Local Reasoning on Recursive Data Structures

DUC-HIEP CHU, IST Austria
JOXAN JAFFAR, National University of Singapore

We consider the problem of verifying programs which manipulate recursive data structures. A main challenge here is how to perform local reasoning so that the verification of subprograms, which operate on different components of the data structure, can be combined. Separation Logic (SL) was a significant advance in program verification of data structures. It used a "separating" conjoin operator in data structure specifications to construct heaps from disjoint subheaps, and a "frame rule" to very elegantly realize local reasoning. Consequently, when a program is verified in SL, the proof is very natural and succinct. However, SL does not easily accommodate *shared* or *cyclic* data structures.

In this paper, our new framework serves to maintain the essential advantage, local reasoning, of SL. We begin with a domain of discourse of explicit subheaps with recursive definitions. The resulting specification language can describe arbitrary data structures, and arbitrary sharing therein, thus enabling a very precise specification of frames. The main contribution then is a program verification method which combines strongest postcondition reasoning in the form *symbolic execution*, and *unfolding recursive definitions* of the data structure in question. Finally, we present an implementation of our verifier, and demonstrate automation on a number of representative programs. In particular, we present the first automatic proof of a classic graph marking algorithm, paving the way for dealing with a class of programs which traverse a complex data structure.

1 INTRODUCTION

We consider the problem of automatically verifying programs which manipulate data structures. The data structures we consider are *recursive*, that is, their formal definition is represented by a recursively defined relation. Many if not most data structures used in practice are recursive. Automatic verification of such programs is challenging, as evidenced by the lack of a systematic method which can routinely prove textbook examples. We believe that one reason is the difficulty of dealing with shared and/or cyclic structures, and in particular, how to perform *local reasoning* on them.

In general, reasoning about heap manipulating programs requires the specification of properties local to regions of memory. In traditional Hoare logic, there is a *frame rule* (CFR) which allows an assertion, which does not mention heap variables or pointers, to be "framed" through a program fragment. There is a side condition that the program fragment does not modify any (free) variable in the assertion. Augmenting this rule to accommodate assertions containing recursively defined predicates has be used from as early as 1982 [5, 23].

Proposition 1.1 (Classic Frame Rule).

$$\frac{\{\phi\}\ P\ \{\psi\}}{\{\phi \wedge \pi\}\ P\ \{\psi \wedge \pi\}}\ Mod(P) \cap FV(\pi) = \emptyset \tag{CFR}$$

where $Mod(P)$ denotes the variables that P modifies, and $FV(\pi)$ denotes the free variables of π. □

It was Separation Logic [26, 29] (SL) which made a significant advance. Two key ideas here are: associating a predicate with a notion of *heap*, and composing predicates with the notion of *separating conjunction* of heaps. As a result, SL has an extremely elegant frame rule (SFR). The key idea is that when a program fragment is "enclosed" in some heap, then any formula π whose "footprint" is separate from this heap can be "framed" through the program. This notion of separation is indicated by the "separating conjunction" operator "*" which states that the footprints of its two operands (which are logical predicates) are disjoint. Note that the validity of the triple $\{\phi\}\ P\ \{\psi\}$ entails that all heap accesses in P, read or write, are confined to the implicit heap of ϕ, or to fresh addresses. This provides for truly local reasoning, because the proof of P is done without any *prior knowledge* about the frame π.

At this point, note there are *two kinds of footprints* at play. One concerns what is associated with the specification describing some data structure properties, called *specification footprint*; and the other, that is concerned with the heap updates in the code or simply the *code footprint*. The key issue is how to *connect* these two in the verification process, so that framing can take place. As mentioned above, SL admirably addresses these two footprints, and their connections, and there SL facilitates the important methodology of *local reasoning*.

$$\frac{\{\phi\}\ P\ \{\psi\}}{\{\phi * \pi\}\ P\ \{\psi * \pi\}}\ \text{(SFR)}$$

There is, however, a critical limitation of SL. A recursive definition of a data structure, comprising specifications of its constituent sub-data structures, may use a "separating conjunction" connective. This means that these sub-data structures must have disjoint footprints. Clearly this is a very significant restriction, omitting the proof of *shared* data structures. (See [13] for a detailed discussion of this.)

We now present an example problem of marking a (possibly cyclic) graph, in Fig. 1. Note that we have here a shared data structure, and so we cannot (directly) apply SL. Instead, we shall demonstrate what it would take to prove this program, with emphasis on roles of the specification and code footprints mentioned above. For simplicity, let a graph node have two successor fields `left` and `right`.

```
struct node {
   int mark;
   struct node *left;
   struct node *right;
};

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

Fig. 1. Mark Graph Example

The top-level precondition is that the graph is unmarked, and the postcondition is that the graph is fully marked. Because the function is recursive, clearly its precondition cannot simply be that the graph is fully unmarked. The required precondition is rather complicated, and we relegate the details to section 5. Here it suffices to say that the precondition must state that every encountered marked node is either previously encountered, or all of its successor nodes are already marked. The take-away is that this property is not naturally expressible without using a recursive definition. Furthermore, to have local reasoning, the first recursive call must not destroy what is needed as the precondition of the second call, and the second call should *not negate* the effects of the first. In other words, we need to describe: (1) the write footprint of the first call, (2) the footprint of the precondition of the second call, (3) the footprint of the postcondition of the first call, and (4) the write footprint of the second call. The verification process needs to "connect" and figure out that (1) and (2) are disjoint and that (3) and (4) are also disjoint.

We are unaware of any existing systematic method that can verify this example. But perhaps more importantly, this example displays a formal of a basic algorithm which traverses a complex data structure and is able to connect the formal specification, written in the standard form of a well-founded recursive definition, and the code, which operates using the standard method of marking. We elaborate in Section 5.

In this paper, we begin with an assertion language in which subheaps may be *explicitly* defined within predicates [11], and the effect of separation obtained by specifying that certain heaps are disjoint. In other words, heaps are *first-class* in this language. One main contribution of [11] is to refine the "overloaded meaning" of the separation conjunction, so that predicates can be conjoined in the traditional way. In this paper, we first extend the assertion language of [11] by removing the *implicit* "heap reality" of any subheaps appearing in a recursive predicate. Instead, heap reality is explicitly specified by connecting (ghost) subheaps to the distinguished heap variable $\mathcal{M}$, which represents the global heap memory at the current state. We then show how to capture complex properties about *both* sharing and separation.

Our verification framework consists of two parts. In the first part, we deal with the part of a heap that is *possibly changed* by a straight-line program fragment. This is handled by a *strongest postcondition* transform, so that the proof of a triple $\{\phi\}\ P\ \{\psi\}$ will just require the proof of ψ given the strongest postcondition of P from ϕ. The transformation, inherited from [11], can be easily automated, providing a basis towards automated verification.

Our contribution lies in the second part of the verification framework: to perform compositional reasoning by automatically framing properties of heap that are *definitely unchanged*. Indeed, the main contribution of this paper is a new frame rule to reclaim the power of local reasoning. Before proceeding, let us detail why the traditional frame rule from SL cannot be simply adapted to our new specification language with explicit heaps. A first reason is explained in [11]: that with a *strongest postcondition* approach to program verification, the frame rule, suitably translated into the language of explicit heaps, is simply *not valid*. In other words, if $\{\phi\}\ P\ \{\psi\}$ is established because ψ follows from the strongest postcondition of P executed from ϕ, it is not the case that any heap separate from ϕ remains unchanged by the execution of P. A second reason is that while the assertion language refers to multiple heaps, only those which are affected by the program must be isolated. In contrast, the traditional rule deals with a single (implicit) heap and so separation refers unambiguously to this heap alone.

Our new frame rule is used by explicitly naming *subheaps* in the specifications as part of the frame, in order to elegantly isolate relevant portions of the global heap $\mathcal{M}$. Consequently, a significant distinction is that our frame rule is concerned only on heap *updates*, as opposed to *all* heap references as in traditional SL.

More specifically, we firstly facilitate the propagation of subheap properties from the precondition to the postcondition, when they are not involved in program heap updates. 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 *separation* information. Toward this end, we introduce a concept of *evolution* in a triple: when a collection of subheaps in the precondition evolves to another collection of subheaps in the postcondition, it follows that separation from the first collection implies separation from the second. Thus while SL advanced Hoare reasoning with the implicit use of disjoint heaps, our logic advances SL with the explicit use of arbitrary subheaps.

Finally, we give evidence that our verification framework has a good level of automation. In Section 5, we automatically prove one significant example for the first time: marking a graph. This example exhibits important relationships between data structures that have so far not been addressed by automatic verification: processing recursive data structures with sharing. We will present an implementation in Section 6, submitted as supplementary material for this paper, and a demonstration of automatic verification on a number of representative programs. We demonstrate the phases of specification, verification condition generation and finally theorem-proving. We stress here that we shall be using *existing* and not custom technology for the theorem-proving.

In summary, we contend that a new large of applications is now automatically verifiable in the sense that there is an expressive assertion language of heaps, there is a symbolic execution algorithm to generate verification conditions from annotated programs, and finally, these conditions can be dispatched by standard techniques of unfolding recursive definitions.

2 THE ASSERTION LANGUAGE

We assume a vanilla imperative programming language with functions but no loops (which are tacitly compiled into tail-recursive functions). Other than standard non-heap statements, the language contains The following are *heap manipulation statements:*[1]

- sets x to be the value pointed to by y: `x = *y;`
- sets the value pointed to by x to be y: `*x = y;`
- points x to a freshly allocated cell: `x =` **malloc**`(1);`
- deallocates the cell pointed to by x: **free**`(x).`

The heap is not explicitly mentioned in the program. Instead, it is dereferenced using the "$*$" notation as in the C language. (Not to be confused with the operator "$*$" in SL or our heap constraint language.) Since our later discussion will involve symbolic execution, we also assume that branch condition is translated to *assume*(_) statement. We first give a brief overview of Hoare and Separation Logic, then we introduce the language in [11].

[1] We assume (de)allocation of single heap cells; this can be easily generalized, and indeed so in our implementation.

2.1 Background

Hoare Logic [12] is a formal system for reasoning about program correctness. Hoare Logic is defined in terms of axioms over *triples* of the form $\{\phi\}\ P\ \{\psi\}$, where ϕ is the *precondition*, ψ is the *postcondition*, and P is some code fragment. Both ϕ and ψ are formulae over the *program variables* in P. The meaning of the triple is as follows: for all program states σ_1, σ_2 such that $\sigma_1 \models \phi$ and executing σ_1 through P derives σ_2, then $\sigma_2 \models \psi$. For example, the triple $\{x < y\}\ \mathtt{x} = \mathtt{x} + 1\ \{x \leq y\}$, x and y are integers, is *valid*. Note that under this definition, a triple is automatically valid if P is non-terminating or otherwise has undefined behavior. This is known as *partial correctness*.

Separation Logic (SL) [29] is a popular extension of Hoare Logic [12] for reasoning over *heap manipulating programs*. SL extends predicate calculus with new logical connectives – namely *empty heap* (**emp**), *singleton heap* ($p \mapsto v$), and *separating conjunction* ($F_1 {*} F_2$) – such that the structure of assertions reflects the structure of the underlying heap. For example, the precondition in the following valid Separation Logic triple

$$\{x \mapsto \_ * y \mapsto 2\}\ {*}\mathtt{x} = {*}\mathtt{y} + 1\ \{x \mapsto 3 * y \mapsto 2\}$$

represents a heap comprised of two *disjoint singleton* heaps, indicating that both x and y are *allocated* and that location y points to the value 2. In the postcondition, x points to value 3, as expected. SL also allows *recursively-defined* heaps for reasoning over data structures, such as **list** and **tree**. An SL *triple* $\{\phi\}\ P\ \{\psi\}$ additionally guarantees that any state satisfying ϕ will not cause a memory access violation in P. For example, the triple $\{\mathbf{emp}\}\ {*}\mathtt{x} := 1\ \{x \mapsto 1\}$ is *invalid* since x is a dangling pointer in a state satisfying the precondition.

A Constraint Language of Explicit Heaps [11]: We have a set of Values (e.g. integers) and we define Heaps to be all *finite partial maps* between values, i.e., $\mathsf{Heaps} \overset{\text{def}}{=} (\mathsf{Values} \rightharpoonup_{\text{fin}} \mathsf{Values})$. There is a special value null ("null" pointer) and a special heap emp ("empty" heap). Where $\mathcal{V}_v$ and $\mathcal{V}_h$ denote the sets of value and heap variables respectively, our *heap expressions* *HE* are as follows:

$$\begin{aligned} H &::= \mathcal{V}_h \\ v &::= \mathcal{V}_v \\ \mathit{HE} &::= H \mid \mathsf{emp} \mid (v \mapsto v) \mid \mathit{HE} * \mathit{HE} \end{aligned}$$

An *interpretation* $\mathcal{I}$ maps $\mathcal{V}_h$ to Heaps and $\mathcal{V}_v$ to Values. Syntactically, a *heap constraint* is of the form $(\mathit{HE} \simeq \mathit{HE})$. An interpretation $\mathcal{I}$ satisfies a heap constraint $(\mathit{HE}_1 \simeq \mathit{HE}_2)$ iff $\mathcal{I}(\mathit{HE}_1) = \mathcal{I}(\mathit{HE}_2)$ are the same heap, and the separation properties within HE_1 and HE_2 hold.

Let $dom(H)$ be the *domain* of the heap H. As in [11], heap constraints can be normalized into three basic forms:

$$\begin{aligned} &H \simeq \mathsf{emp}\ (\textsc{Empty}) \\ &H \simeq (p \mapsto v)\ (\textsc{Singleton}) \\ &H \simeq H_1 * H_2\ (\textsc{Separation}) \end{aligned}$$

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) constraints H to be the 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 $dom(H_1) \cap dom(H_2) = \emptyset$).

We will also use *sub-heap relation* ($H_1 \sqsubseteq H_2$), *domain membership* ($p \in dom(H)$), and (overloaded) for brevity, *separation relation* ($H_1 * H_2$). In fact, writing $H_1 \sqsubseteq H_2$ is equivalent to $H_2 \doteq H_1 * \_$, $p \in dom(H)$ to $H \doteq (p \mapsto \_) * \_$, $p \notin dom(H)$ to $\_ \doteq H * (p \mapsto \_)$, and $H_1 * H_2$ to $\_ \doteq H_1 * H_2$; where the underscore in each instance denotes a fresh variable.

Finally, we have a *recursive constraint*. This is an expression of the form

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

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 symbol is a *recursive definition*. We use the framework of *Constraint Logic Programming* (CLP) [15] to inherit its syntax, semantics, and its built-in notions of unfolding rules, for realizing recursive definitions. The *semantics* of a set of rules is traditionally known as the "least model" semantics [15]. For brevity, we only informally explain the language. The following constitutes a recursive definition of $\texttt{list}(h, x)$, specifying a *skeleton list* in the heap h rooted at x.

$$\texttt{list}(h, x) \ \texttt{:-}\ h \doteq \mathsf{emp}, x = \mathsf{null}.$$
$$\texttt{list}(h, x) \ \texttt{:-}\ h \doteq (x \mapsto y) * h_1,\ \texttt{list}(h_1, y).$$

Note that the comma-separated expressions in the body of each rule is either *value constraint* (e.g. $x = \mathsf{null}$), a heap constraint (e.g. $h \doteq \mathsf{emp}$), or a recursive constraint (e.g. $\texttt{list}(h_1, y)$). In this paper, our value (i.e. "pure") constraints will either be arithmetic or basic set constraints over values.

2.2 Program Verification with Explicit Heaps

Hoare Triples: We first define an *assertion* A as a formula over $\mathcal{V}_v$, $\mathcal{V}_h$:

$$A ::= VF \mid HF \mid RC \mid A \wedge A \mid A \vee A$$

where VF, HF, and RC are value, heap, and recursive constraints, respectively. We next connect the interpretation of assertions with the program semantics.

Programs operate over an unbounded set of *program variables* $\mathcal{V}_P$, which are the *value variables*. Thus $\mathcal{V}_P \subseteq \mathcal{V}_v$. We use one distinguished heap variable $\mathcal{M} \in \mathcal{V}_h$ to represent the *global heap memory*. Variables other than the program variables and $\mathcal{M}$ may appear in assertions; they are existential or *ghost* variables. 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 some other "existential" heap. A common instance of (b) is the heap corresponding to the global heap at some *other* program point in the past.

We use the terminology "ghost heap" in accordance to standard practice that subheaps are existential, but in assertions, they can be used to constrain the value of the global heap. Importantly, as ghost variables, their values *cannot be changed* by the program. We will see later that this is important in practice because (a) predicates in assertions often need to be defined only using ghost subheaps, and (b) it is automatic that these predicates can be "framed through" any program fragment P because P cannot change the value of a ghost variable.

Before proceeding, we stress that our *interpretation* of triples follows Hoare logic: the postcondition holds *provided* the start state satisfies the precondition, *and* there is a terminating execution of the program. In contrast, in SL, a triple entails that the program is memory-safe.

Note that we shall present 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 $\mathcal{H}$ in assertions. Also, we use "**,**" in assertions as shorthand for logical conjunction.

Example: see the annotated program and the definition of `inc_list` in Fig. 2. The program increments all the data values in an acyclic list by 1.

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

```
{ $list(\mathcal{H}, x)$, $\mathcal{H} \sqsubseteq \mathcal{M}$ }
  y = x;
  while (y) {
    y->data += 1;
    y = y->next;
  }
{ $inc\_list(\mathcal{H}_1, \mathcal{H}, x)$, $\mathcal{H}_1 \sqsubseteq \mathcal{M}$ }
```

```
inc_list($h_1, h_2, x$) :-
   $h_1 \doteq \mathsf{emp}, h_2 \doteq \mathsf{emp}, x = \mathsf{null}$.
inc_list($h_1, h_2, x$) :-
   $h_1 \doteq (x \mapsto (d+1, next)) * h_1'$,
   $h_2 \doteq (x \mapsto (d, next)) * h_2'$,
   inc_list($h1'$, $h_2'$, $next$).
```

Fig. 2. Incrementing data values in an acyclic list.

As before, $\texttt{list}(\mathcal{H}, x)$ describes a heap $\mathcal{H}$ which houses an acyclic list rooted at x. The constraint $\mathcal{H} \sqsubseteq \mathcal{M}$ states that it resembles a part of the global heap. The other recursive constraint $\texttt{inc\_list}(\mathcal{H}_1, \mathcal{H}, x)$ similarly defines that x is the head of a list resides in the heap $\mathcal{H}_1$. It has another argument, the ghost heap $\mathcal{H}$, which also appears in the precondition. This, importantly, allows us to consider the triple as a *summary*, relating values in the precondition and postcondition (using the ghost variable as an anchor value). In this case, we are stating that the final list elements are one bigger than the corresponding initial elements. Further, we are also stating that all the links (the *next* pointers) are not modified.

We conclude this section by mentioning that our framework does not provide for memory safety as an intrinsic property. We can easily enforce memory safety by, e.g., asserting that dereferences (e.g., `x->next`) and deallocations (e.g., `free(x)`), have their arguments (`x`) pointing to a valid cell in the current global heap ($x \in dom(\mathcal{M})$). Not enforcing memory safety up front is not a weakness of the framework. It allows us to be flexible enough to perform reasoning even when memory safety is not the property of interest. Furthermore, SL may disapprove of a memory safe program whose specifications of some functions are not sufficiently complete. In contrast, our framework can still proceed, but possibly not by means of local reasoning, for example.

3 SYMBOLIC EXECUTION WITH EXPLICIT HEAPS

Symbolic execution of a program uses *symbolic values* as inputs, and can be used for program verification in a standard way. We start with 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 *strongest postcondition* of the precondition. For a loop-free program with no function calls, symbolic execution facilitates verification by considering a disjunction of all such path postconditions, which must then imply the desired postcondition. With function calls (or loops), to achieve modular verification, we need a frame rule.

We now describe how to obtain a the strongest postcondition transform as in [11]. It suffices to consider only the four heap-manipulating primitives.

Proposition 3.1 (Strongest Postcondition). *In the following Hoare-triples, the postcondition shown is the strongest postcondition of the primitive heap operation with respect to a precondition ϕ.*

$$\begin{array}{ll}
\{\ \phi\ \}\ x = \mathbf{malloc}(1)\ \{\ \mathsf{alloc}(\phi, x)\ \} & (\textit{Heap allocation}) \\
\{\ \phi\ \}\ \mathbf{free}(x)\ \{\ \mathsf{free}(\phi, x)\ \} & (\textit{Heap deallocation}) \\
\{\ \phi\ \}\ x = *y\ \{\ \mathsf{access}(\phi, y, x)\ \} & (\textit{Heap access}) \\
\{\ \phi\ \}\ \ *x = y\ \{\ \mathsf{assign}(\phi, x, y)\ \} & (\textit{Heap assignment})
\end{array}$$

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

$$\begin{array}{lll}
\mathsf{alloc}(\phi, x) & \stackrel{\text{def}}{=} & \mathcal{M} \doteq (x \mapsto v) * \mathcal{H} \wedge \phi[\mathcal{H}/\mathcal{M}, v_1/x] \\
\mathsf{free}(\phi, x) & \stackrel{\text{def}}{=} & \mathcal{H} \doteq (x \mapsto v) * \mathcal{M} \wedge \phi[\mathcal{H}/\mathcal{M}] \\
\mathsf{access}(\phi, y, x) & \stackrel{\text{def}}{=} & \mathcal{M} \doteq (y \mapsto x) * \mathcal{H} \wedge \phi[v/x] \\
\mathsf{assign}(\phi, x, y) & \stackrel{\text{def}}{=} & \mathcal{M} \doteq (x \mapsto y) * \mathcal{H}_1 \wedge \\
 & & \mathcal{H} \doteq (x \mapsto v) * \mathcal{H}_1 \wedge \phi[\mathcal{H}/\mathcal{M}]
\end{array}$$

where $\mathcal{H}$ and $\mathcal{H}_1$ are fresh *heap variables, and v and v_1 are fresh value variables. The notation $\phi[x/y]$ means formula ϕ with variable x substituted for y.* □

We will demonstrate the usefulness (and partly the correctness) of Proposition 3.1 with a simple example. Consider:

$$\{\mathcal{H}_{99} \doteq \mathcal{M}\}\ \texttt{*x += 1; *x -= 1;}\ \{\mathcal{H}_{99} \doteq \mathcal{M}\}$$

In other words, the heap is unchanged after an increment and then a decrement. We rewrite the program so that only one heap operation is performed per program statement; in Fig. 3 we show the rewritten program fragment together with the propagation of the formulas. (For brevity, we also perform a simplification step.) It is then easy to show that the final formula implies $\mathcal{H}_{99} \doteq \mathcal{M}$, by first establishing that $\mathcal{H}_1 \doteq \mathcal{H}_3 \doteq \mathcal{H}_4$ and $v = t_2 = t_1 + 1$. This example provides a program *summary* that the heap is the same before and after execution.

4 THE FRAME RULE

Recall the classic frame rule (CFR) from Section 1 where from $\{\phi\}\ P\ \{\psi\}$ we may infer $\{\phi \wedge \pi\}\ P\ \{\psi \wedge \pi\}$ with the side condition that P does not modify any free variable in π. In

```
{ $\mathcal{H}_{99} \bumpeq \mathcal{M}$ }
  t$_1$ = *x;
{ $\mathcal{M} \bumpeq (x \mapsto t_1)$ * $\mathcal{H}_1$, $\mathcal{H}_{99} \bumpeq \mathcal{M}$ }
  *x = t$_1$ + 1;
{ $\mathcal{M} \bumpeq (x \mapsto t_1 + 1)$ * $\mathcal{H}_1$, $\mathcal{H}_2 \bumpeq (x \mapsto t_1)$ * $\mathcal{H}_1$,
 $\mathcal{H}_2 \bumpeq (x \mapsto t_1)$ * $\mathcal{H}_1$, $\mathcal{H}_{99} \bumpeq \mathcal{H}_2$ }
  $\Downarrow$ // (simplification)
{ $\mathcal{M} \bumpeq (x \mapsto t_1 + 1)$ * $\mathcal{H}_1$, $\mathcal{H}_{99} \bumpeq (x \mapsto t_1)$ * $\mathcal{H}_1$ }
  t$_2$ = *x;
{ $\mathcal{M} \bumpeq (x \mapsto t_2)$ * $\mathcal{H}_3$, $\mathcal{M} \bumpeq (x \mapsto t_1 + 1)$ * $\mathcal{H}_1$,
 $\mathcal{H}_{99} \bumpeq (x \mapsto t_1)$ * $\mathcal{H}_1$ }
  *x = t$_2$ - 1;
{ $\mathcal{M} \bumpeq (x \mapsto t_2 - 1)$ * $\mathcal{H}_4$, $\mathcal{H}_5 \bumpeq (x \mapsto v)$ * $\mathcal{H}_4$,
 $\mathcal{H}_5 \bumpeq (x \mapsto t_2)$ * $\mathcal{H}_3$, $\mathcal{H}_5 \bumpeq (x \mapsto t_1 + 1)$ * $\mathcal{H}_1$,
 $\mathcal{H}_{99} \bumpeq (x \mapsto t_1)$ * $\mathcal{H}_1$ }
```

Fig. 3. Demonstrating Symbolic Execution

our current setting where P now may contain heap references, this frame rule in fact *still* can be used if π only contains free heap variables that are *ghost*. However, because the global heap memory can in general be changed by P, what *cannot* be framed through with this rule, is the property that a ghost variable $\mathcal{H}$ is consistent with the global heap memory $\mathcal{M}$, i.e., $\mathcal{H} \sqsubseteq \mathcal{M}$. We call such a property the "heap reality" of $\mathcal{H}$.

In Separation Logic, where heaps are of the main interest, a key step is that when a program fragment is "enclosed" in some heap, then any formula π whose "footprint" is separate from this heap can be framed through the program. Recall the SL frame rule (SFR) from Section 1 wherein the premise $\{\phi\}\ P\ \{\psi\}$ ensures that the implicit heap arising from the formula ϕ captures all the heap accesses, read or write, in the program fragment P. Therefore $\{\phi * \pi\}\ P\ \{\psi * \pi\}$ naturally follows.

In our setting of explicit heaps, the frame rule, suitably translated into this language, is simply not valid (without some additional machinery ensuring enclosure). The concept of enclosure is to have an explicit subheap (or a collection of subheaps) which contains the program heap updates. These updates are defined to be the cells that the program *writes to*, or *deallocates*. This is because the property $\mathcal{H} \sqsubseteq \mathcal{M}$, where $\mathcal{H}$ is a ghost variable, is falsified *just in case* the program has written to or deallocated some cell in $\mathcal{M}$ whose address is also in $dom(\mathcal{H})$. Thus, the heap reality of $\mathcal{H}$ is lost. Note that **malloc** changes $\mathcal{M}$, but it does not affect cells that are already in $\mathcal{M}$.

Definition 4.1 (Heap Update). Given an address value v, a *heap update* to location v is defined as a statement that either *writes to* or *deallocates* the location v. □

Before formalizing our notion of "enclosure", however, we first need a concept of heap "evolution". Let us use the notation $\tilde{\mathcal{H}}$ to denote the union $\bigcup_i \mathcal{H}_i$ of a collection of subheaps $\mathcal{H}_1, \cdots, \mathcal{H}_n$, $n \geq 2$. Thus for example, $\tilde{\mathcal{H}} \sqsubseteq \mathcal{M}$ simply abbreviates $\mathcal{H}_1 \sqsubseteq \mathcal{M} \wedge \cdots \wedge \mathcal{H}_n \sqsubseteq \mathcal{M}$.

Definition 4.2 (Evolution). Given a valid triple $\{\phi\}\ P\ \{\psi\}$, we say that a collection $\tilde{\mathcal{H}}$ in ϕ, where $\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M}$, *evolves* to a collection $\tilde{\mathcal{H}}'$ in ψ, where $\psi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M}$, if for each model $\mathcal{I}$ of ϕ, executing P from $\mathcal{I}$ will result in $\mathcal{I}'$, such that for any (address) value v, $v \in (dom(\mathcal{I}(\mathcal{M}))\ \setminus\ dom(\mathcal{I}(\tilde{\mathcal{H}})))$ implies $v \notin dom(\mathcal{I}'(\tilde{\mathcal{H}}'))$.

We shall use the notation $\{\phi\}\ P\ \{\psi\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')$ to denote such evolution. □

Intuitively, $\{\phi\}\ P\ \{\psi\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')$ means that the largest $\tilde{\mathcal{H}}'$ can be is $\tilde{\mathcal{H}}$ plus any new cells allocated by P, and minus any that are freed by P. Note also that because the triple is valid, $\mathcal{I}'$ will be a model of ψ. One important usage of the evolution concept is as follows: any heap $\mathcal{H}_i$ such that $\mathcal{H}_i * \tilde{\mathcal{H}}$ and $\mathcal{H}_i \sqsubseteq \mathcal{M}$ at the point of the precondition ϕ (i.e., before P is executed), $\mathcal{H}_i$ will be separate from $\tilde{\mathcal{H}}'$ at the point of the postcondition (i.e., after P is executed).

Consider the **struct** `node` defined in Section 2 and the triple shown below.

```
{ list(H1, x), H1 ⊑ M }
   z = malloc(sizeof(struct node));
   z->next = x;
{ list(H1', z), H1' ⊑ M }
```

We say that $\mathcal{H}_1'$ is an *evolution* of $\mathcal{H}_1$, or $\texttt{evolve}(\mathcal{H}_1, \mathcal{H}_1')$, notationally. Now assume that the triple represents only a local proof (i.e., we are also interested in other parts of $\mathcal{M}$). How should we compose this local triple to obtain a new triple? Formally, we have the following:

$$\frac{\{\phi\}\ P\ \{\psi\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')}{\{\phi\ \wedge\ \tilde{\mathcal{H}} * \mathcal{H}_0\ \wedge\ \mathcal{H}_0 \sqsubseteq \mathcal{M}\}\ P\ \{\psi\ \wedge\ \tilde{\mathcal{H}}' * \mathcal{H}_0\}} \qquad \text{(EV)}$$

Theorem 4.3 (Propagation of Separation). *The rule* (EV) *is correct.* □

Proof Sketch 1. *Let $\mathcal{I}$ be a model of ϕ that is also a model of $\tilde{\mathcal{H}} * \mathcal{H}_0\ \wedge\ \mathcal{H}_0 \sqsubseteq \mathcal{M}$. Let $\mathcal{I}'$ be the result of executing P from $\mathcal{I}$. For each address $v \in dom(\mathcal{I}'(\mathcal{H}_0))$, because $\mathcal{H}_0$ is a ghost variable, i.e., its domain is not affected by executing P, we also have $v \in dom(\mathcal{I}(\mathcal{H}_0))$. It follows that $v \in (dom(\mathcal{I}(\mathcal{M}))\ \setminus\ dom(\mathcal{I}(\tilde{\mathcal{H}})))$. Directly from the definition of evolution, we deduce $v \notin dom(\mathcal{I}'(\tilde{\mathcal{H}}'))$ must hold. As a result, $\mathcal{I}'$ also satisfies $\tilde{\mathcal{H}}' * \mathcal{H}_0$.* □

We are now ready to describe our notion of enclosure. We wish to describe, given a program P and a heap collection $\tilde{\mathcal{H}}$ in a precondition description ϕ, that all heap updates (heap assignments or deallocations) in P, are confined to an evolution of $\tilde{\mathcal{H}}$. The following definition, intuitively, is about one aspect of memory-safety: the heap updates are safe.

Definition 4.4 (Enclose). Suppose we have a valid triple $T = \{\phi\}\ P\ \{\_\}$, $\tilde{\mathcal{H}}$ appears in ϕ, and that $\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M}$. We say $\tilde{\mathcal{H}}$ *encloses* all heap updates of P if for any model $\mathcal{I}$ of ϕ and for any execution path of P of the form $P_1; s; P_2$ where s is a heap update to a location v, it follows that there exists $\tilde{\mathcal{H}}'$ s.t. $\{\phi\}\ P_1\ \{\_\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')$ and $v \in dom(\mathcal{I}'(\tilde{\mathcal{H}}'))$ hold, where $\mathcal{I}'$ is the result of executing P_1 from $\mathcal{I}$.

We shall use the notation $T \rightsquigarrow \texttt{enclose}(\tilde{\mathcal{H}})$ to denote that $\tilde{\mathcal{H}}$ encloses all the updates of P wrt. T. □

We now can introduce our frame rule. It is in fact all about "preserving the heap reality". Recall that a recursive constraint, which satisfies the standard side condition and of which the heap variables are all ghost (and this is a common situation), *remains true* from precondition to postcondition. What may no longer hold in the postcondition is the heap reality of some $\mathcal{H}_0$. That is, $\mathcal{H}_0 \sqsubseteq \mathcal{M}$ may hold at the precondition, but no longer so at the postcondition.

In other words, given local reasoning for a code fragment P and the fact that $\mathcal{H}_0 \sqsubseteq \mathcal{M}$ holds before executing P, how would we preserve this heap reality, without the need to reconsider the code fragment P? Our answer is the following Hoare-style rule, our new frame rule:

$$\frac{\{\phi\}\ P\ \{\psi\} \leadsto \texttt{enclose}(\tilde{\mathcal{H}})}{\{\phi\ \wedge\ \tilde{\mathcal{H}} * \mathcal{H}_0\ \wedge\ \mathcal{H}_0 \sqsubseteq \mathcal{M}\}\ P\ \{\psi\ \wedge\ \mathcal{H}_0 \sqsubseteq \mathcal{M}\}} \qquad (\text{FR})$$

Theorem 4.5 (Frame Rule). *The rule* (FR) *is correct.* □

Proof Sketch 2. *We prove by contradiction. Assume it is not the case, meaning that there is model $\mathcal{I}$ of ϕ that is also a model of $\tilde{\mathcal{H}} * \mathcal{H}_0\ \wedge\ \mathcal{H}_0 \sqsubseteq \mathcal{M}$ and $\mathcal{I}'$ is the result of executing P from $\mathcal{I}$, but $\mathcal{I}'$ does not satisfy $\mathcal{H}_0 \sqsubseteq \mathcal{M}$. Thus there must be a cell $(v \mapsto \_)$ that belongs to $\mathcal{I}'(\mathcal{H}_0)$ but not $\mathcal{I}'(\mathcal{M})$. Because $\mathcal{I}(\mathcal{H}_0) \sqsubseteq \mathcal{I}(\mathcal{M})$, the fragment P must have updated the location v. Therefore, there must be an execution path of P which is of the form $P_1; s; P_2$, where s is a heap update to the location v. Let $\overline{\mathcal{I}}$ be the result of executing P_1 from $\mathcal{I}$. By the definition of enclosure, assume $\{\phi\}\ P_1\ \{\_\} \leadsto \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')$ and $v \in dom(\overline{\mathcal{I}}(\tilde{\mathcal{H}}'))$ hold. By (EV) rule, we have $\overline{\mathcal{I}}$ satisfies $\tilde{\mathcal{H}}' * \mathcal{H}_0$. Since $\mathcal{H}_0$ is a ghost variable, its domain is not affected by executing P_1, i.e., $v \in dom(\overline{\mathcal{I}}(\tilde{\mathcal{H}}_0))$ holds. This is a contradiction.* □

Let us demonstrate the use of the two theorems on a very simple example. Consider the triple:

$$\{((x \mapsto \_)\ *\ \mathcal{H}) \sqsubseteq \mathcal{M}\}\ \texttt{*x = 1;}\ \{((x \mapsto 1)\ *\ \mathcal{H}) \sqsubseteq \mathcal{M}\}$$

We could follow the symbolic execution rules presented in Section 3 and also be able to prove this triple. But, for the sake of discussion, we consider local reasoning over triple T:

$$\{(x \mapsto \_) \sqsubseteq \mathcal{M}\}\ \ *x = 1;\ \ \{(x \mapsto 1) \sqsubseteq \mathcal{M}\}\,,$$

which holds trivially. Also, we can clearly see that both $T \leadsto \texttt{evolve}((x \mapsto \_), (x \mapsto 1))$ and $T \leadsto \texttt{enclose}((x \mapsto \_))$ hold. Applying the rule (EV), we deduce that $(x \mapsto 1) * \mathcal{H}$ holds after executing the code fragment. Furthermore, applying the frame rule, rule (FR), we deduce that $\mathcal{H} \sqsubseteq \mathcal{M}$ remains true, i.e., the heap reality of $\mathcal{H}$ is preserved. Putting the pieces together, we can establish the truth of the original triple by making use of the two theorems.

Recall that we use traditional conjunction, as opposed to separating conjunction in SL. We thus emphasize that all the rules presented above (CFR, EV and FR in particular) can be used in combination because in our framework: $\{\phi\}\ P\ \{\psi_1\}$ and $\{\phi\}\ P\ \{\psi_2\}$ imply $\{\phi\}\ P\ \{\psi_1 \wedge \psi_2\}$.

Our frame rules vs. SL's frame rule: We now elaborate the connection of our two rules (EV) and (FR) with the traditional frame rule in Separation Logic (SL). First, why do we have two rules while SL has one, as introduced in the beginning of this section? The reason is that SL, succinctly, captures *two* important properties: that

- π can be added to precondition ϕ and it *remains true* in the postcondition;
- π *retains its separateness*, from precondition ϕ to postcondition ψ.

$$\underline{[\textbf{MALLOC}]}$$

$$\frac{\begin{array}{c}\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \qquad \psi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M} \\ \psi \models \mathit{dom}(\tilde{\mathcal{H}}') \subseteq \mathit{dom}(\tilde{\mathcal{H}}) \cup \{x\}\end{array}}{\{\phi\}\ \texttt{x = malloc(1)}\ \{\psi\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')}$$

$$\underline{[\textbf{FREE}]}$$

$$\frac{\begin{array}{c}\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \qquad \psi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M} \\ \psi \models \mathit{dom}(\tilde{\mathcal{H}}') \subseteq \mathit{dom}(\tilde{\mathcal{H}}) \setminus \{x\}\end{array}}{\{\phi\}\ \texttt{free(x)}\ \{\psi\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')}$$

$$\underline{[\textbf{OTHER}-\textbf{STATEMENTS}]}$$

$$\frac{\begin{array}{c}\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \qquad \psi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M} \\ \psi \models \mathit{dom}(\tilde{\mathcal{H}}') \subseteq \mathit{dom}(\tilde{\mathcal{H}})\end{array}}{\{\phi\}\ s\ \{\psi\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')}$$

$$\underline{[\textbf{SEQ}-\textbf{COMPOSITION}]}$$

$$\frac{\begin{array}{c}\{\phi\}\ P\ \{\psi\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}') \\ \{\psi\}\ Q\ \{\gamma\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}', \tilde{\mathcal{H}}'')\end{array}}{\{\phi\}\ P;Q\ \{\gamma\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}'')}$$

$$\underline{[\textbf{CALL}]}$$

$$\frac{[\{\phi\}\ \texttt{f()}\ \{\psi\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')] \in \mathit{Specs} \qquad \phi' \models \phi}{\{\phi'\}\ \texttt{call f()}\ \{\_\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')}$$

$$\underline{[\textbf{COMPOSITION}]}$$

$$\frac{\begin{array}{l}\{\phi\}\ P\ \{\psi\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}_1, \tilde{\mathcal{H}}_1') \\ \{\phi\}\ P\ \{\psi\} \rightsquigarrow \texttt{enclose}(\tilde{\mathcal{H}}) \qquad \phi \models \tilde{\mathcal{H}} \ * \ \tilde{\mathcal{H}}_2 \wedge \tilde{\mathcal{H}}_2 \sqsubseteq \mathcal{M}\end{array}}{\{\phi\}\ P\ \{\psi\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}_1 \cup \tilde{\mathcal{H}}_2, \tilde{\mathcal{H}}_1' \cup \tilde{\mathcal{H}}_2)}$$

Fig. 4. Hoare-style Rules for Evolution. OTHER-STATEMENTS applies to statement s not of the kind covered by the rules above.

The second property is important for *successive* uses of the frame rules. Our rule (FR) above only provides for the first property. We accommodate the second property with the other rule (EV), i.e., the "propagation of separation" rule.

The two concepts of evolution and enclosure in fact exist in SL, *implicitly*. Given the triple $T = \{\phi\}\ P\ \{\psi\}$, assume that $\mathcal{H}$ is the heap housing the precondition ϕ and $\mathcal{H}'$ is the heap housing the postcondition ψ. In SL, the frame rule also requires that $T \rightsquigarrow \texttt{evolve}(\mathcal{H}, \mathcal{H}')$ and that $T \rightsquigarrow \texttt{enclose}(\mathcal{H})$. In short, this means that whenever the traditional frame rule in SL[2] is applicable, our frame rules are also applicable without any additional complexity.

[2] We assume an SL fragment without magic wands.

$$\underline{[\textbf{HEAP}-\textbf{ASSIGN}]}$$

$$\frac{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \qquad x \in dom(\tilde{\mathcal{H}})}{\{\phi\}\ \ \texttt{*x = y}\ \ \{\_\} \rightsquigarrow \texttt{enclose}(\tilde{\mathcal{H}})}$$

$$\underline{[\textbf{FREE}]}$$

$$\frac{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \qquad x \in dom(\tilde{\mathcal{H}})}{\{\phi\}\ \ \texttt{free(x)}\ \ \{\_\} \rightsquigarrow \texttt{enclose}(\tilde{\mathcal{H}})}$$

$$\underline{[\textbf{OTHER}-\textbf{STATEMENTS}]}$$

$$\frac{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M}}{\{\phi\}\ \ s\ \ \{\_\} \rightsquigarrow \texttt{enclose}(\tilde{\mathcal{H}})}$$

$$\underline{[\textbf{SEQ}-\textbf{COMPOSITION}]}$$

$$\frac{\begin{array}{l}\{\phi\}\ \ P\ \ \{\psi\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')\\ \{\phi\}\ \ P\ \ \{\psi\} \rightsquigarrow \texttt{enclose}(\tilde{\mathcal{H}}) \qquad \{\psi\}\ \ Q\ \ \{\gamma\} \rightsquigarrow \texttt{enclose}(\tilde{\mathcal{H}}')\end{array}}{\{\phi\}\ \ P;Q\ \ \{\gamma\} \rightsquigarrow \texttt{enclose}(\tilde{\mathcal{H}})}$$

$$\underline{[\textbf{CALL}]}$$

$$\frac{\begin{array}{l}[\{\phi\}\ \ \texttt{f()}\ \ \{\psi\} \rightsquigarrow \texttt{enclose}(\tilde{\mathcal{H}})] \in \mathit{Specs}\\ \phi' \models \phi \wedge \tilde{\mathcal{H}} \sqsubseteq \mathcal{M}\end{array}}{\{\phi'\}\ \ \texttt{call f()}\ \ \{\_\} \rightsquigarrow \texttt{enclose}(\tilde{\mathcal{H}})}$$

$$\underline{[\textbf{COMPOSITION}]}$$

$$\frac{\{\phi\}\ \ P\ \ \{\psi\} \rightsquigarrow \texttt{enclose}(\tilde{\mathcal{H}}) \qquad \phi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M}}{\{\phi\}\ \ P\ \ \{\psi\} \rightsquigarrow \texttt{enclose}(\tilde{\mathcal{H}} \cup \tilde{\mathcal{H}}')}$$

Fig. 5. Hoare-style Rules for Enclosure. OTHER-STATEMENTS applies to statement s not of the kind covered by the rules above.

However, in general our assertion language allows for multiple subheaps, which entails more expressive power, but at the cost that we no longer can resort to the above default. For this paper, we require the specifications to also nominate the subheaps participating in the evolution and/or enclosure relations. Such relations are stated under the keyword **frame**, following the typical **requires** and **ensures** keywords. We demonstrate this in Section 5 when presenting our driving example.

Proving the Evolution and Enclosure relations. The next question of interest is how the evolution and enclosure relations are practically checked. For evolution, we use the rules in Fig. 4. In the rule [CALL],

$$[\{\phi\}\ \texttt{f()}\ \{\psi\} \rightsquigarrow \texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')] \in \mathit{Specs}$$

means that we have nominated $\texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')$ the specifications of function `f`. Similarly for enclosure relation, which can be effectively checked using the rules presented in Fig. 5. Checking evolution and enclosure relations is also performed modularly. Specifically, at call sites, we make use of the rule [**CALL**] and then achieve compositional reasoning with the rule [**COMPOSITION**].

We note that other frameworks (e.g., Separation Logic, Implicit Dynamic Frames) would need a similar mechanism to ensure such "compliance". However, our rules are tailored more towards the flavor of symbolic execution. For example, in a typical implementation, to prove $\texttt{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')$ for a symbolic path, at any point in the path we would track the largest possible subheap $\overline{\mathcal{H}}$ such that $\texttt{evolve}(\tilde{\mathcal{H}}, \overline{\mathcal{H}})$. In the end, the remaining obligation is to prove that $\overline{\mathcal{H}} \sqsubseteq \tilde{\mathcal{H}}'$. For the same reason, our implementation will not suffer from any noticeable degree of non-determinism when dealing with the [**SEQ−COMPOSITION**] rules in Figures 4 and 5.

We finally conclude this section with two Lemmas about the correctness of the rules presented in Figures 4 and 5. The proofs of the two lemmas follow similar (but more tedious) steps as in proving our two main theorems. For brevity, we omit the details.

Lemma 4.6 (Evolution). *Given a valid triple $T = \{\phi\}\ P\ \{\psi\}$ where $\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M}$ and $\psi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M}$, $T \rightsquigarrow \mathit{evolve}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')$ holds if it follows from the rules in Fig. 4.* □

Lemma 4.7 (Enclose). *Given a valid triple $T = \{\phi\}\ P\ \{\_\}$ where $\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M}$, $T \rightsquigarrow \mathit{enclose}(\tilde{\mathcal{H}})$ holds if it follows from the rules in Fig. 5.* □

5 A BREAKTHROUGH EXAMPLE

Reconsider the graph marking example, whose program was presented earlier in Fig. 1. Now, initially the graph is unmarked, and we want to prove that at the end, the graph is fully marked. The definition of `mgraph` in Figure 6 simply states that a graph is fully marked. A node is marked if its `mark` field is 1; otherwise if the value is 0. The parameter t^{in} is of heap type, representing the "history" that includes all the visited nodes — starting off from a root node of interest and an empty history. The usage of a "history" is critical in defining a possibly *cyclic* graph.

There are some subtle but critical points that makes the example extremely challenging. First, note that despite the need for a history in the specification, the program itself *does not implement* any such notion. But without some form of history, *how does the program ensure termination*? The answer is, intuitively, that it uses the `mark` field for termination. Thus one of the central difficulties example is in fact to make a connection between a node's history and its mark.

A second subtlety is this. Though it is obvious that the postcondition must be a fully marked graph, *what is the precondition*? Clearly the program cannot (fully) mark an arbitrary input graph (e.g. it immediately terminates upon encountering a marked node). It is also easy to see that the function should allow an input graph that is "mark successor closed", i.e. any successor node of a marked node is itself already marked. This concept covers both fully unmarked graphs as well as fully marked graphs. However, this intuitively appealing condition is, surprisingly, *too strong*.

```
mgraph($h, x, t^{in}$) :-
    $h \bumpeq \mathsf{emp}, x = \mathsf{null}$.
mgraph($h, x, t^{in}$) :-
    $(x \mapsto (1, \_, \_)) \sqsubseteq t^{in}$, $h \bumpeq \mathsf{emp}$.
mgraph($h, x, t^{in}$) :-
    $h_x \bumpeq (x \mapsto (1, l, r))$, $t_l \bumpeq h_x * t^{in}$,
    mgraph($h_l, l, t_l$), $t_r \bumpeq h_l * t_l$,
    mgraph($h_r, r, t_r$), $h \bumpeq h_x * h_l * h_r$, $h * t^{in}$.

pmg($h, x, t^{in}, t^{out}$) :-
    $h \bumpeq \mathsf{emp}$, $x = \mathsf{null}$, $t^{out} \bumpeq t^{in}$.
pmg($h, x, t^{in}, t^{out}$) :-
     $(x \mapsto (1, \_, \_)) \sqsubseteq t^{in}$, $h \bumpeq \mathsf{emp}$, $t^{out} \bumpeq t^{in}$.
pmg($h, x, t^{in}, t^{out}$) :-
    $h_x \bumpeq (x \mapsto (1, l, r))$,
    $t_l \bumpeq h_x * t^{in}$, mgraph($h_l, l, t_l$),
    $t_r \bumpeq h_l * t_l$, mgraph($h_r, r, t_r$),
    $t_{out} \bumpeq h_r * t_r$, $h \bumpeq h_x * h_l * h_r$, $h * t^{in}$.
pmg($h, x, t^{in}, t^{out}$) :-
    $h_x \bumpeq (x \mapsto (0, l, r))$,
    $t_l^{in} \bumpeq (x \mapsto (1, l, r)) * t^{in}$,
    pmg($h_l, l, t_l^{in}, t_r^{in}$),
    pmg($h_r, r, t_r^{in}, t^{out}$),
    $h \bumpeq h_x * h_l * h_r$, $h * t^{in}$.
```

Fig. 6. Definitions of `mgraph` and `pmg`

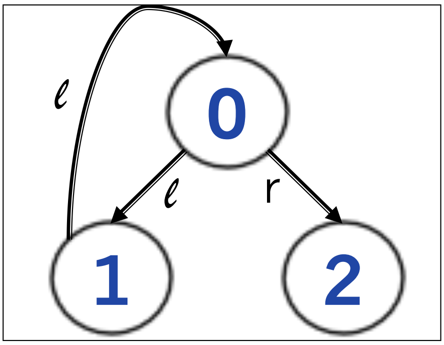

Fig. 7. A Cyclic Graph

Now consider a simple cyclic graph in Fig. 7, assuming that initially all nodes are unmarked and we start the `markgraph` function with the root node 0. We proceed by first marking 0. We then proceed with the first recursive call and mark the node 1. Then from 1, we go back to 0, which has already been marked. But at 0 now, the graph is no longer "marked successor closed", because while 0 has been marked, one of its successors, 2, has not yet been marked.

We can see that the actual precondition is somewhat complicated, because it also acts an *invariant*. Before discussing the precondition, called "properly (partially) marked graph" or `pmg` predicate in Fig. 6, let us now dissect the `markgraph` function more carefully.

There are indeed four scenarios. (1) The function terminates upon seeing a null pointer. The function also terminates upon encountering a marked node. For this there are two possibilities: (2) the current node has been encountered before (in the history); or (3) the subgraph rooted at the current node had already been fully marked (modulo the history). Finally, when encountering an unmarked node (4), the function first marks the node, then invokes two recursive calls to deal with the left and right subgraphs. This last scenario poses a technical challenge, concerning separation of the two recursive calls, so that a frame rule can be used to protect the effects of the first call from the that of the second. In actual fact, the second call can refer to a portion of the heap modified by the first call. The important point however is the second call does not *write* to this subheap.

The four rules in our definition of $\texttt{pmg}(h, x, t^{in}, t^{out})$ correspond to the four scenarios identified above. We address the technical challenges by having: (a) h encloses the *write* footprint of the code while precisely excludes the nodes that had been visited in the history; (b) t^{in} captures the history, i.e. nodes visited starting from the root node to the current node x; and importantly, (c) t^{out} captures the output history, which would be the set of visited nodes right after the function `markgraph` finishes processing the subgraph rooted at x. The use of t^{out} resembles a form of "continuation passing".

The importance of t^{out} can be understood by investigating the 4^{th} scenario identified above. Encountering the node x that is unmarked, the function first marks it before recursively processing the left subgraph and then the right subgraph. What then should be used as the histories for these recursive calls? The history used for the first call can be easily constructed by conjoining the history of the call to x with the updated node x (the mark field has been set). However, the actual history used for the second call very much depends on the shape of the original graph. We choose to construct t^{out} recursively, thus the output history of the first recursive call can be conveniently used as the input history for the second call. The 4^{th} rule in the definition of `pmg` closely follows these intuitions.

Before proceeding, we contrast here our use of the predicate `pmg` with the way predicates are used in SL. In SL, a predicate describes (a part of) the current heap; in `pmg`, we simultaneously describe *three heaps* corresponding to different stages of computation.

In Fig. 8 we show the specification of the function `markgraph` and the proof for the most interesting case: x is not `null` and its `mark` field has not been marked. In the precondition, $\mathcal{H}$, the first component of the definition of `pmg`, appropriately encloses the *write* footprints of the function. It is thus easy to derive, $\textsc{enclose}(\mathcal{H})$. Proving that $\textsc{evolve}(\mathcal{H}, \mathcal{H}')$ is also standard, thus we will not elaborate on this. Instead, let us focus the discussion on how the frame rules are used.

The assertion after step 1 is obtained by unfolding the definition of `pmg` using the fourth rule and instantiating the values of l and r. Note that this unfolding is triggered since the footprint of x is touched. (Using the other rules will lead to a conflict with the guard `assume(x && x->m != 1)`.) At the recursive call `mark(l)` (point 3), we need to prove that the assertion after program point 2 implies the precondition of the function `markgraph`. In this context, the precondition is: $\texttt{pmg}(\mathcal{H}^l, l, t^{in^l}, t^{out^l})\texttt{,}\ \mathcal{H}^l \sqsubseteq \mathcal{M}\texttt{,}\ t^{in^l} \sqsubseteq \mathcal{M}$. Such a proof can be achieved simply by matching $\mathcal{H}^l$ with $\mathcal{H}_l$, t^{in^l} with t_l^{in}, and t^{out^l} with t_r^{in}.

The assertion after this call (step 3) is then obtained by application of framing. First we use the specification to replace the first occurrence of `pmg` by `mgraph`. What we would like to focus on here is the shaded heap formula. First, applying rule (FR), we frame $\mathcal{H}_r * t_l^{in} \sqsubseteq \mathcal{M}$ through the step 3 because the heaps $\mathcal{H}_r$ and t_l^{in} lie outside the updates of the recursive call `markgraph(l)`; note that $\mathcal{H}_l \sqsubseteq \mathcal{M}$, however, no longer holds and is removed. Second, $\mathcal{H}$ evolves into $\mathcal{H}'$, so a heap's separation from $\mathcal{H}$ before the step was propagated into its separation from $\mathcal{H}'$ after the step, shown as the application of rule (EV).

```
requires:  pmg($\mathcal{H}, x, t^{in}, t^{out}$), $\mathcal{H} \sqsubseteq \mathcal{M}$, $t^{in} \sqsubseteq \mathcal{M}$
ensures:   mgraph($\mathcal{H}', x, t^{in}$), $\mathcal{H}' \sqsubseteq \mathcal{M}$, $t^{out} \bumpeq t^{in} * \mathcal{H}'$,
frame:     ENCLOSE($\mathcal{H}$), EVOLVE($\mathcal{H}, \mathcal{H}'$)

void markgraph(struct node *x) {
{ $pmg(\mathcal{H}, x, t^{in}, t^{out})$, $\mathcal{H} \sqsubseteq \mathcal{M}$, $t^{in} \sqsubseteq \mathcal{M}$ }
1  assume(x && x->mark != 1); l = x->left; r = x->right;
{ $\mathcal{H}_x \bumpeq (x \mapsto (0, l, r))$, $t_l^{in} \bumpeq (x \mapsto (1, l, r)) * t^{in}$,
 $pmg(\mathcal{H}_l, l, t_l^{in}, t_r^{in})$, $pmg(\mathcal{H}_r, r, t_r^{in}, t^{out})$,
 $\mathcal{H} \bumpeq \mathcal{H}_x * \mathcal{H}_l * \mathcal{H}_r$, $\mathcal{H} * t^{in}$, $\mathcal{H} \sqsubseteq \mathcal{M}$, $t^{in} \sqsubseteq \mathcal{M}$ }
2  x->mark = 1;
{ $pmg(\mathcal{H}_l, l, t_l^{in}, t_r^{in})$, $t_l^{in} \bumpeq (x \mapsto (1, l, r)) * t^{in}$, $\mathcal{H}_l * \mathcal{H}_r * t_l^{in} \sqsubseteq \mathcal{M}$, $pmg(\mathcal{H}_r, r, t_r^{in}, t^{out})$ }
3  markgraph(l);
{ $mgraph(\mathcal{H}_l', l, t_l^{in})$, $\mathcal{H}_l' \sqsubseteq \mathcal{M}$, $t_r^{in} \bumpeq t_l^{in} * \mathcal{H}_l'$,   // postcondition
 $t_l^{in} \bumpeq (x \mapsto (1, l, r)) * t^{in}$, $pmg(\mathcal{H}_r, r, t_r^{in}, t^{out})$,   // (CFR)
 $\mathcal{H}_l' * \mathcal{H}_r * t_l^{in}$,   // (EV)
 $\mathcal{H}_r * t_l^{in} \sqsubseteq \mathcal{M}$ } // (FR)
4  markgraph(r);
{ $mgraph(\mathcal{H}_l', l, t_l^{in})$, $t_r^{in} \bumpeq t_l^{in} * \mathcal{H}_l'$, $t_l^{in} \bumpeq (x \mapsto (1, l, r)) * t^{in}$,   // (CFR)
 $mgraph(\mathcal{H}_r', r, t_r^{in})$, $\mathcal{H}_r' \sqsubseteq \mathcal{M}$, $t^{out} \bumpeq t_r^{in} * \mathcal{H}_r'$, // postcondition
 $\mathcal{H}_r' * \mathcal{H}_l' * t_l^{in}$,   // (EV)
 $\mathcal{H}_l' \sqsubseteq \mathcal{M}$, $t_l^{in} \sqsubseteq \mathcal{M}$ }  // (FR)
}
```

Fig. 8. Mark Graph Example

This explanation is easily adapted for the call at program point 4. Finally, the postcondition is proved by unfolding mgraph($\mathcal{H}', x, t^{in}$) using the third rule, followed by appropriate variable matching.

In our graph marking example, our “invariant” precondition involves the predicate pmg while the final postcondition involves the predicate mgraph. The fact that pmg resembles the code is coincidental but unsurprising, since it needs to describe the subheaps relevant to the two recursive calls. One might argue that the top-level specification mgraph is contrived so as to be similar to pmg. One could notice that the former definition is “left-askew”, as the “history” used for the right subgraph is computed by conjoining the footprint and the history of the left subgraph If instead we had used a “right-askew” definition, the final entailment may become very hard to prove. In the end, this paper is ultimately about automation, and not about how we can hide implementation details and use highly declarative specifications.

Remark: There are two published proofs which deserve some mention in comparison, even though they are not dealing with recursive predicates. An important one is in [24] which considered the *same* graph marking algorithm. The critical difference is that their method precondition does not require that the input graph to be “properly marked”. This means that the final graph might not be completely marked. Therefore, their postcondition *cannot imply* that the final graph is completely marked. The crucial point here is that the proof in [24] does not prove the same thing as we do. As an aside, the proof is not about local reasoning; it does not use framing at all. Indeed, the specification even refers to addresses *outside* its code footprint. The dynamic frame of the method, *and* those of its sub-methods, are all the same: it represents the one global graph.

The second published proof [20] is about the Schorr-Waite algorithm. However, the program considered is completely different: it comprises a *single* non-recursive function and so it has just one dynamic frame. Hence the proof is not concerned about the two technical points we are so concerned with: that the input graph is "properly marked", allowing for a mark-successor-closed graph, and the intricate frame reasoning when dealing with two successive calls.

6 A PROTOTYPE IMPLEMENTATION

We implemented a prototype in CLP($\mathcal{R}$) [16], submitted as supplementary material for this paper. We used an Intel 2.3 GHz machine running Linux (Ubuntu 14.04.3 LTS), with 4GB memory. Results appear in Table 1.

- We assume that function specifications are given, and loops are compiled into tail-recursive functions[3].
- For each function, we prove one symbolic path at a time. A program is first converted into transitions of three types according to statement types: (a) those which access/manipulate only the *stack* memory, (b) heap-manipulating statements identified in Section 2, and (c) function calls. For (a), standard symbolic execution is assumed to be well-understood. For (b), i.e. basic heap-manipulating statements, symbolic execution rules presented in Proposition 3.1 are used.
- At function calls, the frame rules in Section 4 are employed to achieve compositional reasoning.

The rules in Fig. 4 and Fig. 5 are incorporated into our verification framework and work in tandem with our symbolic execution and frame rules.

The remaining task is to discharge proofs of *entailments* between recursive definitions at call sites and at the end of a function. To demonstrate full automation, our prototype adapted an entailment check procedure from [8, 28]. There they use a general strategy of unfolding a predicate in both the premise and conclusion until the entailment becomes obvious; [9] describes this strategy as "unfold-and-match" (U+M) and we will follow this terminology. In particular:

- We unfold a recursive constraint on a pointer x when its "footprint" (e.g., `x->next`) is touched by the code [28]. This step is performed during symbolic execution.
- At a call site or the end of a function, we deal with obligations of the form $\mathcal{L} \models \mathcal{R}$, performing a sequence of left unfolds (unfolding $\mathcal{L}$) and/or right unfolds (unfolding $\mathcal{R}$) until the proof obligation is simple enough such that a "proof by matching" is successful. At this point, recursive predicates are treated as *uninterpreted.* After dealing with with the heap constraints, an SMT solver – Z3 [10] – can be employed to discharge the obligation.

Note that our entailment check procedure does *not* employ any user-defined lemmas or axioms. Neither does it involve newer technology such as automatic induction [9]. The point here is that our automation is not obtained from a custom theorem-proving method.

[3] For example, we manually translate the example in Fig. 2 to a recursive function, used later as a benchmark program increment in Table 1.

Table 1. Benchmarking Our Prototype Implementation. `# VCs` denotes the number of entailment checks; `# Z3 calls` denotes the number of calls to Z3.

Benchmark Group	Program	Time (s)	# VCs	# Z3 calls
simple	ex1 (Fig. 3)	0.2	1	11
	*buggy-ex1	0.2	1	11
	other examples	0.3	4	(total) 11
sll	append	0.9	3	86
	copy	8.4	3	70
	filter	2.0	5	91
	increment	0.7	3	60
	insert	0.4	1	19
	insert-end	2.1	3	74
	length	0.1	3	25
	*buggy-length	0.2	3	34
	remove	0.1	2	17
	traverse	0.1	1	10
	zero	0.9	3	60
tree	bst-search	1.0	6	87
	isocopy	12.9	4	74
	*buggy-isocopy	0.1	0	10
	traverse	0.5	4	38
graph	markgraph	6.8	6	174
	*buggy-mark1	33.7	1	260
	*buggy-mark2	11.9	3	148

Benchmark Description. To demonstrate the applicability of our framework, other than our breakthrough example and examples presented throughout this paper, we have also selected a number of example programs from the GRASSHOPPER system [27]. As sanity checks, we also introduce a number of buggy variants (prefixed by `*buggy-`) which, as expected, our prototype will fail to verify.

Our benchmarks are in four categories:

- *heap manipulations.* The properties to be proved do not involve recursive constraints.
- *singly-linked lists.* The properties (collectively) involve reasoning about the shape, data, and size of a list.
- *trees.* Programs here traverse a binary and binary search tree. We also have a distinguished example isocopy which has not been verified before in as general a manner.
- The last group is about our driving example: graph marking, and some buggy variants. The purpose here is simply to present some performance metrics.

Proving isomorphic trees. Consider the benchmark isocopy, which is about the classic problem of copying a tree. This program has been previously used by [4] to demonstrate symbolic execution with Separation Logic (SL). However, [4] simply proves that the new tree is separate from the original one; Here we prove a more challenging property, that the copy, also a tree, is *isomorphic* to the original tree. Specifying such property is easy using our framework of explicit heaps, as we can simultaneously describe different heaps corresponding to different stages of computation.

On buggy examples. We have deliberately injected a number of different bugs into originally safe programs. To name a few: wrongly specified "enclosure" heap (*buggy-isocopy), buggy recursive definitions (*buggy-mark2), buggy stack manipulating statements (*buggy-length), and buggy heap-manipulating statements (*buggy-mark1). For these cases, the performance of our verifier can diverge significantly. For most examples, we fail and terminate quickly. Notably, however, for the case of *buggy-mark1, our entailment check procedure exhausts its options without being able to find a successful proof.

7 RELATED WORK AND DISCUSSION

It is possible, but very difficult, to reason in Hoare logic about programs with pointers; [5, 23] explore this direction. The resulting proofs are inelegant and remain too low-level to be widely applicable, let alone being automated.

Separation Logic (SL) [26, 29] was a significant advance with local reasoning via a frame rule, influencing modern verification tools. For example, [3, 7, 14] implement SL-based symbolic execution, as described in [4]. But there was a problem in accommodating data structures with *sharing*.

Bornat et al. [6] present a pioneering SL-based approach for reasoning about data structures with intrinsic sharing. The attempt results in "dauntingly subtle" [6] definitions and verifications. Thus it is unclear how to automate such proofs.

Explicit naming of heaps naturally emerged as extensions of SL [11]. Reynolds [30] conjectured that referring explicitly to the current heap in specifications would allow better handles on data structures with sharing. One major advance of this paper over [11] is in providing a proof method for propagating and reasoning about recursive definitions. More specifically, we now considered *entailments* between such definitions, whereas [11] only considered simple safety properties, which can be translated to the satisfiability problem restricted to non-recursive definitions. But more importantly, it is this current paper that fully realizes Reynolds' conjecture by connecting the explicit subheaps to the global heap ($\mathcal{M}$) with the concept of heap reality and formalizing the concepts of "evolution" and "enclosure". This leads to a new frame rule, and consequently enables local reasoning.

Next consider [13] which addressed sharing (but not automation). Recall the mark function, but now consider its application on a DAG, Fig. 9. The "ramify" rule in [13] would isolate the shaded heap portion 1 and prove that the portion 1 has all been marked. With the help of the *magic wand*, this seems general. Its application, however, is counter-intuitive and hard to automate, because the portion 1 is *artificial*: it

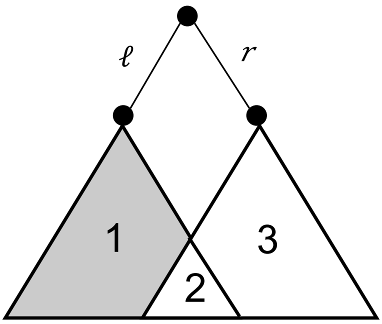

Fig. 9. mark DAG

does not correspond to the actual traversal of the code.

After the development of SL, newer verification frameworks have generally adopted the method of *dynamic frames* [18] (DF), and later, the refinement to *implicit* dynamic frames (IDF) [33]. Some prominent verifiers that use DF/IDF are Vericool [32], Verifast [14], Dafny [20], Chalice [21] and Viper [1]. A dynamic frame is an expression describing a set of addresses. This set is intended to enclose the write footprint of a method[4] or code fragment. These works have the distinct advantage over SL: the code footprint can be defined more precisely and further, *independently* of the specification footprint. (Recall that in SL, the latter is used for the former.)

On the other hand, the use of dynamic frames requires additional machinery to *prove* that the heap updates (by the code) are indeed enclosed by the appropriate dynamic frames. (Whereas, in SL, this is ensured by the logic itself and the accompanied inference rules.) For example, in some verifiers, e.g., Dafny [20], ghost variables are used to explicitly describe the dynamic frame, and the code may be annotated with ghost variable assignments. Correctness then requires that the heap updates are enclosed in the distinguished ghost variable nominated as the dynamic frame of the code. A disadvantage is the added verbosity required on the ghost variable expressions, and the added risk of bugs in matching these expressions against program variable expressions.

IDF approaches, equipped with a new kind of assertion called an *accessibility predicate*, state that heap dereference expressions (whether in assertions or in method bodies) are only allowed if a corresponding permission has already been acquired. This mechanism style allows a method frame to be calculated implicitly from its precondition. In this regard, IDF is similar to the our framework because the accessible addresses can be contrasted with our "enclosing" explicit subheaps. In particular, our notions of "evolution" and "enclosure" are closely related to the concepts of "swinging-pivot" and "self-framing" [17], respectively.

However there remains a general and challenging problem that all works using DF/IDF have not addressed: how to *connect* the code footprint (or dynamic frame) to the specification footprint when these footprints are necessarily recursively-defined. For example, it is notoriously known that many important properties of data structures are in the form of a reachability property, and thus they are difficult to reason about (automatically) without using recursive definitions. It is also known that for a large number of programs that work on data structures, the set of nodes actually modified by a function is a subset of what reachable from an anchor node. Of course such a set is more naturally expressible using recursive definitions. Amongst the state-of-the-art verifiers, only Vericool allows a recursive definition of its dynamic frame, and it is generally accepted that Vericool is not an automated system.

We finally mention here, in our comparison with DF/IDF, that it is not just the case that existing DF/IDF methods have not implemented recursively defined specifications. More to the point is that a specification of a complex data structure requires an *unbounded* number of frames. We have shown that our recursive specifications suffice, but is not the main contribution. Instead, it is that there is a *combination* of reasoning about the specification (again, this must encompass an unbounded number of frames) and reasoning about code.

[4] In this context, we use "method" and "function" interchangeably.

The work [25] shows that by choosing less straightforward definitions of heaps and of heap union in Coq, we can obtain effective reasoning with abstract heap variables, and hence support full separation logic without resulting in excessive proof obligations. As a result, proofs of a number of simple but realistic programs have been successfully mechanized. Similarly, the work [31], which described a mechanized proof of a concurrent in-place spanning tree construction algorithm, bears resemblance to our graph marking example. This is because they traverse via two recursive calls (but they are unconcerned about their relative order). Therefore this work does address the challenge of dealing with the interaction of two recursive calls. Both these works [25, 31] do not address the automation of local reasoning.

We had earlier carefully discussed Separation Logic (SL) and Dynamic Frames (DF). Here we briefly mention some recent work on Region Logic, see e.g. [2]. This work is related to DF: it is essentially a form of Hoare logic for object-based programs. A region, like a dynamic frame, is an expression to describe the footprint of a function. Finally, we also mention [19, 22] that address sharing in the context of shape analysis. In contrast, our current paper focuses on functional verification, thus the ability to perform strongest postcondition propagation is crucial. In this context, automated framing of such assertion formulea is a non-trivial task.

Limitations: We finally remark about the intrinsic limitations of "proof by framing". Consider the following example: a modification of the markgraph example, but instead working on DAGs.

```
void countpath(struct node *x) {
   if (!x) return;
   struct node *l = x->left, *r = x->right;
   x->mark = x->mark + 1;
   countpath(l); countpath(r);
}
```

This program, intuitively, counts the number of "paths" from the root to each node in the DAG. It cannot be verified using our frame rule(s), simply because the sets of cells modified by left and right recursive calls overlap: what established by the first cannot be framed over the subsequent fragment. However, in this case, it is questionable whether "local reasoning" with framing is the way to proceed. (It does not mean that we cannot prove such program using a manual or a non-compositional method.)

8 CONCLUSION

We considered the problem of automatically verifying programs which manipulate complex data structures. These structures, which may exhibit unrestricted sharing including being cyclic, are defined in our specification language which uses recursive definitions. A key feature of our definitions is their use of explicit heaps in order to frame away constituent substructures, in preparation for local reasoning. Because a data structure definition accommodates data structures of arbitrary sizes, it follows that our recursive definition also accommodates the specification of an unbounded number of frames.

Our main contribution is then, given a program which has been annotated with preconditions, postconditions (and loop invariants if needed), a method to:

- generate verification conditions via symbolic execution which realizes strongest postcondition reasoning; and
- discharge the verification conditions by using standard methods of unfolding recursive definitions.

Finally, we finally presented a prototype implementation and demonstrated it over a number of representative programs. In particular, we focused on a graph marking program and presented its first verification by systematic means. An practical outcome of this important example is that its proof provides a template for the formal proof of a class of programs which traverse possibly cyclic data structures.

REFERENCES

[1] http://www.pm.inf.ethz.ch/research/viper.html.

[2] A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, pages 387–411, 2008.

[3] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

[4] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.

[5] R. Bornat. Proving Pointer Programs in Hoare Logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[6] R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation, and aliasing. In *Proc. 2nd workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, 2004.

[7] M. Botinčan, M. Parkinson, and W. Schulte. Separation logic verification of C programs with an SMT solver. *Electronic Notes in Theoretical Computer Science*, 254:5–23, October 2009.

[8] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. In *Science of Computer Programming, 77(9)*, pages 1006–1036, 2012.

[9] D. H. Chu, J. Jaffar, and M. T. Trinh. Automatic induction proofs of data-structures in imperative programs. In *PLDI*, pages 457–466, 2015.

[10] L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *TACAS*, 2008.

[11] G. Duck, J. Jaffar, and Nicolas Koh. Constraint-based program reasoning with heaps and separation. In *CP*, pages 282–298, 2013.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 1969.

[13] A. Hobor and J. Villard. The ramifications of sharing in data structures. In *POPL*, pages 523–536, 2013.

[14] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NFM*, pages 41–55, 2011.

[15] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL*, pages 111–119, 1987.

[16] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.

[17] D. Kassios. Dynamic frames and automated verifcation – a tutorial, 2011.

[18] I. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, pages 268–283, 2006.

[19] O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *CAV*, pages 592–608, 2011.

[20] K. R. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR'10*, 2010.

[21] K. R. M. Leino and P. Mueller. A basis for verifying multi-threaded programs. In *ESOP*, volume 5502, pages 378–393. Springer, 2009.

[22] H. Li, B.-Y. E. Chang, and X. Rival. Shape analysis for unstructured sharing. In *SAS*, pages 90–108, 2015.

[23] J. M. Morris. A general axiom of assignment. assignment and linked data structures. a proof of the schorr-waite algorithm. In *Theoretical Foundations of Programming Methodology*, 1982.

[24] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In *CAV*, 2016.

[25] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *Symposium on Principles of Programming Languages*, January 2010.

[26] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, pages 1–19, 2001.

[27] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using smt. In *CAV*, 2013.

[28] X. K. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242, 2013.

[29] J. C. Reynolds. Separation logic: A logic for shared mutable data objects. In *LICS*, pages 55–74, 2002.

[30] J. C. Reynolds. A short course on separation logic. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/wwwaac2003/aac.html, 2003.

[31] I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87, 2015.

[32] J. Smans, F. Piessens B. Jacobs, and Wolfram Schulte. An automatic verifier for java-like programs based on dynamic frames. In *FASE*, pages 261–275, 2008.

[33] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, pages 148–172, 2009.