wedge (S_2, L_2){=}XMem(\kappa_2) \\
\Longleftrightarrow\ s,h \models \kappa_1 \wedge s, h \models \kappa_2 \wedge \\
\quad Compatible(\kappa_1 \mathbb{\&} \kappa_2) \wedge s \models S_1{=}S_2 (=h) \qquad (\textit{definition}\ 1)
\end{array}
$$

$$
\begin{array}{l}
case\ [\mathbf{SPLIT{-}COMBINE{-}FA}]: \\
\Longleftrightarrow\ h{=}[s(\_){\mapsto}r] \wedge r{=}c[f{\mapsto}_u s(\_)]^* \wedge \\
\quad h'{=}[s(\_){\mapsto}r] \wedge r{=}c[f{\mapsto}_{@A} s(\_)]^* \wedge h'{\subset}h \wedge \\
\quad Compatible(\kappa_1 \mathbb{\&} \kappa_2) \wedge s \models S_1{=}S_2 \\
\Longleftrightarrow\ s,h \models \kappa_1 \wedge s, h' \models \kappa_2 \wedge h'{\subset}h \\
\quad \wedge s \models S_1{=}S_2 \qquad (Compatible_{FA}) \\
\Longrightarrow\ s,h \models \kappa_1 \wedge \kappa_2 \wedge S_1{=}S_2 \qquad (\textit{definition}\ 1)
\end{array}
$$

$$
\begin{array}{l}
case\ [\mathbf{SPLIT{-}READ{-}FA}]: \\
\Longleftrightarrow\ h{=}[s(\_){\mapsto}r] \wedge r{=}c[f{\mapsto}_{@I} s(\_)]^* \wedge \\
\quad h'{=}[s(\_){\mapsto}r] \wedge r{=}c[f{\mapsto}_{@I} s(\_)]^* \wedge h'{\subset}h \wedge \\
\quad Compatible(\kappa_1 \mathbb{\&} \kappa_2) \wedge s \models S_1{=}S_2 \\
\Longleftrightarrow\ s,h \models \kappa_1 \wedge s, h' \models \kappa_2 \wedge h'{\subset}h \\
\quad \wedge s \models S_1{=}S_2 \qquad (Compatible_{FA}) \\
\Longrightarrow\ s,h \models \kappa_1 \wedge \kappa_2 \wedge S_1{=}S_2 \qquad (\textit{definition}\ 1)
\end{array}
$$

$$
Thus, \kappa_1 \mathbb{\&} \kappa_2 \rightsquigarrow \kappa_1 \wedge \kappa_2 \wedge S_1{=}S_2 \qquad \square
$$

There are two ways of splitting the *overlaid* heaps - in the first case we use the [**SPLIT−COMBINE−FA**] to combine them back as the fact that they are in compatible sharing means that the field annotations can only be from the pairs given in table for $Compatible_{FA}$ in Sec 5.2 and we prove the second case similarly using the [**SPLIT−READ−FA**] rule. Soundness of the underlying entailment procedure (as shown in [5]) and the soundness of the rules given in Fig. 4 together establish the soundness of verification with compatible sharing.

6 Experiments

We have built a prototype system using Objective Caml called `HIPComp`.[4] 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 [14] and Mona [15]). 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 inherent sharing and aliasing in heap. For each of these examples, we verify methods that insert, find and remove nodes from the overlaid data structure.

[4] `http://loris-7.ddns.comp.nus.edu.sg/~project/HIPComp/`

Program	*Invariant*	*LOC*	*Time [secs]*	*Sharing*	*Comp*
Parameterized List	$\texttt{al}\langle\texttt{x}\rangle$ 𝔸 $(\texttt{rl}\langle\texttt{y}\rangle * \texttt{sl}\langle\texttt{z}\rangle)$	30	0.28	100	40
Compatible Pairs	$\texttt{x}{\mapsto}\texttt{pair}\langle\texttt{f}_1, @\texttt{A}\rangle$ 𝔸 $\texttt{y}{\mapsto}\texttt{pair}\langle @\texttt{A}, \texttt{s}_2\rangle$	12	0.09	100	25
LL and SortedLL	$\texttt{ll}\langle\texttt{x}\rangle$ 𝔸 $\texttt{sll}\langle\texttt{y}\rangle$	175	0.61	22	22
LL and Tree	$\texttt{ll}\langle\texttt{x}\rangle$ 𝔸 $\texttt{tree}\langle\texttt{t}\rangle$	70	0.24	16	7
Doubly Circular List	$\texttt{llnext}\langle\texttt{x}\rangle$ 𝔸 $\texttt{lldown}\langle\texttt{y}\rangle$	50	0.41	50	32
Process Scheduler	$\texttt{al}\langle\texttt{x}\rangle$ 𝔸 $(\texttt{rl}\langle\texttt{y}\rangle * \texttt{sl}\langle\texttt{z}\rangle)$	70	0.47	33	23
Disk IO Scheduler	$(\texttt{ll}\langle\texttt{x}\rangle$ 𝔸 $\texttt{tree}\langle\texttt{t}\rangle) * \texttt{ll}\langle\texttt{y}\rangle$	88	1.3	16	27

The above table summarises a suite of small 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. The second column shows how we use the overlaid conjunction 𝔸 to concisely specify the overlaid data structures in our experiments. As shown in the table, for the last two programs, the key invariant of the overlaid data structure can also be a composite structure which intermixes $*$ and 𝔸 operators. It is essential to reason about compatible sharing when specifying and verifying such programs. The third column lists the lines of code (including specifications) in the program. The annotation burden due to specifications is about 30% of the total number of lines of code. In the fifth column, we show the sharing degree, it is defined as the percentage of specifications that use compatible sharing using field annotations. The sharing degree varies across examples depending on the percentage of methods that use overlaid conjunction in their specifications.

As is clear from our benchmark programs, the ability to specify sharing is important to verify these data structures. The last column (*Comp*) is the percentage of total entailments generated that make use of compatible sharing. The compatibility percentage depends on the number of entailments that make use of the [ELIM–OVER–CONJ] rule to eliminate the overlaid conjunction. The compatibility check is essential to verify sharing in these programs.

7 Related Work and Conclusions

Our sharing and aliasing logic is most closely related to Hobor and Villard [12]; our work verifies only a subset of what they can do but we do so mechanically/automatically. The problem of sharing has also been explored in the context of concurrent data structures [7,20]. 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 set of addresses as footprint of formulas. These regions are used with dynamic frames to enable local reasoning of programs. Memory layouts [10] were used by Gast, as a way to formally specify the structure of individual memory blocks. A grammar of memory layouts enable distinguishing between variable, array, or other data structures. This shows that when dealing with shared regions of memory knowing the layout of memory can be quite helpful for reasoning. We use field annotations to specify access to memory in shared and overlaid data structures.

In the area of program analysis [8,4] the work most closely related to ours is by Lee et al. [17] 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. Thus, we present an automated framework which can be used to reason about compatible sharing in data structures. An initial set of experiments with small but challenging programs confirms the usefulness of our method. Similarly, the recent work of Dragoi et al. [8] considers only the shape analysis of overlaid lists. In addition, Enea et al. [9] consider the compositional invariant checking of overlaid and nested 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. A separation logic based program analysis has been used to handle non-linear data structures like trees and graphs [4]. In order to handle cycles they keep track of the nodes which are already visited using multi-sets.

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 framework 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. For future work, we want to explore the use of memory regions and field annotations to enable automated verification of other intrinsic shared data structures that do not satisfy compatible sharing (like dags and graphs).

Acknowlegement. This work is supported by MoE 2013-T2-2-146 and Yale-NUS College R-607-265-045-121.

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. Renato Cherini, Lucas Rearte, and Javier O. Blanco. A shape analysis for non-linear data structures. In *SAS*, pages 201–217, 2010.
5. 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.
6. Cristina David and Wei-Ngan Chin. Immutable specifications for more concise and precise verification. In *OOPSLA*, pages 359–374, 2011.
7. Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.

8. Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Local shape analysis for overlaid data structures. In *SAS*, pages 150–171, 2013.
9. Constantin Enea, Vlad Saveluc, and Mihaela Sighireanu. Compositional invariant checking for overlaid and nested linked lists. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013*, pages 129–148, 2013.
10. Holger Gast. Reasoning about memory layouts. In *FM*, pages 628–643, 2009.
11. 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.
12. Aquinas Hobor and Jules Villard. The Ramifications of Sharing in Data Structures. In *POPL*, 2013.
13. 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.
14. P. Kelly, V. Maslov, W. Pugh, and et al. *The Omega Library Version 1.1.0 Interface Guide*, November 1996.
15. N. Klarlund and A. Moller. MONA Version 1.4 - User Manual. BRICS Notes Series, January 2001.
16. Xuan Bach Le, Cristian Gherghina, and Aquinas Hobor. Decision procedures over sophisticated fractional permissions. In *APLAS*, pages 368–385, 2012.
17. Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program analysis for overlaid data structures. In *CAV*, pages 592–608, 2011.
18. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, pages 55–74, 2002.
19. Minh-Thai Trinh, Quang Loc Le, Cristina David, and Wei-Ngan Chin. Bi-abduction with pure properties for specification inference. In *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, pages 107–123, 2013.
20. Aaron Joseph Turon and Mitchell Wand. A separation logic for refining concurrent objects. In *POPL*, pages 247–258, 2011.