Local Reasoning with First-Class Heaps and a New Frame Rule

Duc-Hiep Chu and Joxan Jaffar
National University of Singapore
hiepcd,joxan@comp.nus.edu.sg

Abstract. Separation Logic (SL) brought an advance to program verification of data structures by interpreting (recursively defined) predicates as implicit heaps, and using a separating conjoin operator to construct heaps from disjoint subheaps. While the Frame Rule of SL facilitated local reasoning in program fragments, its restriction to disjoint subheaps means that any form of sharing between predicates is problematic. With this as background motivation, we present (a) 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; (b) a symbolic execution framework, and (c) a new frame rule to accommodate subheaps and non-separating conjoining of subheaps so as to provide compositional reasoning. A key property of the overall framework is that it is automatable. We demonstrate this on two important problem areas that so far have not been adequately solved by SL: summaries of program fragments, and structure sharing.

1 Introduction

An important part of reasoning over heap manipulating programs is the ability to specify properties local to non-overlapping regions of memory. While traditional Hoare logic augmented with recursively defined predicates can be used (from as early as 1982 [14]), it was Separation Logic [17] (SL) which made a significant advance. The key ideas here are: associating a predicate with a notion of heap and composing predicates with the notion of separating conjunction of heaps, and accommodating local reasoning by means of a frame rule. A main shortcoming, however, is that the frame rule may not apply to a predicate which shares its heap with another.

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. In other words, heaps are first-class in this language. Predicates can then be conjoined in the traditional way.

We begin by adopting the constraint language of [9] which introduced explicit heaps and 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 to capture complex properties about both sharing and separation. Our verification framework consists of 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 a triple \{ \phi \} P \{ \psi \} will just require the proof of \psi given the strongest postcondition of \emph{P} from \phi. This result, adapted from \cite{9}, provides a basis for automated verification.

The second and major part of our verification framework is a new frame rule. It is important to note that our frame rule considers only heap \emph{updates} and is specified using explicitly named \emph{subheaps} in order to elegantly isolate relevant portions of the global heap.

The first property we facilitate is the propagation of subheap properties, when they are not involved in heap updates, from the precondition to the postcondition. This is intuitively the key intention of a frame rule: the propagation of unaffected properties. Secondly and just as importantly, the rule needs also to propagate \emph{separation} information. Toward this end, our rule has a concept of \emph{evolution} in a triple: when a collection of subheaps in the precondition evolves to another collection of subheaps in the postcondition, it will follow that separation from the first collection implies separation from the second.

In summary, the main contribution is first an expressive assertion language, then a symbolic execution framework, and finally and most importantly, a new frame rule. This rule is easy to apply in practice because of two reasons: (1) it only concerns heap updates and not reads, and symbolic execution allows us to distinguish them conveniently; and (2) the assertion language easily expresses footprints that are separate from these updates. Thus while SL advanced Hoare reasoning with the implicit use of disjoint heaps, our logic advances SL with the explicit use of arbitrary subheaps. We retain the advantage of SL simply because we encompass it, but we also provide far greater expressivity.

Most importantly, our logic is \emph{automatable}. This is because large proofs can be decomposed using the frame rule, while localized reasoning can utilize the postcondition computation available from the symbolic execution framework. Finally, we will demonstrate our reasoning framework on two well-known instances of important problem areas that so far have not been adequately solved by SL. These are: \emph{summaries} of program fragments, and \emph{structure sharing}.

## 2 Heap Formulas

We overview the \emph{\HL} constraint language for reasoning over \emph{program heaps} first described in \cite{9}. We have a set of \emph{Values} (typically Values \emph{\def} \mathbb{Z}) and we define the set of \emph{Heaps} to be all \emph{finite partial maps} between values, i.e., Heaps \emph{\def} (Values \rightarrowfin Values). We shall distinguish one value called \emph{null} to represent a null pointer, and one heap \emph{emp} to represent an empty heap. Where \(V_v\) and \(V_h\) denote the sets of value and heap variables respectively, our \emph{heap expressions} \(HE\) are as follows:

\[
H ::= V_h \quad v ::= V_v \quad HE ::= H \mid \text{emp} \mid (v \mapsto v) \mid HE \ast HE
\]

An \emph{interpretation} \(\I\) maps \(V_h\) to Heaps and \(V_v\) to Values. Syntactically, a \emph{heap constraint} is a literal of the form \(HE \simeq HE\). An interpretation \(\I\) satisfies a
heap constraint \( (HE_1 \simeq HE_2) \) iff \( \mathcal{I}(HE_1) = \mathcal{I}(HE_2) \) are the same heap, and the separation properties within \( HE_1 \) and \( HE_2 \) hold.

Let \( \text{dom}(H) \) be the domain of the heap \( H \). As shown in \[9\], heap constraints can be normalized into three basic forms:

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

where \( H, H_1, H_2 \in \mathcal{V}_h \) and \( p, v \in \mathcal{V}_v \). Here (Empty) constrains \( H \) to be the empty heap (i.e., \( H = \emptyset \) as a set), (Singleton) constrains \( H \) to be the singleton heap mapping \( p \) to \( v \) (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 \( \mathcal{HL} \)-constraints is presented in \[9\].

In addition to the basic heap constraints, we assume definitions for sub-heap relation \( (H_1 \subseteq H_2) \), domain membership \( (p \in \text{dom}(H)) \), and (overloaded) for brevity, separation relation \( (H_1 * H_2) \). For example, writing \( H_1 \subseteq H_2 \) is equivalent to \( H_2 \simeq H_1 * \_ \_ \) \( p \in \text{dom}(H) \) to \( H \simeq (p \mapsto \_) * \_ \_ \) \( p \notin \text{dom}(H) \) to \( \_ \simeq H * (p \mapsto \_) \) and \( H_1 * H_2 \) to \( \_ \simeq H_1 * H_2 \) where the underscore in each instance denotes a fresh variable.

Finally, we have a recursive constraint. This is an expression of the form \( p(h_1, \ldots, h_n, v_1, \ldots, v_m) \) where \( p \) is a (user-defined) predicate symbol, the \( h_i \in \mathcal{V}_h, 0 \leq i \leq n \) and the \( v_j \in \mathcal{V}_v, 0 \leq j \leq m \). Associated with such a predicate is a recursive definition. We use the framework of Constraint Logic Programming (CLP) \[12\] to inherit its syntax, semantics, and its built-in notions of unfolding rules, for realizing recursive definitions. For brevity, we only informally explain the language. The following rules constitute a recursive definition of predicate \( \text{list}(h, x) \), specifying a skeleton list in the heap \( h \) rooted at \( x \).

\[
\begin{align*}
\text{list}(h, x) & \ :- \ h \simeq \text{emp}, x = \text{null}. \\
\text{list}(h, x) & \ :- \ h \simeq (x \mapsto y) * h_1, \text{list}(h_1, y).
\end{align*}
\]

The semantics of a set of rules is traditionally known as the “least model” semantics \[12\]. 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 that \( h \) is a heap which houses an acyclic list rooted at \( x \) and which is null terminated.

### 3 Programs and the Assertion Language

We assume a vanilla imperative programming language, and we omit a formal definition. The language contains the following heap manipulation statements:

- **heap access** \( (x = \_y) \) sets \( x \) to be the value pointed to by \( y \)
- **heap assignment** \( (x = y) \) sets the value pointed to by \( x \) to be \( y \)
- **heap allocation** \( (x = \text{malloc}(1)) \) points \( x \) to a freshly allocated heap cell
- **heap deallocation** \( (\text{free}(x)) \) deallocates the cell pointed to by \( x \).

\[1\] Here we assume the (de)allocation of single heap cells. This can be easily generalized.
Note that in the program syntax the heap is not explicitly mentioned. Instead, the heap is manipulated using the "∗" notation as in the C language. (There should be no confusion with the separation operator "∗" as understood in Separation Logic or in our heap constraint language.)

Programs operate over an unbounded set of program variables $\mathcal{V}_P$, and these are the value variables and one distinguished heap variable $\mathcal{M}$ to represent the global heap memory. Additional existential or ghost variables may appear in assertions. A ghost variable of type heap will be called a subheap.

The subheaps serve two essential and distinct purposes: (a) to describe subheaps of the global heap $\mathcal{M}$ at the current program point, and (b) to describe the global heap at some other program point.

An assertion $A$ is a formula over $\mathcal{V}_P$ and ghost variables, and is of the form:

$$A ::= VF \mid HF \mid RC \mid A \land A$$

where $VF$ is a value formula, $HF$ a heap constraint and $RC$ a recursive constraint.

An interpretation of an assertion is obtained in the traditional way: value terms are interpreted into values, and heap terms are interpreted into finite maps from values to values, and finally value and heap formulas are interpreted into $false$ or $true$ (in which case it is a model). Consider the annotated C-like program fragment below, which increments all the data values in an acyclic list by 1. Note that we use "∗" in assertions as shorthand for logical conjunction.

```c
struct node {
    int data; struct node *next;
};

{ list(\mathcal{H}, x), \mathcal{H} \subseteq \mathcal{M} }
y = x; while (y) { y->data++; y = y->next; }
{ increment_list(\mathcal{H}_1, \mathcal{H}, x), \mathcal{H}_1 \subseteq \mathcal{M} }
```

As before, the recursive constraint $\text{list}(\mathcal{H}, x)$ describes a heap $\mathcal{H}$ which houses an acyclic list rooted at $x$. The fact that this heap of interest is part of the global heap memory is represented by the constraint $\mathcal{H} \subseteq \mathcal{M}$. Note that unlike before, each node here has a data value in addition to a "next" pointer. Before proceeding, note that in a triple, a ghost variable appearing both in the precondition and postcondition means that we are referring to the same entity e.g., $\mathcal{H}$ above, while this not the case for program variables, e.g., $\mathcal{M}$ and $x$ above.

The recursive constraint $\text{increment_list}(\mathcal{H}_1, \mathcal{H}, x)$ in the postcondition similarly defines that $x$ is a list, but this time the list resides in the heap $\mathcal{H}_1$. It has a second argument, the ghost heap $\mathcal{H}$, which importantly, also appears in the precondition. This allows us to conveniently state the (summary of) changes of

\[ \text{increment_list}(h_1, h_2, x) :- \]
\[ h_1 \bowtie \text{emp}, h_2 \bowtie \text{emp}, x = \text{null}. \]
\[ \text{increment_list}(h_1, h_2, x) :- \]
\[ h_1 \bowtie (x \mapsto (d + 1, \text{next})) \ast h'_1, \]
\[ h_2 \bowtie (x \mapsto (d, \text{next})) \ast h'_2, \]
\[ \text{increment_list}(h_1', h_2', \text{next}). \]

\[ ^2 \text{Hence the terminology "ghost heap".} \]
\[ ^3 \text{Also called a “pure formula” in the literature. We shall not define them further.} \]
\[ ^4 \text{By analogy, the triple } \{ x = x_0 \} \ x++ \{ x = x_0 + 1 \} \text{ describes an increment of } x. \]
subheaps done by the code. The definition of \texttt{increment_list}, shown separately above, states that each datum in the list rooted \texttt{x} in \texttt{H1} is one more that the corresponding datum in the list rooted \texttt{x} in \texttt{H}. It can now be seen that, in the postcondition, (a) the new heap of interest housing the list in the memory is \texttt{H1} (by virtue of the constraint \texttt{H1} \sqsubseteq \texttt{M}), and (b) the values in this list is related to the list in the precondition because the latter list has been captured in the variable \texttt{H}, now at the postcondition is a ghost variable. It is important to note also that the recursive constraint \texttt{list(H, x)} is unaffected by the code and still holds at the postcondition, though its usefulness is questionable. On the other hand, \texttt{M} \sqsubseteq \texttt{H} no longer holds, because the memory \texttt{M} has been updated.

Note that we shall always present the rules that define recursive constraints using fresh variables. Notationally, for heaps, we shall use the small letter ‘h’ in rules, while using the large letter \texttt{H} in assertions.

4 Symbolic Execution and Program Verification

Symbolic execution involves executing a program using \textit{symbolic values} as inputs, and it can be used for program verification in a standard way, as follows. In our setting, an input is an assertion, and is used as a precondition. The output of symbolic execution on a program path is a formula representing the symbolic state obtained at the end of a path, or the \textit{strongest postcondition} of the precondition. For a loop-free program with no function call, symbolic execution facilitates program verification by a disjunction all such path postconditions, which must then imply the desired postcondition. With loops and/or function calls, we often need the help from the frame rule.

We now describe a systematic method to obtain a strongest postcondition from a given precondition \texttt{\phi}.

\textbf{Theorem 1 (Strongest Postcondition).} In the following Hoare-triples, the postcondition shown is the strongest postcondition of the primitive heap operation with respect to a precondition \texttt{\phi}.

\begin{align*}
\{ \texttt{\phi} \} & \quad \texttt{x = malloc(1)} \quad \{ \texttt{alloc(\phi, x)} \} \quad \text{(Heap allocation)} \\
\{ \texttt{\phi} \} & \quad \texttt{free(\phi, x)} \quad \{ \texttt{free(\phi, x)} \} \quad \text{(Heap deallocation)} \\
\{ \texttt{\phi} \} & \quad \texttt{x = \ast y} \quad \{ \texttt{access(\phi, y, x)} \} \quad \text{(Heap access)} \\
\{ \texttt{\phi} \} & \quad \texttt{\ast x = y} \quad \{ \texttt{assign(\phi, x, y)} \} \quad \text{(Heap assignment)}
\end{align*}

where the auxiliary macros \texttt{alloc}, \texttt{free}, \texttt{access}, and \texttt{assign} expand as follows:

\begin{align*}
\texttt{alloc(\phi, x)} & \quad \overset{\text{def}}{=} \quad \texttt{M} \overset{\text{def}}{=} (x \mapsto v) \ast \beta[\texttt{H/ M}, v_1/x] \\
\texttt{free(\phi, x)} & \quad \overset{\text{def}}{=} \quad \beta \overset{\text{def}}{=} (x \mapsto v) \ast \beta[\texttt{H/ M}] \\
\texttt{access(\phi, y, x)} & \quad \overset{\text{def}}{=} \quad \texttt{M} \overset{\text{def}}{=} (y \mapsto x) \ast \beta[\texttt{H/ M}] \\
\texttt{assign(\phi, x, y)} & \quad \overset{\text{def}}{=} \quad \texttt{M} \overset{\text{def}}{=} (x \mapsto v) \ast \beta[\texttt{H/ M}]
\end{align*}

where \texttt{H} and \texttt{H1} are fresh heap variables, and \texttt{v} and \texttt{v1} are fresh value variables. The notation \texttt{\phi[x/y]} means formula \texttt{\phi} with variable \texttt{x} substituted for \texttt{y}. \hfill \square

The proof that the expressions above are indeed the strongest postconditions is intricate but conceptually straightforward. We choose to instead present
a simple example, which should demonstrate the usefulness (and partly the correctness) of Theorem 1.

Consider proving \( \{ H_{99} \approx M \} *x++; *x--; \{ H_{99} \approx M \} \), that is, to prove that the heap is unchanged after an increment and then decrement of the cell \( x \). In order to use Theorem 1 conveniently, let us reduce the program so that only one heap operation is performed per program statement; below we show the rewritten program fragment together with the propagation of the formulas. (For brevity, we also perform a simplification step, where the text color is changed.)

\[
\begin{align*}
\{ H_{99} \approx M \} \\
t_1 &= *x; \\
\{ M \approx (x \mapsto t_1) \ast H_1, H_{99} \approx M \} \\
* x &= t_1 + 1; \\
\{ M \approx (x \mapsto t_1 + 1) \ast H_1, H_2 \approx (x \mapsto t_1) \ast H_1, H_{99} \approx H_2 \} \\
\{ M \approx (x \mapsto t_1 + 1) \ast H_1, H_{99} \approx (x \mapsto t_1) \ast H_1 \} &// \text{ simplification here} \\
t_2 &= *x; \\
\{ M \approx (x \mapsto t_2) \ast H_3, M \approx (x \mapsto t_1 + 1) \ast H_1, H_{99} \approx (x \mapsto t_1) \ast H_1 \} \\
* x &= t_2 - 1; \\
\{ M \approx (x \mapsto t_2 - 1) \ast H_4, H_5 \approx (x \mapsto v) \ast H_4, H_5 \approx (x \mapsto t_2) \ast H_4, H_5 \approx (x \mapsto t_1 + 1) \ast H_1, H_{99} \approx (x \mapsto t_1) \ast H_1 \}
\end{align*}
\]

It is then easy to show that the final formula implies \( H_{99} \approx M \), by first establishing that \( H_1 \approx H_3 \approx H_4 \) and \( v = t_2 = t_1 + 1 \).

5 The Frame Rule

The first pillar of our program verification framework deals with parts of the global heap which are possibly changed. We can now derive, from Theorem 1, the strongest postcondition of an assertion along a program path. Specifically, in order to prove the Hoare triple \( \{ \phi \} S \{ \psi \} \) where \( S \) contains neither loops nor function calls, we simply generate strongest postcondition \( \psi' \) along each of its straight-line paths and obtain the verification condition \( \psi' \models \psi \).

The second and more important pillar of our program verification framework deals with parts of the heap which are definitely unchanged.

Recall that in Separation Logic, a key step is that when a program fragment is “enclosed” in some heap, then any formula \( \pi \) whose “footprint” is separate from this heap can “frame” through the program. In the frame rule above, the premise \( \{ \phi \} P \{ \psi \} \) ensures that the implicit heap arising from the formula \( \phi \) captures all the heap accesses, read or write, in the program fragment \( P \). We now can extend the proof in its premise to that in the conclusion because \( \pi \) can be “framed” by virtue of it being a separate heap from \( \phi \).

In our setting, the concept of enclosure we use is roughly to have an explicit subheap which contains the program updates. These updates are defined to be the cells that the program writes to, or deallocates. This is because the property \( H \sqsubseteq M \), where \( H \) is a ghost variable, is falsified just in case the program code has written to or deallocated some cell in \( H \).
Before formalizing our notion of “enclosure”, however, we first need a concept of heap evolution, as defined below.

**Definition 1 (Evolve).** Given a triple \( T = \{ \phi \} P \{ \psi \} \), we say that a collection \( H \) of subheaps \( H \) in \( \phi \), where \( \phi \models H \subseteq M \), evolves to a collection of subheaps \( H' \) in \( \psi \), where \( \psi \models H' \subseteq M \), if in any model \( I \) of \( T \) and for any (address) value \( v, v \notin \text{dom}(I(H)) \) implies \( v \notin \text{dom}(I(H')) \). We shall use the notation \( H \triangleright H' \) to denote such evolution. \( \square \)

Intuitively, \( H \triangleright H' \) means that the largest \( H' \) can be is \( H \) plus any new cells allocated by the code fragment \( P \). One usage of the evolution concept is as follows: if \( H \triangleright H' \) then any heap is separate from \( H \) at the point of the precondition (i.e., before \( P \) is executed) will be separate from \( H' \) at the point of the postcondition (i.e., after \( P \) is executed).

Consider the **struct node** (for **list** defined in Section 3 and the triple shown in Fig. 1) In this example, we would say that \( H' \) is an evolution of \( H \), or notionally, \( H_1 \triangleright H_1' \). Now assume that the triple represents only a local proof (i.e., there are other parts of \( M \) of interest). How should we compose this local triple to obtain a new triple? One important step toward this is: if \( H_2 \triangleright H_1 \) holds at the precondition for some \( H_2 \), then \( H_2 \triangleright H_1 \) holds at the postcondition. Formally, we have the following.

**Theorem 2 (Propagation of Separation).** Suppose we have a triple \( T = \{ \phi \} P \{ \psi \} \) and that \( \phi \models H_1 \subseteq M, \psi \models H_1' \subseteq M, H \triangleright H' \); then for all \( H_0 : \{ \phi \wedge H_0 \} P \{ \psi \wedge H_0 \} \) holds. \( \square \)

This theorem justifies the Hoare-style rule below:

\[
\begin{align*}
\{ \phi \} P \{ \psi \} & \quad \phi \models H_1 \subseteq M \quad \psi \models H_1' \subseteq M \\
H \triangleright H' & \quad \{ \phi \wedge H_0 \} P \{ \psi \wedge H_0 \}
\end{align*}
\]

(5.1)

We are now ready to describe our notion of enclosure, as discussed above.

**Definition 2 (Enclose).** Suppose we have a triple \( T = \{ \phi \} P \{ \psi \} \) and a collection of subheaps \( H \) in \( \phi \) where \( \phi \models H \subseteq M \). We say \( H \) encloses the updates of \( P \) denoted by \( \text{enclose_upd}(H, P, \phi) \), if in any model \( I \) of \( T \) and at any point of \( P \) which updates the heap location \( x \) (i.e., either \( \text{free}(x) \) or \( \ast x := _{} \)), there exists \( H' \) s.t. \( H' \subseteq M, H \triangleright H', \) and \( x \in \text{dom}(I(H')) \) hold at the update point. \( \square \)

We can now introduce our frame rule. In our setting, the frame rule is all about “preserving the heap reality”. Note that any recursive constraint which contains only ghost variables \( \hat{H} \) (and this is the common situation) remains true from precondition to postcondition. What may no longer hold in the postcondition is the heap reality of \( \hat{H} \), that is, \( H \subseteq M \) may not hold. In other words, given

\[ H \subseteq M \]
local reasoning for a code fragment $P$ and the fact that $\mathcal{H} \subseteq \mathcal{M}$ before executing $P$, how would we preserve this heap reality, without the need to reconsider the code fragment $P$? One answer follows.

**Theorem 3 (Frame Rule).**
Suppose $\{ \phi \} P \{ \psi \}$, $\phi \models \mathcal{H} \subseteq \mathcal{M}$, and $\text{encode}_\text{upd}(\mathcal{H}, P, \phi)$. Then for all $\mathcal{H}_0$, $\{ \phi \land \mathcal{H} \ast \mathcal{H}_0 \land \mathcal{H}_0 \subseteq \mathcal{M} \} \quad P \{ \phi \land \mathcal{H}_0 \subseteq \mathcal{M} \}$ holds. □

This result justifies the Hoare-style rule below:

$\begin{align*}
\{ \phi \} P \{ \psi \} \quad \phi \models \mathcal{H} \subseteq \mathcal{M} \\
\text{enclose}_\text{upd}(\mathcal{H}, P, \phi) \\
\{ \phi \land \mathcal{H} \ast \mathcal{H}_0 \land \mathcal{H}_0 \subseteq \mathcal{M} \} \quad P \{ \psi \land \mathcal{H}_0 \subseteq \mathcal{M} \}
\end{align*}$

(5.2)

Let us demonstrate the use of the two theorems on a very simple example. Consider the triple on the right. Of course, one could follow the symbolic execution rule presented in Section 4 and would be able to prove this triple. This approach, however, is rather tedious since symbolic execution involves introducing a number of existential variables and a substitution.

In contrast, we can start by considering the local reasoning on the triple

$\begin{align*}
\{ \phi \equiv ((x \mapsto \_ ) \subseteq \mathcal{M} ) \} \quad P \equiv \ast x = 1; \ { (x \mapsto 1 ) \subseteq \mathcal{M} } 
\end{align*}$

which holds trivially. Also, we can clearly see that both $(x \mapsto \_ ) \triangleright (x \mapsto 1)$ and $\text{encode}_\text{upd}((x \mapsto \_ ), P, \phi)$ hold. Applying the rule (5.1), we deduce that $(x \mapsto 1) \ast \mathcal{H}$ holds after executing the code fragment $P$. On the other hand, applying the frame rule, rule (5.2), we deduce that $\mathcal{H} \subseteq \mathcal{M}$ remains true, i.e., the heap reality of $\mathcal{H}$ is preserved. Putting the pieces together, we have shown that the truth of the original triple can be established, rather conveniently, by making use of the two theorems.

We now revisit the example on Fig. 1, but now in the context that there exists a tree housed by a heap $\mathcal{H}_2$ that is separate from $\mathcal{H}_1$. From the fact that $\mathcal{H}_1 \triangleright \mathcal{H}_1'$ and $\text{encode}_\text{upd}(\text{emp}, P, \phi)$ hold, where $P$ is the code fragment of interest (shown below), and applying the two rules (5.1) and (5.2), we can establish the validity of the triple below.

$\begin{align*}
\{ \phi \equiv (\text{list}(\mathcal{H}_1, x), \ \text{tree}(\mathcal{H}_2, y), \ \mathcal{H}_1 \ast \mathcal{H}_2 \subseteq \mathcal{M} ) \} \\
\text{z = malloc(sizeof(struct node));} \\
\ast\text{z = x;} \\
\{ \text{list}(\mathcal{H}_1', z), \ \text{tree}(\mathcal{H}_2, y), \ \mathcal{H}_1' \ast \mathcal{H}_2 \subseteq \mathcal{M} \}
\end{align*}$

We now elaborate the connection of our two rules (5.1) and (5.2) with the traditional frame rule in Separation Logic (SL). First, why do we have two rules while SL has one? The reason is that SL, succinctly, captures two important properties: that

- $\pi$ can be added to precondition $\phi$ and it remains true in the postcondition;
- $\pi$ retains its separateness, from precondition $\phi$ to postcondition $\psi$. 

The second property is important for successive uses of the frame rules. Our rule (5.2) above only provides for the first property. We accommodate the second property with the other rule (5.1), i.e., the “propagation of separation” rule.

The two concepts of evolution and enclosure in fact exist in SL, implicitly. Given the triple \( \{ \phi \} P \{ \psi \} \), assume that \( \mathcal{H} \) is the heap housing the precondition \( \phi \) and \( \mathcal{H}' \) is the heap housing the postcondition \( \psi \). In SL, in order for the frame rule work, we are obliged to prove that \( \mathcal{H} \triangleright \mathcal{H}' \) and \( \text{enclose}_{\text{upd}}(\mathcal{H}, P, \phi) \). Because our assertion language allows for the usage of multiple subheaps, which entails more expressive power, we no longer can resort to such a default. Instead, we require the specifications to also nominate the subheaps participating in the evolution and/or enclosure relations. Such relations are stated under the keyword \texttt{frame}, following the typical \texttt{requires} and \texttt{ensures} keywords. We will demonstrate this when we present our driving examples in Section 6.

We finally remark that one is obliged to prove the validity of the \textit{frame conditions}, namely properties about evolution and enclosure of updates, in order to soundly use the two presented rules. For space reasons, and also because the task is routine, we delegate to the Appendix a simple calculus for this.

6 Driving Examples

In the first example, we consider the classic problem of copying a tree. Here the challenge is to prove that the copy, also a tree, is isomorphic to the original tree. The second example concerns structure sharing when dealing with graphs. We will first show how to define subheaps within a recursive definition of cyclic (i.e., general) graphs, then prove the classic marking algorithm on graphs.

6.1 Summaries

Consider the C-like program above. Below, the rules on the left define a standard binary tree, whereas the rules on the right define \texttt{isocopy}(h_1, x, h_2, y) stating...
that the tree in $h_1$ rooted at $x$ has a separate and isomorphic copy, housed in $h_2$ rooted at $y$.

\[ \text{isocopy}(h_x, x, h_y, y) :\]
\[ h_x \approx \text{emp}, x = \text{null}, h_y \approx \text{emp}, y = \text{null}. \]

\[ \text{tree}(h, x) :\]
\[ h \approx \text{emp}, x = \text{null}. \]

\[ \text{isocopy}(h_x, x, h_y, y) :\]
\[ h_x \approx (x \mapsto (\_, \text{left}, \text{right})) \ast h_1 \ast h_2, \]
\[ h_y \approx (y \mapsto (d, \text{left}_y, \text{right}_y)) \ast h_3 \ast h_4, \]
\[ \text{tree}(h_1, \text{left}), \]
\[ \text{tree}(h_2, \text{right}). \]

Below we show the specification of the function \textit{copytree}, together with a proof path. Note the conditions stated within the keyword \textit{frame}. Because the frame conditions are tied to the specific precondition and code fragment, the second and third arguments for \textit{enclose upd}() are unnecessary and thus omitted. In \textit{copytree}, updated cells are only those newly allocated, therefore, they are enclosed by the empty heap \textit{emp} starting at the precondition. (Note that for all $H$, \textit{emp} $\ast H$ holds.) Similarly, because $H_{\text{ret}}$, the heap housing the copy of the original tree, contains only newly allocated cells, it is clear to see that it is evolved from an empty heap, thus \textit{emp} $\vDash H_{\text{ret}}$.

In case $x = \text{null}$, the proof is trivial by unfolding the definition of \textit{isocopy} using the first rule. In the above, we only consider the other case, i.e., $x \neq \text{null}$. Note that certain complex statements are broken down into simpler statements, via the use of temporary fresh variables. For readability, we simplify the formulas using variable substitutions and then eliminating redundant existential variables.

Note that we follow typical strategy of existing proof techniques such as [16], in unfolding the recursive definition on $x$, i.e., \textit{tree}(\textit{H}_x, x), when the code touches $x$’s “footprint”. For brevity, the unfolded formula is abbreviated using $\tau$, and will only be expanded later when necessary.

We now focus on some key steps in the proof below. At program point 4, the step is similar to our simple example discussed in Section 5 where a simple application of the frame rule allows us to maintain $H_x \subseteq M$ while updating the data field of node $y$ simultaneously. Importantly, at program point 5 and 6, where function calls necessitate modular reasoning, we highlight the effects of our rules, namely (5.1) and (5.2), with the constraints in shaded boxes. Finally, at program point 8, the postcondition is proved by renaming $y$ to $\text{ret}$, a special variable denoting the returned value, and unfolding \textit{isocopy}(\textit{H}_x, x, \textit{H}_{\text{ret}}, \text{ret}) in the postcondition using the second rule. We also performed other appropriate variable matching. Thus the proof used here is in fact an instance of U+M, mentioned above.

6.2 Structure Sharing

We begin here with a generic predicate \textit{graph} in Fig. 2(b) which describes a general, possibly cyclic, graph. For simplicity, we assume that each (not \text{null}) node has two successors, named “left” and “right”. The important points:
Let \( \tau(H_x, d, l, r, H_1, H_2) \) be \( H_x \equiv (x \mapsto (d, l, r)) \ast H_1 \ast H_2 \), \( \text{tree}(H_1, l) \), \( \text{tree}(H_2, r) \)

requires: \( \text{tree}(H_x, x) \), \( H_x \subseteq M \)
ensures: \( \text{isocopy}(H_x, x, H_{\text{ret}}, \text{ret}) \), \( H_x \subseteq M \), \( H_{\text{ret}} \subseteq M \)
frame: \( \text{enclose}_{\text{upd}}(\text{emp}, \cdot) \), \( \text{emp} \triangleright H_{\text{ret}} \)

struct node \*copytree(struct node \*x) {
    \{ \text{tree}(H_x, x), H_x \subseteq M \}
    1 assume(x); l = x->left; r = x->right;
    \{ \tau(H_x, l, r, H_1, H_2), H_x \subseteq M \}
    \# y = malloc(sizeof(struct node));
    \{ \tau(H_x, l, r, H_1, H_2), (y \mapsto (\_ \_ \_ \_)) \ast H_x \subseteq M \}
    3 d = x->data;
    \{ \tau(H_x, l, r, H_1, H_2), (y \mapsto (\_ \_ \_ \_)) \ast H_x \subseteq M \}
    \# y->data = d;
    \{ \tau(H_x, l, r, H_1, H_2), (y \mapsto (d, \_ \_ \_ \_)) \ast H_x \subseteq M \}
    \downarrow \quad // \text{(expanding } \tau) \}
    \{ \text{tree}(H_1, l), H_1 \subseteq M, \text{tree}(H_2, r), H_2 \subseteq M \}
    \# (x \mapsto (d, l, r)) \ast H_1 \ast H_2, (y \mapsto (d, \_ \_ \_ \_)) \ast H_x \subseteq M \}
    5 \# z_1 = copytree(l);
    \{ \text{isocopy}(H_1, l, H'_1(z_1), H_1 \subseteq M, H'_1 \subseteq M, \text{tree}(H_2, r), H_2 \subseteq M, \}
    \quad \quad \quad H_x \equiv (x \mapsto (d, l, r)) \ast H_1 \ast H_2, (x \mapsto (d, l, r)) \subseteq M, \}
    \quad \quad \quad \quad (y \mapsto (d, \_ \_ \_ \_)) \ast (x \mapsto (d, l, r)) \ast H_1 \ast H_2 \ast H'_1 \}
    6 \quad z_2 = copytree(r);
    \{ \text{isocopy}(H_1, l, H'_1, H_1 \subseteq M, H'_1 \subseteq M, \text{isocopy}(H_2, r, H'_2, z_2), H_2 \subseteq M, H'_2 \subseteq M, \}
    \quad \quad \quad H_x \equiv (x \mapsto (d, l, r)) \ast H_1 \ast H_2, (x \mapsto (d, l, r)) \subseteq M, \}
    \quad \quad \quad \quad (y \mapsto (d, \_ \_ \_ \_)) \ast (x \mapsto (d, l, r)) \ast H_1 \ast H_2 \ast H'_1 \ast H'_2 \}
    7 \quad y->left = z_1; y->right = t_3;
    \{ \text{isocopy}(H_1, l, H'_1, z_1), H_1 \subseteq M, H'_1 \subseteq M, \text{isocopy}(H_2, r, H'_2, z_2), H_2 \subseteq M, H'_2 \subseteq M, \}
    \quad \quad \quad H_x \equiv (x \mapsto (d, l, r)) \ast H_1 \ast H_2, (x \mapsto (d, l, r)) \subseteq M, \}
    \quad \quad \quad (y \mapsto (d, z_1, z_2)) \ast (x \mapsto (d, l, r)) \ast H_1 \ast H_2 \ast H'_1 \ast H'_2 \}
    8 \# \text{return } y;
}\{ \text{isocopy}(H_x, x, H_y, y), H_x \subseteq M, H_y \subseteq M \}

- the subheaps \( h_1 \) and \( h_2 \) are separate and together house a graph rooted at \( x \) and where the “visited” nodes are kept in the set of values \( t \).
- \( t \) represents a set of locations, “visited” during previous processing of a predecessor node. By construction \( t \) will be disjoint from \( \text{dom}(h_1) \cup \text{dom}(h_2) \).
- 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.

Consider the graph in Fig. 2(a) as a model for \( \text{graph}_\text{root}(h_1, h_2, x) \), or equivalently \( \text{graph}(h_1, h_2, x, \emptyset) \) where \( x \) is node 0. The heap \( h_1 \) comprises nodes 0,1,3,4; while \( h_2 \) comprises just node 2. Consider \( \text{graph}(h_{2a}, h_{2b}, \text{right}, t_2) \)

\[ 6 \quad \text{We have surreptitiously introduced a third type, set, into our recursive rules here.} \]
(a) Illustration: Graph Traversal

(b) Recursive Definitions for Graph

\begin{align*}
\text{graph}\_\text{root}(h_1, h_2, x) & :\neg \text{graph}(h_1, h_2, x, \emptyset). \\
\text{graph}(h_1, h_2, x, t) & :\neg \\
& h_1 \equiv \text{emp}, \ h_2 \equiv \text{emp}, \ x = \text{null} \lor x \in t.
\end{align*}

\begin{align*}
\text{graph}(h_1, h_2, x, t) & :\neg \\
& h_x \equiv (x \mapsto (_, \text{left}, \text{right})), \ \\
& x \not\in t, \ t_1 = t \cup \{x\}, \ \\
& \text{graph}(h_{1a}, h_{1b}, \text{left}, t_1), \ h_1 \equiv h_{1a} * h_{1b}, \ \\
& t_2 = t_1 \cup \text{dom}(h_{1a}) \cup \text{dom}(h_{1b}) \ \\
& \text{graph}(h_{2a}, h_{2b}, \text{right}, t_2), \ h_2 \equiv h_{2a} * h_{2b}.
\end{align*}

Fig. 2: Cyclic Graph

where right is node 2. This is in fact an expression obtained by unfolding \text{graph}(h_1, h_2, x, \emptyset). Now \( h_{2a} \) comprises just node 2, while \( h_{2b} \equiv \text{emp} \).

Before proceeding to actual example of this subsection, we comment on the asymmetry between \text{left} and \text{right} above, which is consistent with a traversal order where the left successor is traversed first. An immediate question is about what happens if the traversal order is in the other direction? A more classic example, in fact, is that of a list, which is traditionally defined tail recursively, and a program loop which moves a pointer starting at the list head and then traversing. The natural loop invariant would stipulate that the structure between the list head and the current pointer is a list segment. A verification problem would then entail proving something about a list given a premise about a list segment, a notoriously hard proof (which eludes the state-of-the-art such as [7,16] and only recently accommodated [8]). We remark here that this disconnect is in general challenging for automated program verification. The important point we wish to make here is that this issue of difference between “motion” in the code and in the rule definitions is orthogonal to this paper.

We now consider the classic marking algorithm:

\begin{verbatim}
struct node {
    int m;
    struct node *left;
    struct node *right;
};

void mark(struct node *x) {
    if (!x || x->m == 1) return;
    struct node *l = x->left, *r = x->right;
    x->m = 1; mark(l); mark(r);
}
\end{verbatim}

Initially the graph is unmarked, and we prove that at the end, the graph is fully marked. Note that when \text{mark} is recursively invoked, and because of sharing, what will be passed on is indeed a partially marked graph, as captured by the first recursive definition below. In a “partially marked graph”, if a node is marked, then all its successor nodes are marked. This is an invariant property that allows us to stop when an already marked node is encountered. 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.

\[\text{pmgraph}(h_1, h_2, x, t) : - \quad h_1 \doteq \text{emp}, h_2 \doteq \text{emp}, x = \text{null} \lor x \in t.\]

\[\text{pmgraph}(h_1, h_2, x, t) : - \quad \text{if } x \not\in (0, \text{left}, \text{right}), \quad x \not\in t, t_1 = t \cup \{x\},\]

\[\text{pmgraph}(h_{1a}, h_{1b}, \text{left}, t_1), h_1 \doteq h_x \ast h_{1a} \ast h_{1b},\]

\[t_2 = t_1 \cup \text{dom}(h_{1a}) \cup \text{dom}(h_{1b}),\]

\[\text{pmgraph}(h_{2a}, h_{2b}, \text{right}, t_2), h_2 \doteq h_{2a} \ast h_{2b}, h_1 \ast h_2.\]

\[\text{mgraph}(h_1, h_2, x, t) : - \quad \text{if } x \not\in (1, \text{left}, \text{right}), \quad x \not\in t, t_1 = t \cup \{x\},\]

\[\text{mgraph}(h_{1a}, h_{1b}, \text{left}, t_1), h_1 \doteq h_x \ast h_{1a} \ast h_{1b},\]

\[t_2 = t_1 \cup \text{dom}(h_{1a}) \cup \text{dom}(h_{1b}),\]

\[\text{mgraph}(h_{2a}, h_{2b}, \text{right}, t_2), h_2 \doteq h_{2a} \ast h_{2b}, h_1 \ast h_2.\]

Below we show the specification\(^7\) of the function/procedure \textit{mark} and how the proof can be established for the interesting case that \(x\) is not null and its \(m\) field has not been marked.

The assertion after step 1 is obtained, as before, by unfolding the definition of \textit{pmgraph} using the second rule (the others are inapplicable), and instantiating the values of \(l\) and \(r\). The assertion after step 3, a recursive call \textit{mark}(1), is obtained by using the specification to replace one occurrence of \textit{pmgraph} (the left one) by \textit{mgraph}. What we would like to focus on here is the shaded heap formula, which was framed through step 3 because this heap lies outside the updates of the recursive call \textit{mark}(1). We also note for this step that \(H_{1a}, H_{1b}\) evolved into \(H'_{1a}, H'_{1b}\) so that the shaded heap’s separation from \(H_{1a}, H_{1b}\) before the step was propagated into its separation from \(H'_{1a}, H'_{1b}\) after the step. This explanation is easily adapted to explain step 4 and the shaded formula after it.

As before, the postcondition is proved by unfolding \textit{mgraph}(\(H'_1, H'_2, x, t\)) in the postcondition using the second rule, followed by appropriate variable matching.

\[7\quad\text{One might recognize that } \text{dom}(H_1) = \text{dom}(H'_1) \text{ implies } H_1 \vDash H'_1 \text{ and that } \text{dom}(H_2) = \text{dom}(H'_2) \text{ implies } H_2 \vDash H'_2. \text{ Thus for this example, we do not need to separately construct the proofs for the evolution relations.}\]

7 Related Work

It is possible, but very difficult, to reason in Hoare logic about programs which address and modify data structures defined by pointers. Early works [144] explore this direction. The resulting proofs are rather inelegant and remains too low-level to be widely applicable, let alone being automated.
It was Separation Logic [17][15] (SL) which made a significant advance with local reasoning via a frame rule, influencing modern verification tools that deal with heap manipulating programs. Unfortunately, we usually cannot use the frame rule directly when verifying data structures with unrestricted sharing.

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 [18]. With this as motivation background, the work [9] extends Hoare Logic with explicit heaps, contributing a basic constraint-based symbolic execution, and is therefore suitable for “practical program verification” [9]. This paper extends [9] to deal with recursive definitions, maintaining the stature of first-class (sub-)heaps. What is new is a connecting of subheaps to the global heap and formalizing the concepts of “evolution" and “enclosure", leading to a new and practical frame rule.

It is important to note that several systems [2][1][1] implement Separation Logic-based symbolic execution, as described in [3]. However, due to the memory-safety requirements of Separation Logic, symbolic execution is limited to formulæ over the footprint of the code. In contrast, our symbolic execution, following [9], works for arbitrary formulæ. This is convenient when memory-safety is not the only property of interest.

We next mention [10], which also addresses the problem of sharing, and proposes to regain compositionality

\[ \{ \phi \} \ P \ { \psi \} \ ramify(\Pi, \psi, \phi, \Pi') \]

\[ \{ \Pi \} \ P \ { \Pi' \} \]
via applications of the *ramify* rule shown above, where $\text{ramify}(\Pi, \psi, \phi, \Pi') \overset{\text{def}}{=} \Pi \vdash \psi \ast (\phi \ast \Pi')$. This approach isolates the complicated heap 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 as the traditional frame rule. In general, a collection of lemmas are needed to help reduce the complexity of ramifications and related entailments, thus limiting the opportunity for automation.

Our approach, however, attacks the sharing problem head on and regains the power of the intuitive frame rule, posing great promises for automation. The cost we need to pay is the more expressive specifications.

Let us revisit the *mark* function in Section 6 but consider the application of it on a DAG instead. See Fig. 3. Focus on the second recursive call of the function. Explicit heaps allow us to capture precisely the heap portion, portion 3, being modified by the (second) recursive call with the input $r$. Consequently, we can easily frame over the fact that the left part has been fully marked.

On the contrary, the ramify rule \[10\] would attempt to isolate the shaded heap portion 1 (in $\phi \ast \Pi'$) and prove that the portion 1 has all been marked. With the help of the magic wand, this approach seems more general. Its application, however, is counter-intuitive and hard to automate, since the heap portion 1 is *artificial*: this portion does not correspond to the actual traversal of the code.

There are other orthogonal works, but slightly related to concept of explicit heap with framing. Examples are (Implicit) Dynamic Frames \[13\] and Region Logic \[1\]. The underlying approach is to represent the heap $M$ 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 disjointness, i.e., $F_1 \cap F_2 = \emptyset$. None of these consider explicit heaps expressed over general recursive definitions and most importantly, compositional/local reasoning.

**Conclusion** We have presented a program logic which extends Separation Logic by extending the use of predicates with explicit subheaps in assertions. Its first advantage is expressibility, and in particular, we can express sharing in predicates and summaries across program fragments. A second advantage is that the assertions are amenable to symbolic execution. Third, we presented a frame rule which allows for local reasoning in the new and more expressive setting. Consequently, our logic is automatable.
References

A Appendix: A Calculus for Proving Frame Conditions

In Fig. 4 and Fig. 5, we use $T \rightsquigarrow R$ to denote that the relation $R$ (evolution or enclosure) is tied to the triple $T$. Also, $\text{post}(\phi, s)$ denotes the postcondition computation given the precondition $\phi$ and a (generic) program statement $s$.

\[
\begin{align*}
\text{[MALLOC]} & \quad \phi \models \mathcal{H} \subseteq M \quad \text{alloc}(\phi, x) \vdash \psi \\ & \quad \psi \models \mathcal{H}' \subseteq M \quad \text{dom}(\mathcal{H}') \subseteq \text{dom}(\mathcal{H}) \cup \{x\} \\
& \quad \{ \phi \} \ x := \text{malloc}(1) \ {\psi} \models \mathcal{H} \triangleright \mathcal{H}'
\end{align*}
\]

\[
\begin{align*}
\text{[FREE]} & \quad \phi \models \mathcal{H} \subseteq M \quad \text{free}(\phi, x) \vdash \psi \\ & \quad \psi \models \mathcal{H}' \subseteq M \quad \text{dom}(\mathcal{H}') \subseteq \text{dom}(\mathcal{H}) \ \{x\} \\
& \quad \{ \phi \} \ \text{free}(x) \ \{\psi\} \models \mathcal{H} \triangleright \mathcal{H}'
\end{align*}
\]

\[
\begin{align*}
\text{[OTHER-STATEMENTS]} & \quad \phi \models \mathcal{H} \subseteq M \quad \text{post}(\phi, s) \vdash \psi \\ & \quad \psi \models \mathcal{H}' \subseteq M \quad \text{dom}(\mathcal{H}') \subseteq \text{dom}(\mathcal{H}) \\
& \quad \{ \phi \} \ s \ \{ \psi \} \models \mathcal{H} \triangleright \mathcal{H}'
\end{align*}
\]

\[
\begin{align*}
\text{[SEQ-COMPOSITION]} & \quad \{ \phi \} \ P \ \{ \psi \} \models \mathcal{H} \triangleright \mathcal{H}' \\
& \quad \{ \psi \} \ Q \ \{ \gamma \} \models \mathcal{H}' \triangleright \mathcal{H}'' \\
& \quad \{ \phi \} \ P ; Q \ \{ \gamma \} \models \mathcal{H} \triangleright \mathcal{H}''
\end{align*}
\]

\[
\begin{align*}
\text{[COMPOSITION]} & \quad \{ \phi \} \ P \ \{ \psi \} \models \mathcal{H}_1 \triangleright \mathcal{H}'_1 \\
& \quad \{ \phi \} \ P \ \{ \psi \} \models \mathcal{H}_2 \triangleright \mathcal{H}'_2 \\
& \quad \{ \phi \} \ P \ \{ \psi \} \models (\mathcal{H}_1 \cup \mathcal{H}_2) \triangleright (\mathcal{H}'_1 \cup \mathcal{H}'_2)
\end{align*}
\]

\[
\begin{align*}
\text{[IF-THEN-ELSE]} & \quad \{ \phi \} \ \text{assume}(b); P_1 \ \{ \psi_1 \} \models \mathcal{H} \triangleright \mathcal{H}'_1 \\
& \quad \mathcal{H}'_1 \subseteq \mathcal{H}'_1 \\
& \quad \{ \phi \} \ \text{assume}(\neg b); P_2 \ \{ \psi_2 \} \models \mathcal{H} \triangleright \mathcal{H}'_2 \\
& \quad \mathcal{H}'_2 \subseteq \mathcal{H}'_2 \\
& \quad \{ \phi \} \ \text{if} \ (b) \ \text{then} \ P_1 \ \text{else} \ P_2 \ \{ \psi_1 \lor \psi_2 \} \models \mathcal{H} \triangleright \mathcal{H}'
\end{align*}
\]

Fig. 4: Hoare-style Rules for Evolution ($\triangleright$)
[HEAP–ASSIGN]
\[\phi \models \mathcal{H} \sqsubseteq \mathcal{M} \quad \text{assign}(\phi, x, y) \vdash \psi \quad x \in \text{dom}(\mathcal{H})\]
\[\{ \phi \} \ast x := y \{ \psi \} \rightsquigarrow \text{enclose}_\text{upd}(\mathcal{H}, \ast x := y, \phi)\]

[FREE]
\[\phi \models \mathcal{H} \sqsubseteq \mathcal{M} \quad \text{free}(\phi, x) \vdash \psi \quad x \in \text{dom}(\mathcal{H})\]
\[\{ \phi \} \text{free}(x) \{ \psi \} \rightsquigarrow \text{enclose}_\text{upd}(\mathcal{H}, \text{free}(x), \phi)\]

[OTHER–STATEMENTS]
\[\phi \models \mathcal{H} \sqsubseteq \mathcal{M} \quad \text{post}(\phi, s) \vdash \psi\]
\[\{ \phi \} s \{ \psi \} \rightsquigarrow \text{enclose}_\text{upd}(\mathcal{H}, s, \phi)\]

[SEQ–COMPOSITION]
\[\{ \phi \} P \{ \psi \} \rightsquigarrow \text{enclose}_\text{upd}(\mathcal{H}, P, \phi)\]
\[\{ \psi \} Q \{ \gamma \} \rightsquigarrow \text{enclose}_\text{upd}(\mathcal{H}', Q, \psi)\]
\[\{ \phi \} P; Q \{ \gamma \} \rightsquigarrow \text{enclose}_\text{upd}(\mathcal{H}, P; Q, \phi)\]

[COMPOSITION]
\[\{ \phi \} P \{ \psi \} \rightsquigarrow \text{enclose}_\text{upd}(\mathcal{H}, P, \phi)\]
\[\phi \models \mathcal{H}' \sqsubseteq \mathcal{M}\]
\[\{ \phi \} P \{ \gamma \} \rightsquigarrow \text{enclose}_\text{upd}((\mathcal{H} \cup \mathcal{H}'), P, \phi)\]

[IF–THEN–ELSE]
\[\{ \phi \} \text{assume}(b); P_1 \{ \psi_1 \} \rightsquigarrow \text{enclose}_\text{upd}(\mathcal{H}, \text{assume}(b); P_1, \phi)\]
\[\{ \phi \} \text{assume}(\neg b); P_2 \{ \psi_2 \} \rightsquigarrow \text{enclose}_\text{upd}(\mathcal{H}, \text{assume}(\neg b); P_2, \phi)\]
\[\{ \phi \} P \equiv \text{if } (b) \text{ then } P_1 \text{ else } P_2 \{ \psi_1 \lor \psi_2 \} \rightsquigarrow \text{enclose}_\text{upd}(\mathcal{H}, P, \phi)\]

Fig. 5: Hoare-style Rules for Updates (\text{enclose}_\text{upd}(\cdot,\cdot,\cdot))