Separation Logic with First-Class Heaps and a New Frame Rule

Duc-Hiep Chu  
National University of Singapore  
hiepcd@comp.nus.edu.sg

Joxan Jaffar  
National University of Singapore  
joxan@comp.nus.edu.sg

Abstract
Separation Logic brought an advance to program verification of data structures through its use of (recursively defined) predicates to implicitly represent heaps, and the separation operator to construct heaps from disjoint subheaps. While this facilitated local reasoning in program fragments, the consideration of subheaps that are disjoint meant that any form of sharing between predicates is problematic and often requires manual proofs whose complexity may outweigh the core benefits of Separation Logic itself. With this as background motivation, we present an assertion language in which subheaps may be explicitly defined within predicates, and the effect of separation obtained by specifying that certain heaps are disjoint. Predicates can then be conjoined in the traditional way. We then present a new frame rule that is conditioned upon the heap-update operations of a program fragment. Essentially, a predicate can be framed over the program fragment if its footprint is disjoint from the updates. The main contribution is to demonstrate that the induced program verification method now provides local reasoning over heaps, such as list, and the separation operator to construct implicitly subheaps may be overloaded to specify values for x and y, as well as to specify, by virtue of the separating conjunction “∗”, that the heaps in which these two formulas are accomodated are separate from one another. Formally, (x → y)∗ list(y), one heap whose first component is the one-element heap at address x with content y, and whose second component is the heap accomodating the list y.

There are several Hoare triples which are axioms, and several inference rules for deriving new triples. The key advantage of Separation Logic, however, is in its provision for local reasoning. This is manifested in the frame rule:

\[ \{ \psi \} \ P \{ \phi \} \]
\[ \{ \psi \ast \Pi \} \ P \{ \phi \ast \Pi \} \]  

(1.1)

where the premise \( \{ \psi \ast \Pi \} \ P \{ \phi \ast \Pi \} \) ensures that the implicit heap arising from the formula \( \psi \) captures all the heap accesses, read or write, in the program fragment \( P \). This means that in any triple, there is an obligation to show that the program fragment is memory safe wrt. its precondition \( \psi \), or that it is contained with the footprint of (the heap represented by) \( \psi \). The rule (1.1) now is easy to understand: we can extend the proof in its premise to that in the conclusion because the new condition \( \Pi \) can be “framed” by virtue of it being a separate heap from \( \psi \).

We now paraphrase [4] to describe one the biggest advantages of the frame rule, as well as one of its biggest limitations: “The frame rule buys us compositionality in the presence of the heap: we can reason about the effect a program has on the portions of heap it accesses, and reuse that spec in any bigger heap. This has given rise to concise, compositional proofs of programs, even in the presence of some forms of sharing where one knows what is shared by whom. Unfortunately, we usually cannot use the frame rule directly when verifying programs that manipulate data structures with unrestricted sharing because such structures cannot easily be massaged into the form \( \psi \ast \Pi \): for example, the left and right descendants of a dag node are not usually disjoint.”

With this as background motivation, we will present an assertion language in which subheaps may be explicitly defined within predicates, and the effect of separation obtained by specifying that certain heaps are disjoint. In other words, heaps are first-class in this language. Predicates can then be conjoined in the traditional way, and not restricted to separating conjunction. From a usability point of view, this actually removes a very significant restriction.

1. Introduction
An important part of reasoning over heap manipulating programs is the ability to specify properties local to separate (i.e. non-overlapping) regions of memory. Separation Logic [8] brought a significant advance to program verification of data structures through its use of (recursively defined) predicates to implicitly represent heaps, such as list and tree above. It then explicates separation between regions of memory through the separating conjunction construct ∗. For example, the formula

\[ \text{list(l) } \ast \text{ tree(t)} \]

represents a program heap comprised of two separate sub-heaps: one containing a linked-list and the other a tree data-structure. A classic example recursive definition is that of an acyclic skeletal list, and in this paper, we will use the following syntax to describe their rules:

\[ \text{list}(x) :\text{=} x = \text{null} \]
\[ \text{list}(x) :\text{=} (x \rightarrow y) \ast \text{list}(y) . \]

Clearly the first rule deals with the (base) case of a null list (and implicitly associates it to the empty heap) while the second rule deals with the remaining case of a nonempty list. The key idea is in the body of the second rule, where the two formulas, \((x \rightarrow y)\) and \(\text{list}(y)\) are overloaded to specify values for x and y, as well as to specify, by virtue of the separating conjunction “∗”, that the heaps in which these two formulas are accomodated are separate from one another. Formally, \((x \rightarrow y)\ast \text{list}(y)\), one heap whose first component is the one-element heap at address x with content y, and whose second component is the heap accomodating the list y.

There are several Hoare triples which are axioms, and several inference rules for deriving new triples. The key advantage of Separation Logic, however, is in its provision for local reasoning. This is manifested in the frame rule:

\[ \{ \psi \} \ P \{ \phi \} \]
\[ \{ \psi \ast \Pi \} \ P \{ \phi \ast \Pi \} \]  

(1.1)

where the premise \( \{ \psi \ast \Pi \} \ P \{ \phi \ast \Pi \} \) ensures that the implicit heap arising from the formula \( \psi \) captures all the heap accesses, read or write, in the program fragment \( P \). This means that in any triple, there is an obligation to show that the program fragment is memory safe wrt. its precondition \( \psi \), or that it is contained with the footprint of (the heap represented by) \( \psi \). The rule (1.1) now is easy to understand: we can extend the proof in its premise to that in the conclusion because the new condition \( \Pi \) can be “framed” by virtue of it being a separate heap from \( \psi \).

We now paraphrase [4] to describe one the biggest advantages of the frame rule, as well as one of its biggest limitations: “The frame rule buys us compositionality in the presence of the heap: we can reason about the effect a program has on the portions of heap it accesses, and reuse that spec in any bigger heap. This has given rise to concise, compositional proofs of programs, even in the presence of some forms of sharing where one knows what is shared by whom. Unfortunately, we usually cannot use the frame rule directly when verifying programs that manipulate data structures with unrestricted sharing because such structures cannot easily be massaged into the form \( \psi \ast \Pi \): for example, the left and right descendants of a dag node are not usually disjoint.”

With this as background motivation, we will present an assertion language in which subheaps may be explicitly defined within predicates, and the effect of separation obtained by specifying that certain heaps are disjoint. In other words, heaps are first-class in this language. Predicates can then be conjoined in the traditional way, and not restricted to separating conjunction. From a usability point of view, this actually removes a very significant restriction.
We begin by adopting the constraint language of [3] which first introduced explicit heaps and which realized the separation concept simply as a disjointness condition between (the domains of) heaps. We then present a methodology of recursive rules which use the heap constraint language as its primary expressions. We show how capture complex complex properties about both sharing and separation. The predicates arising from our recursive definitions can then be used in program verification by simply connecting the variables in the definitions to (actual) program variables.

We then present the verification framework in two main parts. First, we deal with the part of a heap that is possibly changed by a program fragment. This is easily handled by a strongest postcondition transform, so that the proof of any triple \( \{ \psi \} P \{ \phi \} \) will just require the proof of \( \phi \) given the strongest postcondition of \( P \) from \( \psi \). This result easily follows from the previous work [3].

The second part of our verification framework is a new frame rule. Now recall that in Separation Logic the frame rule, eg. (1.1) above, requires, in proving a triple such as \( \{ \psi \} P \{ \phi \} \), showing that no memory errors can happen when \( P \) is executed in any state satisfying \( \psi \). This, in turn, requires showing that both the read and write operations lie in the heap of \( \psi \).

In contrast, our frame rule only considers write operations. We first define the notion of a footprint of a predicate, which is a set of heap locations. Informally, this says the following: if an instance of the predicate is \textit{true} (on some valuation of its expressions), then any change to the global heap outside these locations, then the valuation on the predicate remains \textit{true}. (Note that in general, the expressions in the predicate are connected to the global heap.) The next step is easy: a predicate can be framed over the program fragment if its footprint is disjoint from the heap updates in the program fragment.

In the end, the main contribution of this paper is to demonstrate that the induced program verification method now provides local reasoning on certain use cases that are important in practice, and for which previous verification frameworks have so far not been adequate. These use cases are to accomodate:

- **structure sharing** in data structures, such as in cyclic graphs, where different program fragments act on different parts on these data structures;
- **summaries** of program fragments where there is a recursively defined relationship between the global heap at the entry and exit program points of the fragment; and
- **incremental updates** to complex data structures. Typically different parts of a complex data structure is manipulated by different program fragments (and this depends on the stored values).

2. Preliminaries

We use the \( \mathcal{H} \) constraint language for reasoning over program heaps first described in [3]. Here we give a brief overview.

The \( \mathcal{H} \)-language is related to Separation Logic. Here we assume as given a set of Values (typically Values \( \equiv \mathbb{Z} \)) and define the set of Heaps to be all finite partial maps between values, i.e. Heaps \( \equiv (\text{Values} \to \text{Values}) \). We shall distinguish one value called to represent a null pointer. Likewise we assume as given sets of variables \( \text{Var}_v \) and \( \text{Var}_h \) that are disjoint from each other. Our Separation Logic-like formulas \( F \) are as follows:

\[
H ::= \text{Var}_h \\
v ::= \text{Var}_v \\
F ::= H \mid \text{emp} \mid (v \mapsto v) \mid F + F
\]

A valuation \( \varphi \) maps \( \text{Var}_h \) to Heaps and \( \text{Var}_v \) to Values. Syntactically, a heap constraint is a literal of the form \( (F \models F) \). A valuation \( \varphi \) satisfies a heap constraint \( (F_1 \models F_2) \) iff \( (1) \varphi(F_1) \) and \( \varphi(F_2) \) hold as ground Separation Logic formulae, and \( (2) \varphi(F_1) = \varphi(F_2) \) are the same heap.

Let \( \text{dom}(H) \) be the domain of the heap \( H \). We sometimes abuse notation and treat heaps \( H \) the set of pointer-value pairs \( \{(p,v) \mid p \in \text{dom}(H)\} \). As shown in [3], heap constraints can be normalized into three basic forms:

\[
H \simeq \text{emp}(\text{Empty}) \\
H \simeq (p \mapsto v)(\text{Singleton}) \\
H \simeq H_1 + H_2 (\text{Separation})
\]

where \( H, H_1, H_2 \in \text{Var}_h \) and \( p, v \in \text{Var}_v \). Here (Empty) constrains \( H \) to be the empty heap (i.e. \( H = \emptyset \) as sets), (Singleton) constrains \( H \) to be the singleton heap mapping \( p \mapsto v \) (i.e. \( H = \{(p,v)\} \) as sets), and (Separation) constrains \( H \) to be heap that is partitioned into two disjoint sub-heaps \( H_1 \) and \( H_2 \) (i.e. \( H = H_1 \cup H_2 \) as sets and \( \text{dom}(H_1) \cap \text{dom}(H_2) = \emptyset \)). A sound and complete constraint solver for normalized \( \mathcal{H} \)-constraints is presented in [3].

In addition to the basic heap constraints, we assume definitions for domain membership \( (p \in \text{dom}(H)) \), heap membership \( (H(p,v)) \), i.e. \( (p,v) \in H \) as sets), a sub-heap relation \( (H_1 \cont H_2) \); as defined in [3].

Program Analysis via Symbolic Execution

Symbolic execution involves executing a program using symbolic values as inputs. The output of symbolic execution is a formula representing a given path through the program.

By convention we use a distinguished heap variable \( \mathcal{H} \) to represent the current program heap at any given program point. We can therefore describe symbolic execution of programs as Hoare triples over the \( \mathcal{H} \)-language as follows:

\[
\{ \phi \} x := y \{ \text{access}(\phi, x, y) \} \quad \text{(Heap access)} \\
\{ \phi \} x := y \{ \text{assign}(\phi, x, y) \} \quad \text{(Heap assignment)} \\
\{ \phi \} x := \text{malloc}(1) \{ \text{alloc}(\phi, x) \} \quad \text{(Heap allocation)}
\]

where auxiliary macros access, assign and alloc expand as follows:

\[
\text{access}(\phi, x, y) \equiv \exists H, x_0 : \mathcal{H} \models (y \mapsto x) \ast H \land \phi[x_0/x] \\
\text{assign}(\phi, x, y) \equiv \exists H_0, H, v : H_0 \models (x \mapsto v) \ast H \land \mathcal{H} \models (x \mapsto y) \ast H \land \phi[H_0/\mathcal{H}] \\
\text{alloc}(\phi, x) \equiv \exists H_0, v, x_0 : \mathcal{H} \models (x \mapsto v) \ast H_0 \land \phi[H_0/\mathcal{H}, x_0/x]
\]

Here the notation \( \varphi[x/y] \) means formula \( \varphi \) with variable \( x \) substituted for \( y \). We generalize the rules for \text{malloc}(n), constant \( n > 1 \) and for \( x[i] \) in the obvious way. The above triples correspond to the Strongest Post-Condition Encoding from [3].

3. The Assertion Language

We will be assuming a vanilla imperative programming language, and so we omit a formal definition. Only value variables appear in a program, and no heap variables. Heap references are denoted using the “\( \ast \)” notation (as in C). There should be no confusion with the separation operator in SL. In assertions, however, heap variables may be used. Of special distinction is the global heap variable \( \mathcal{H} \). A number of other heap variables may also be used; these are called ghost heap variables.

\footnote{We shall not be concerned with memory deallocation in this paper.}
An assertion $A$ is a formula

$$ A ::= VC \mid HC \mid RC \mid A \land A $$

where $VC$ is a value constraint constructed from value variables and value expressions. Because we will not be concerned with the specific nature of value constraints, we will not define this further. In our examples, they will be simple integer, set or list (of integer) expressions. The symbol $HC$ denotes a heap constraint as defined above. Finally, $RC$ is a recursive constraint. This is expression of the form

$$ p(h_1,\ldots, h_n, v_1,\ldots, v_m) $$

(3.1)

where $p$ is a (user-defined) predicate symbol, the $h_i, 0 \leq i \leq n$ are heap variables from $Vars$, and the $v_i, 0 \leq i \leq m$, are value variables from $Values$. We next describe how to associate a recursive definition with such predicate symbols.

We use the framework of Constraint Logic Programming (CLP) \cite{5} to inherit its syntax, semantics, and most importantly, its built-in notions of unfolding rules. For the sake of brevity, we will only informally explain the language. The following rules constitute a recursive definition of predicate $\text{list}(h, x)$ which specifies a skeleton list in the heap $h$ rooted at $x$.

$$ \text{list}(\text{emp}, \text{null}). $$

$$ \text{list}(\langle x \mapsto y \rangle \ast h, x) :- \text{list}(h, y). $$

The semantics of a set of rules is traditionally known as the “least model” semantics \cite{5}. Essentially, this is the set of valuations of the expressions in the predicate that results in a true value when instantiated into the recursive rules. Thus for example, the semantics of the predicate $\text{list}(H, X)$ is simply the set of heaps $H$ which house an acyclic list rooted at $X$ and which is null terminated.

Now in the above example, the predicate $\text{list}(H, X)$ defined just one heap. In Separation Logic, writing a predicate in the context of a separating conjunction; for example, $\text{list}(X_1) \ast \text{list}(X_2)$, where list is defined in Separation Logic as:

$$ \text{list}(\text{null}). $$

$$ \text{list}(x) :- \langle x \mapsto y \rangle \ast \text{list}(h, y). $$

would implicitly refer to heaps accommodating two lists rooted at $X_1$ and $X_2$, and be able to state their separation. Thus what is the advantage formulating $\text{list}(H, X)$ with the explicit heap $H$?

We now show how, with the use of explicit heaps we can define, amongst many other complex properties, cyclic structures. The following predicate $\text{cycle}(H, X)$ states that $H$ houses a cyclic list rooted and terminated at $X$. Note that this predicate is defined in terms of another predicate $\text{lsseg}(H, H_1, X)$ which states $H_1$ houses a (acyclic) list segment rooted at $X$ and whose final node points into $H$.

$$ \text{cycle}(h, x) :- \text{lsseg}(\langle h \mapsto \_ \rangle, x). $$

$$ \text{lsseg}(h_1, x) :- h \sqsupseteq \langle Y \mapsto \_ \rangle, h_1 \triangleright (x \mapsto y). $$

$$ \text{lsseg}(h, \langle x \mapsto y \rangle \ast h_1, x) :- $$

$$ h \triangleright \langle y \mapsto \_ \rangle, $$

$$ \text{lsseg}(h, h_1, y). $$

We show that, for example, that $\text{cycle}(\langle x \mapsto x \rangle, x)$, representing a one-node cyclic list rooted at $x$, is true. From the first rule,
The following is quite immediate: if a recursive constraint \( \text{RC} \) appears in a canonical assertion \( A \), then the set of all the heap variables \( V \) in \( \text{RC} \) form a footprint of \( \text{RC} \) in the following sense: if \( A \) holds before executing a program fragment which only updates locations outside the domains of \( V \), then \( \text{RC} \) holds after execution.

But there is a good reason why we would want a smaller set than all the heap variables in \( \text{RC} \) to be declared in the footprint. For example, recall the inclist\((h_1, h_2, x)\) predicate above which was intended to state that the in the global heap, a piece of which is represented by \( h_1 \), the list rooted at \( x \) is an “increment” of the corresponding list in the ghost heap, represented by \( h_2 \). In this example, the footprint of the predicate can in fact be more accurately described as the heap \( h_1 \); that is, the heap \( h_2 \) can be excluded. So while we desire to declare that \( h_1 \) (alone) is a footprint for inclist\((h_1, h_2, x)\), what ensures correctness?

To see the technical problem, consider the predicate \( p(h_1, h_2) \) whose definition is:

\[
p(h_1, h_2) \iff h_1 \equiv (10 \rightarrow y) \land h_2 \equiv (20 \rightarrow y).
\]

and which resides in an assertion which states that \( \mathcal{H} \supseteq h_1 \). It is tempting to declare that \( h_2 \) is not in the footprint for the predicate in \( A \), because \( h_2 \) is not “connected” to the global heap \( \mathcal{H} \) in \( A \). But of course this is incorrect because there is in fact a connection via the value expression \( y \). Therefore, \( h_2 \) cannot be excluded from the footprint. This illustration shows that, in general, it is not easy to declare that a particular heap variable can be excluded from the footprint. But, returning to the inclist\((h_1, h_2, x)\) example, what can we do to exclude \( h_2 \), which we all know intuitively is possible? In this particular example, the answer is that \( h_2 \) is referring to the same locations as \( h_1 \), and so it is redundant to include \( h_2 \) into the footprint variable set. So, in summary, the problem is, how to exclude some heap variables appearing in \( \text{RC} \) to be in its footprint?

We do not present a general solution to this problem in this paper, but instead delegate the correctness of the exclusion to the user. This is reasonable at this stage because most common uses of heap variables that are not connected to the global heap, notably ghost variables, can be easily shown to immune from program statements on the global heap. While it is possible to define restrictions on assertions to ensure such variable exclusions, our position in this paper is that common uses have such simple proofs that this endeavour is of little value. The main reason for this is that, most often, domain of the heap variable \( h \) to be excluded is same as that of the other heap variables which describe a part of the global heap.

In this situation, excluding \( h \) (due to redundancy) is trivial. This is the case in all the examples in this paper.

We finish off with

**Definition 2 (Footprint).** A footprint of a recursive constraint \( \text{RC} \) in a canonical assertion \( A \), denoted \( \text{footprint}_A(\text{RC}) \), is a subset of the heap variables \( V \) in \( \text{RC} \) such that if \( A \) holds before executing a program fragment which only updates locations outside the domains of \( V \), then \( \text{RC} \) holds after execution.

In summary for this section, we have presented assertions and proposed that the restricted form of canonical assertions are expressive enough for most uses. We will exemplify this in the assertions used in Section 5 concerning our main examples. Then our main technical contribution is the definition of the concept of footprint, and its implementation in practice by simply identifying a subset of the heap variables appearing in recursive constraints. We next take the footprint concept to a frame rule to finally produce a compositional proof method.

**Theorem 1 (Foundation for Framing).**

\[
x \notin \text{dom}(\text{footprint}_A(\text{RC})) \implies \{ A \} \; \text{P} \{ B \} \quad \text{for} \quad \{ A \} \; \text{P} \{ B \} \; \text{in} \; \text{RC}
\]

The final step is of course the corresponding rule to the frame rule in Separation Logic: that if we have shown that a program fragment \( P \) is update enclosed in some heap \( h \) in an assertion \( A \), then any predicate \( \text{RC} \) in \( A \) whose footprint is separate from \( h \) frames through \( P \). We will not go through the formalism to define this notion of update enclosure, except to stress that it is a much weaker preconditions than what is needed in Separation Logic. That is, in Separation Logic, a precondition is a triple \( \{ \psi \} \; \text{P} \{ \phi \} \) before we can frame and conclude \( \{ \psi \land \Pi \} \; \text{P} \{ \phi \land \Pi \} \). This precondition requires that all heap references, not just updates, are enclosed in the (full) “footprint” of \( \psi \). In contrast, we require just that \( P \) updates a subset of the heaps mentioned in \( \psi \).

Let us denote this concept using the notation

\[
\text{update}_A(P)
\]

which identifies the subset of heap variables in \( A \) which enclose heap updates in \( P \). Then we can have:

**Theorem 2 (Framing).**

\[
x \text{ updated in } P \implies x \notin \text{dom}(\text{update}_A(P)) \implies \{ A \} \; \text{P} \{ A \}
\]

Next, we will put all this together to solve some important use cases that have eluded Separation Logic up till now.
5. Driving Examples

We begin by exemplifying the use of our assertion language on a classic example in Separation Logic. We start with a definition of binary tree whose only data value is a mark. Below we define the predicate \( \text{tree}(h,x) \) where \( H \) is a heap containing a tree of nodes rooted at \( x \). Each node is a 3-element heap comprising a mark field, and two other fields for the left and right successor nodes of \( x \). We also define a similar predicate \( \text{markedtree}(h,x) \) which is similar to \( \text{tree}(h,x) \) except that in this case, all the mark fields in the tree have the value 1.

\[
\begin{align*}
  \text{tree}(\text{emp}, \text{null}), \\
  \text{tree}(x \mapsto (\_, \text{left}, \text{right}) * h_1 * h_2, x) :- \\
  \text{tree}(h_1, \text{left}), \\
  \text{tree}(h_2, \text{right}).
\end{align*}
\]

\[
\begin{align*}
  \text{markedtree}(\text{emp}, \text{null}), \\
  \text{markedtree}(x \mapsto (1, \text{left}, \text{right}) * h_1 * h_2, x) :- \\
  \text{markedtree}(h_1, \text{left}), \\
  \text{markedtree}(h_2, \text{right}).
\end{align*}
\]

Now consider a classic algorithm for marking trees. We use the C program below for this:

```c
struct node {
  int m; // struct node *left, *right;
};

void mark(struct node *x) {
  if (!x) return;
  x->m = 1;
  mark(x->left);
  mark(x->right);
}
```

The proof of the Hoare triple may be outlined as follows. In this description, we shall limit our attention to (a) obtaining an assertion as the postcondition of the first recursive call, and (b) framing this assertion through the second call.

\[
\{ \text{tree}(H, x) \} \text{mark}(x) \{ \text{markedtree}(H, x) \}
\]

may be outlined as follows.

- Assume \( \text{tree}(h, x) \) holds for some \( h \), and that \( x \) is not null. Initially \( h \) is \( H \). When we propagate this predicate into the program fragment to just before the first recursive call, we have that \( \text{tree}(h, x) \) still holds. Unfolding the predicate using second rule in the definition of \( \text{tree} \), we obtain:

\[
h = x \mapsto (1, \text{left}, \text{right}) * h_1 * h_2, \text{tree}(h_1, \text{left}), \text{tree}(h_2, \text{right}).
\]

holds. Note in particular that \( h_1 \) holds.

- After the first recursive call, it is established, by induction, that since \( \text{tree}(h_1, \text{left}) \) holds before the call, \( \text{markedtree}(h_1, \text{left}) \) holds after the call.

- Similarly, the second recursive call establishes that \( \text{markedtree}(h_2, \text{right}) \) holds after the call.

- Now in order to prove that \( \text{markedtree}(h_1, \text{left}) \) continues to hold after the second recursive call, we would use a frame rule.

- In our setting, the (classic) frame rule applies because in the second call, all heap updates are limited to (the domain of) \( h_2 \) which is not in the footprint of \( \text{markedtree}(h_1, \text{left}) \). This is the critical step we would like to highlight, for further reference below.

- Finally, that the root node \( x \) is marked (in \( h \)) and that the two subtrees of \( x \) are marked will show, using an appropriate theorem-prover, that the entire tree rooted at \( x \) is marked. (We will comment on the theorem-proving of our assertions later.)

Moving forward into new territory, we now exemplify the use of our assertion language to specify not just structure sharing in data structures, but to specify cyclic data structures. In a following subsection, we present three driving examples involving structure sharing, summaries, and value-based separation to demonstrate that our framework crosses a fundamental boundary.

Consider a graph, and for simplicity, assume that for each of its nodes, there are at most two successor nodes labelled “left” and “right”. Consider a predicate \( \text{graph}(h, h_1, x, t) \) intended to define that

- the separated \( h_1 \) and \( h_2 \) heaps together house a graph rooted at \( x \) and where the terminal nodes are in the set of values \( t \).

- \( t \) represents a set of locations, disjoint from \( \text{dom}(h_1) \cap \text{dom}(h_2) \), which can be pointed to by nodes in the graph. Thus these can be thought of indicating the terminal nodes of the graph. This is in fact a slippery concept for a cyclic structure. We choose to think of terminal nodes as those which have no successor nodes (as in acyclic structures), or those that have already been “visited” during previous processing of a predecessor node. The set \( t \) represents these previously visited nodes. We will assume for convenience that the local null is in every set of locations.

- the heap \( h_1 \) represents the nodes the left subtree of \( x \) that are visited for the first time in a left-to-right preorder traversal.

- Similarly, the second heap \( h_2 \) represents the nodes the right subtree of \( x \) that are visited for the first time. Clearly this means that \( h_2 \) is separate from both \( x \) and \( h_1 \).

We may now have a definition of the predicate \( \text{graph} \) as follows:

\[
\text{graph}(\text{emp}, \text{emp}, x, t) :- \\
 x = \text{null} \lor x \in t.
\]

\[
\text{graph}(h_x * h_{10} * h_{1b}, h_{2a} * h_{2b}, x, t) :- \\
 h_x \neq x \mapsto (\_, \text{left}, \text{right}), \\
 \text{graph}(h_{1a}, h_{1b} \text{left}, x \cup t), \\
 \text{graph}(h_{2a}, h_{2b} \text{right}, \text{dom}(h_{1a}) \cup \text{dom}(h_{1b}) \cup \{x\} \cup t).
\]

At first glance, it may not be immediate that the recursive rules above really define the two subheaps of a graph as claimed. Let us therefore consider a specific graph, say

![Diagram of a graph](attachment:image.png)

and see how this is one model for the rules. Consider the predicate \( \text{graph}(h_1, h_2, x, \emptyset) \) where \( x \) is node 0. Then the heap \( h_1 \) comprises the nodes 0,1,3,4, while \( h_2 \) comprises just the node 2. Consider \( \text{graph}(h_1', h_2', x' \{0, 1, 3, 4\}) \) where \( x' \) is node 2. This is in fact an expression obtained by unfolding the rules starting from \( \text{graph}(h_1, h_2, x, \emptyset) \). Now \( h_1' \) comprises just the node 2, while \( h_2' = \text{emp} \).

We conclude this subsection with another example, to enhance understanding on how to use recursive definitions to define relevant subheaps. In this example we would like to specify a relationship between two heaps, and not to describe the subheaps of a single
data structure. Such a relationship is useful for example, in specifying a relationship between the global heap at two different program points.

Consider right threaded trees (binary, for simplicity) where the right successor of a node which has an empty right successor is made to point to the node that succeeds this node in an in-order traversal. The following is a specification of the predicate threaded\((h_1, h_2, x, \text{parent})\) which states that, given an unthreaded binary tree in the heap \(h_1\) rooted at \(x\), there is an identical tree in heap \(h_2\) except that the latter tree is threaded. The argument \(\text{parent}\) is a list of locations, initially empty, indicating the potential parent nodes to thread to, encountered while traversing the original tree down to \(x\). Thus if \(x\) had a null right successor, its thread would be the first node in \(\text{parent}\).

\[
\begin{align*}
\text{threaded}(\text{emp}, \text{emp}, \text{null}, \text{null}) & : - \\
\text{threaded}(h_1, h_2, x, \text{parent}) & : - \\
& (x \leftarrow (x, \text{left}, \text{null}) \rightarrow \text{parent}) \\
\text{threaded}(h_1, h_2, x, \text{parent}) & : - \\
& (x \leftarrow (x, \text{left}, \text{right}) \rightarrow \text{parent}) \\
\text{threaded}(h_1, h_2, x, \text{parent}) & : - \\
& (x \leftarrow (x, \text{left} \cup \text{right}, \text{parent}) \rightarrow \text{parent}) .
\end{align*}
\]

We remark that in this example, we have not considered the problem of how to tell if a given (single) tree is threaded or not.

In the rest of this section, we consider three driving examples. The first concerns structure sharing. While the graph example above already shows how to define subheaps according to a traversal strategy terminated by a notion of “visited before”, in the example below we take the idea further by proving the classic marking algorithm on cyclic graphs.

In the second example, we consider the classic problem of copying a DAG. Here the challenge is to prove that the copy, which is a tree and not a DAG, is, in the obvious sense, isomorphic to the original DAG.

In the third and final example, we are interested in the general problem of reasoning about incremental updates to complex data structures. Simply put, real life applications typically update their data structures in different ways, and in different program fragments. The general problem is how to frame away one update from another. Our solution to this general problem can be extrapolated easily from our small example below.

### 5.1 Structure Sharing

The first recursive definition below extends the above definition of graph to become a “partially marked graph”. This is a graph wherein if a node is marked, then all its successor nodes are marked. The second definition below simply states that a graph is fully marked. Note that a node is marked if its “\(m\)” field is 1; otherwise the value is 0.

\[
\begin{align*}
\text{pmgraph}(\text{emp}, \text{emp}, x, t) & : - x \in t. \text{// null \in t} \\
\text{pmgraph}(h_1 \star h_1 \star h_2, h_2 \star h_2, x, t) & : - \\
& h_x \equiv (x \rightarrow (0, \text{left}, \text{right})) \text{// unmarked} \\
\text{pmgraph}(h_1, h_1, \text{left}, \{x\} \cup t) & : - \\
\text{pmgraph}(h_2, h_2, \text{right}, \text{dom}(h_1) \cup \text{dom}(h_2) \cup \{x\} \cup t) .
\end{align*}
\]

We use the following C program below for this:

\[
\begin{align*}
\text{struct node} & \{ \\
& \text{int } m; \text{ struct node } \ast \text{left, } \ast \text{right}; \\
& \}; \\
\text{void mark(struct node } \ast x) & \{ \\
& \text{if } (x \mid x\rightarrow m == 1) \text{ return; } \\
& x\rightarrow m = 1; \\
& \text{mark}(x\rightarrow \text{left}); \\
& \text{mark}(x\rightarrow \text{right}); \\
& \}
\end{align*}
\]

The (inductive) specification of this program fragment is

\[
\begin{align*}
\text{pmgraph}(h_1, h_2, x, t) & \wedge \psi \\
\text{mark}(x) & \{ \\
& (x \mid x\rightarrow m == 1) \text{ return; } \\
& x\rightarrow m = 1; \\
& \text{mark}(x\rightarrow \text{left}); \\
& \text{mark}(x\rightarrow \text{right}); \\
& \}
\end{align*}
\]

where the formula \(\psi\) states that all the heaps therein and the node \(x\) describe the global heap:

\[
\begin{align*}
\mathcal{H} & \equiv (x \rightarrow (\_ \leftarrow \_ \leftarrow \_)) \star h_1 \star h_2 
\end{align*}
\]

The proof can now be done along the lines of the classic proof of markedtree above. As before, we assume that at the start, pmgraph\((h_1, h_2, x, t)\) holds for some \(h\) and \(t\), which are initially empty. Propagating this through the program fragment we first establish that the mark of \(x\) is 0 (but is made 1 in the next statement). At this time we can unfold the predicate pmgraph\((h_1, h_2, x, t)\) using the second rule in the definition of pmgraph and obtain a rather long expression, but the important point is that it contains

\[
\begin{align*}
\text{pmgraph}(h_1, h_1, \text{left}, \{x\} \cup t) & \\
\text{pmgraph}(h_2, h_2, \text{right}, \text{dom}(h_1) \cup \text{dom}(h_2) \cup \{x\} \cup t) \\
\end{align*}
\]

as a subexpression. The first recursive call, because it has the first expression as a precondition, can be inductively reasoned to produce the postcondition

\[
\begin{align*}
\text{ngraph}(h_1, h_1, \text{left}, \{x\} \cup t) \\
\end{align*}
\]

The remaining argument, that this predicate can be framed when considering the second recursive call, follows easily from the fact that this call only operates on heaps \(h_1, h_2\), which are separate from the footprint \(h_1, h_1\) of the above predicate ngraph. We omit further details.
5.2 Summaries

In this example, we consider the copying of a tree into a isomorphic tree. Here the purpose is show how our recursive rules can specify a recursively defined relationship between two portions of the global heap. Along the way, we will also exemplify the use of memory allocation. Consider the C program:

```c
struct node {
    int data;
    struct node *left, *right;
};
struct node *copytree(struct node *x) {
    if (!x) return 0;
    y = malloc(sizeof(struct node));
    y->left = copytree(x->left);
    y->right = copytree(x->right);
    return y;
}
```

The specification we want is a predicate `isocopy(h₁, x, h₂, y)` which states that the tree in `h₁` rooted at `x` has a separate and isomorphic copy, housed in `h₂` rooted at `y`. The following definition achieves this:

```c
isocopy(emp, null, emp, null).
isocopy(h₁, x, h₂, y) :-
    h₁ ≏ h₂,
    h₁ = x → (h₁₁, h₁₂, h₁₃) + h₁₄ + h₁₅,
    h₂ ≏ y → (h₂₁, h₂₂, h₂₃) + h₂₄ + h₂₅,
    isocopy(h₁₁, h₂₁),
    isocopy(h₁₂, h₂₂),
    isocopy(h₁₃, h₂₃),
    isocopy(h₁₄, h₂₄),
    isocopy(h₁₅, h₂₅).
```

Proving the program fragment now is tantamount to proving the triple:

```c
{ tree(h₀ ≏ H, x) } copytree(x) { isocopy(h₀, x, H, Y) }
```

where the ghost variable `h₀` captures the global heap before execution of the program fragment. The proof of this triple can now be done in the style of the proof of the tree marking example at the beginning of this section. The key idea is that the second recursive call operates on a heap that is separate from the footprint of what results from the first recursive call.

5.3 Incremental Updates

In our final example, consider a list representing an employee database. Each node has a “rank” field, either manager or slave, represented by values 1 and 0 respectively, and a “salary” field, represented by some positive integer. In this database, all slaves are poor, i.e. their salaries are less than 99. The point of this example is that we can show that slaves remain poor after we have increased the salaries of managers.

We start with a definition of the employees(h, hₙₐₜ, hₛ, x) predicate which states that `x` is a list in a heap `h` but where the subheaps `hₙₐₜ` represent just the managers’ salaries and `hₛ` represents just the slaves’ salaries. In other words, `h` represents the heap of the rank and `next` fields of nodes, while `hₙₐₜ` and `hₛ` represents the (remaining) fields corresponding to salaries.

```c
employees(emp, emp, emp, null).
employees(h, hₙₐₜ, hₛ, x) :-
    h ≏ (x → 0) + (x + 2 → next) * hₛ,
    hₙₐₜ = (x + 1 → next) * hₙₐₜ,
    employees(hₛ, hₙₐₜ, hₛ, next).
employees(h, hₙₐₜ, hₛ, x) :-
    h ≏ (x → 1) + (x + 2 → next) * hₛ,
    hₙₐₜ = (x + 1 → next) * hₙₐₜ,
    employees(hₛ, hₙₐₜ, hₛ, next).
```

The following two predicates `richmanagers` and `poorslaves` state that managers’ salaries exceed 99 while slaves’ salaries are bounded by 88. The important aspect here is the specification that `h` is the heap representing the list `x` contains just the rank and next fields of the list `x`, while the second heap `hₙₐₜ` represents just the salary field of the managers. The idea is, of course, that if we have a predicate (eg: `poorslaves` below) which does not involve the manager salaries `hₙₐₜ`, then this predicate can be framed over a program which only updates manager salaries.

```c
richmanagers(emp, emp, null).
richmanagers(h, hₙₐₜ, x) :-
    h ≏ (x → 0) + (x + 2 → next) * hₛ,
    richmanagers(hₛ, hₙₐₜ, x).
richmanagers(h, hₙₐₜ, x) :-
    sal > 999,
    h ≏ (x → 1) + (x + 2 → next) * hₛ,
    hₙₐₜ = (x + 1 → next) * hₙₐₜ,
    richmanagers(hₛ, hₙₐₜ, x).
```

Finally, we present a similar definition, this time for poor slaves. Note once again that in this definition, `H` represents the skeletal list and does not include the salary field when the corresponding rank field is not slave.

```c
poorslaves(emp, emp, null).
poorslaves(h, hₛ, x) :-
    sal < 88,
    h ≏ (x → 0) + (x + 2 → next) * hₛ,
    hₛ = (x + 1 → next) * hₛ,
    poorslaves(hₛ, hₛ, x).
```

Consider the triple:

```c
{ employees(h, hₙₐₜ, hₛ, x), poorslaves(h, hₛ, x) } while (x) do
    if (x->rank == 1) x->salary = 999 + 1;
    x = x->next;
endwhile
{ richmanagers(h, hₙₐₜ, x), poorslaves(h, hₛ, x) }
```

The proof of this can now proceed in the style of marktree above. That is, if before the loop body the predicate `poorslaves(h, hₛ, x)` holds, then, by virtue of frame reasoning, it holds after the loop because the heap updates are separate from the footprint of the predicate.

6. Related Work

The explicit naming of heaps has emerged naturally in several extensions of Separation Logic (SL) as an aid to practical program verification. Reynolds conjectured that referring explicitly to the current heap in specifications would allow better handles on data structures with sharing [7]. Duck et al. [3], in this vein, extends Hoare Logic with explicit heaps. This extension allows for
strongest post conditions, and is therefore suitable for "practical program verification" [2] via constraint-based symbolic execution. This paper builds on top of Duck et al. [3], and caters for the purpose of verifying program functional correctness. Consequently, we need to explicitly deal with recursive definitions, while promoting heaps to first class. Specifically, now in the predicate definitions, multiple heaps can be referred to. This leads to the requirement of connecting those heaps to the global heap so that the notion of footprint and a new frame rule can be formalized.

In traditional separation logic, compositionality comes from the usage of the frame rule: we can reason about the effect a program has on the portions of heap it accesses, and reuse that spec in any bigger heap. However, the frame rule no longer works in the presence of an intrinsically shared data structure.

We highlight one important related work: Hobor and Villard [4], whose goals are perhaps very close to ours. This work addresses the problem of sharing and proposes to get back compositionality via applications of the \textit{ramify} rule:

\[
\{ \psi \} P \{ \phi \} \xRightarrow{\text{ramify}} \{ \Pi \} P \{ \phi, \psi, \Pi' \}
\]

where ramification can be expressed as separation logic entailments: \textit{ramify} $\Pi, \phi, \psi, \Pi' \xRightarrow{\text{def}} \Pi \vdash \phi + (\psi -* \Pi')$. These entailments feature the "magic wand" connective of separation logic.

This ramification approach isolates the complicated leap in reasoning at each call site so that the assertions at each program point remain natural, but at the cost of its dependence on the accompanied mathematical objects and the delegation of the remaining proof to the mathematical world. This makes it no longer intuitive and direct as the traditional frame rule. Also a collection of lemmas are need to help reduce the complexity of ramifications and related entailments.

Our approach, however, attacks the sharing problem head on and regains the power of the intuitive frame rule. The cost we need to pay is the more expressive recursive definitions.

Let us revisit the \textit{mark} function for trees in Section 5, but this time considering the application of it on a dag instead of a tree. See Fig. 1. Focus on the propagation of the information, that the right subdag from $2$ is fully marked, over the second recursive call on the node $r$.

Our approach defines the predicate in such a way that we can capture precisely the heap portion being modified by the recursive call with the input $r$, which the heap portion $3$. Consequently, we can directly make use of the frame rule to frame over the fact that the left part has been fully marked.

On the contrary, with ramification, the \textit{ramify} rule would attempt to isolate the shaded heap portion $1$ (in $\psi -* \Pi'$) and prove, in the mathematical world, that the portion $1$ has all been marked. The ramification approach, with the help of the magic wand, seems more general. Its application, however, is counter-intuitive, since the heap portion $1$ is \textit{artificial}: this portion does not correspond to the actual traversal of the code.

There are many other works, orthogonal to ours, but slightly related to concept of explicit heap with framing. Examples are (Implicit) Dynamic Frames [6, 9] and Region Logic [1]. The underlying approach is to represent the heap $H$ as a (possibly implicit) total map over all possible addresses, and to represent access or modification rights as sets of addresses $F$. Separation is represented as set disjoint-ness, i.e. $F_1 \cap F_2 = \emptyset$. None of these consider explicit heaps expressed over general recursive definitions and most importantly, compositional reasoning.

7. Conclusion

We have presented a verification framework based upon an assertion language of explicit heaps over generic values. Recursive definitions over heap and value expressions can be used freely to define data structures, and most importantly, to specify and identify important substructures of these data structures. This solves a major problem in the current standard for compositional reasoning of data structures: Separation Logic. More specifically, the pillar of Separation Logic is its frame rule which allows for the extension of an existing proof to assertions whose footprints are disjoint from this proof. The major problem is when the extension \textit{shares} with the footprint of the original proof, a situation that is common in practice.

In this paper, we started with the expressive heap constraint language of [3] which provided for the definition of arbitrary subheaps of the global heap, thus providing great expressivity, but also provided a full symbolic execution style program verification framework for loop-free programs. The present contribution uses recursive definitions over these heap constraints in order to accommodate (unbounded) data structures. We then showed how these recursive definitions easily captures complex data structures and its parameters. The main contribution however is the formulation of a \textit{footprint} of an assertion, to capture just those heap variables that determine the truth of the assertion. Hereafter it was straightforward to define a frame rule that required only to speak of the heap updates of a program fragment. In contrast to Separation Logic, this involves showing that the heap updates, as opposed to heap accesses, can be enclosed in an expression of some heap variables $h$. Having achieved this, the frame reasoning step was simply that properties on heap locations separate from $h$ can be framed.

In our experimental section, we described three use cases which are most pressing in applications, while at the same problematic for the current state of Separation Logic frameworks. In summary, therefore, our methodology in this paper has broadened of compositional reasoning of data structures, explained in the form of frame reasoning, in a significant way.
References


