The Ramifications of Sharing in Data Structures

Aquinas Hobor
National University of Singapore
hobor@comp.nus.edu.sg

Jules Villard
University College London
j.villard@cs.ucl.ac.uk

Abstract

Programs manipulating mutable data structures with intrinsic sharing present a challenge for modular verification. Deep aliasing inside data structures dramatically complicates reasoning in isolation over parts of these objects because changes to one part of the structure (say, the left child of a dag node) can affect other parts (the right child or some of its descendants) that may point into it. The result is that finding intuitive and compositional proofs of correctness is usually a struggle. We propose a compositional proof system that enables local reasoning in the presence of sharing.

While the AI "frame problem" elegantly captures the reasoning required to verify programs without sharing, we contend that natural reasoning about programs with sharing instead requires an answer to a different and more challenging AI problem, the "ramification problem": reasoning about the indirect consequences of actions. Accordingly, we present a RAMIFY proof rule that attacks the ramification problem head-on and show how to reason with it. Our framework is valid in any separation logic and permits sound compositional and local reasoning in the context of both specified and unspecified sharing. We verify the correctness of a number of examples, including programs that manipulate dags, graphs, and overlaid data structures in nontrivial ways.

Keywords Alias/Pointer, Heap/Shape, Modularity, Separation logic, Specification, Verification.

1. Introduction

Data structures with intrinsic sharing, such as acyclic and unrestricted graphs as well as various kinds of overlaid data structures, are pervasive in computing. An example of an overlaid data structure can be found in the Linux deadline I/O scheduler, in which the set of events forms both a singly linked list and a binary sorted tree, depending on which links one follows. Programs manipulating data structures with sharing are often short, but the reason that they are correct can be subtle, and previous work has not come up with general, intuitive and compositional principles for reasoning about such programs. The key difficulty is that deep aliasing dramatically complicates reasoning in isolation over parts of these objects: changes to one part of the structure (say, the left child of a dag) can affect other parts (the right child or its descendants) that may point into it.

We propose a compositional proof system for programs manipulating shared data structures. Our framework directly addresses the intrinsic sharing present in the data structures and achieves compositionality via applications of the following *ramify rule*:

$$\textsc{Ramify}\quad \frac{\{P\}\ c\ \{Q\} \qquad \mathit{ramify}(R, P, Q, R')}{\{R\}\ c\ \{R'\}}$$

At first glance there seems to be no connection between the known spec $\{P\}\ c\ \{Q\}$ and the desired spec $\{R\}\ c\ \{R'\}$. The connection is given by the *ramification*, indicated by the $\mathit{ramify}(R, P, Q, R')$ premise, which asserts (semantically, although this paper also provides ways to reason logically about it) that the "global" assertion R becomes R' after a "local" transformation from P to Q.

The term "ramification" comes from artificial intelligence [Fin87, Thi01] and refers to the problem of understanding the indirect (global) consequences of (local) actions (*e.g.* relocating a bookcase might reduce the ambient light by blocking the window). Ramification is contrasted with the simpler "frame" problem, which centers on maintaining knowledge after unrelated actions (*e.g.*, relocating the bookcase does not change the number of moons of Jupiter).

Program verification has had significant success handling the frame problem, especially with the frame rule of separation logic [Rey02]:

$$\textsc{Frame}\quad \frac{\{P\}\ c\ \{Q\}}{\{P \star F\}\ c\ \{Q \star F\}}$$

Here the *separating conjunction* $\star$ ensures that P and F cover *disjoint* pieces of heap, allowing the frame rule to guarantee that F is unchanged under the action of c. The frame rule buys us compositionality in the presence of the heap: we can reason about the effect a program has on the portions of heap it accesses, and reuse that spec in any bigger heap. This has given rise to concise, compositional proofs of programs, even in the presence of *some* forms of sharing where one knows *what* is shared *by whom.*

Unfortunately, we usually cannot use the frame rule directly when verifying programs that manipulate data structures with *unrestricted* sharing because such structures cannot easily be massaged into the form $P \star F$: for example, the left and right descendants of a dag node are not usually disjoint. The reason to focus on ramification rather than frame is that the former allows us to reuse specs for c in far more diverse settings than the latter permits. Of course, with great power comes great responsibility: having isolated the parts of the proof that require careful examination of indirect effects on the global structure, we are left with ramification obligations to prove.

As it turns out, ramifications are expressible as separation logic entailments: $\mathit{ramify}(R, P, Q, R') \overset{\text{def}}{=} R \vdash P \star (Q \mathrel{-\!\!\star} R')$. These entailments feature the "magic wand" connective of separation logic ("for all states σ_1 satisfying Q and disjoint from the current state σ_2, the combination of both states $\sigma_1 \oplus \sigma_2$ satisfies R'"), which is notoriously hard to reason about in general given the universal quantification over states. However, appearances of $\mathrel{-\!\!\star}$ in ramifications are restricted to a particular idiom that, together with $\star$, denotes an *update* to the state. Guided by this intuition, we are able to reduce these spatial entailments to more abstract reasoning about the nature of the update on the structure's mathematical representation (*e.g.*, graphs as sets of nodes and edges and transformations on said graphs). The verification process thus divides into two parts: first, showing that a concrete program correctly implements some transformation on an abstract mathematical structure; and second, showing that those mathematical transformations produce the desired specification.

This division gives us the freedom to describe data structures with intrinsic sharing in the most natural way. We will present examples that use the separating conjunction $*$ of separation logic to reason about genuine disjointness (*e.g.*, between the parent of a dag node and its children), the overlapping conjunction $\mathrel{\cup\mkern-12mu*}$ to reason about unspecified sharing (*e.g.*, between the left and right children of a dag node), and the classical conjunction to reason about complete sharing (*e.g.*, an overlaid data structure).

In contrast to previous work, we achieve *compositional* reasoning and *embrace* the sharing. Approaches based on separation logic favored convoluted invariants that hacked the state into the disjoint pieces required by the frame rule. Often the predicate definitions depended heavily on the program at hand (*e.g.* the dag definition used could depend on the order of traversal in the algorithm [BCO04]). In other words, previous attempts to reason about shared data structures with separation logic have stood on their head to avoid the sharing. Other approaches suffered from these problems at least as much and often gave up compositionality altogether [Bor00].

Our key contributions are as follows:

- We present the RAMIFY rule which enables local reasoning while accounting for global effects precisely when they are required. Ramification can reason about programs that manipulate data structures with unrestricted sharing while enabling the small specifications, compositionality, and expressiveness that have led to separation logic's success.
- Although the ramify rule leads to more natural Hoare proofs, the entailment checks can be nontrivial. We have developed a "ramification library" of lemmas that help simplify the ramification conditions. Crucially, we also show how to *prove* ramifications concerned with certain general graph and dag updates in a way that enables a separation of concern between heap manipulations and mathematical reasoning about graphs.
- We have applied the ramify rule to a variety of algorithms that manipulate data structures with nontrivial sharing. Although some of the examples are not long, all involve intricate reasoning due to the heavy use of sharing. We think that a strength of our approach is that the Hoare invariants at each program point are natural and seem to follow our "programmer's intuition" much more closely than traditional proofs.
- We give a semantic account of ramification, and show that RAMIFY and FRAME are each derivable from the other, meaning that our framework is applicable in any separation logic. Moreover, we identify the precise constraints on the underlying model that enable the overlapping conjunction $\mathrel{\cup\mkern-12mu*}$, and show that most separation logics in the literature can therefore follow our recipe to use it to reason about unspecified sharing.

The rest of the paper is organized as follows: we first recall some important concepts from separation logic (§2). We then motivate and present the ramify rule (§3), and show how to reason about it (§4). Based on this, we provide proof sketches for four examples that showcase different aspects of ramification: marking (§5) and copying (§6) a dag, removing from an overlaid data structure (§7), and finally Cheney's garbage collector (§8). Finally, we show how ramification is applicable in virtually any separation logic (§9), compare to related works, and conclude.

Due to space restrictions, we omit most proofs of the semantic facts that support our technique. We refer the interested reader to the companion technical report [HV12].

2. Separation Logic and Trees

Recall the framework of separation logic [IO01, Rey02] while considering the following `mark` procedure, written in C, that recursively marks binary trees, dags, or graphs:

```
1 struct node {int m; struct node *l,*r;};
2 void mark(struct node *x) {
3   if (!x || x->m) return;
4   struct node *l = x->l, *r = x->r;
5   x->m = 1; mark(l); mark(r); }
```

Separation logic allows straightforward inductive definitions of predicates to describe tree-like data structures in the heap. The following definition disregards the actual contents and location of each node, but does make sure that the structure is acyclic (thanks to the $*$ between the root and the subtrees) and that no sharing occurs between subtrees (thanks to the $*$ between the children):

$$\begin{aligned}&\mathsf{tree}(x) \overset{\text{def}}{=}\\ &\quad (x = 0 \wedge \mathsf{emp}) \vee \exists m, l, r.\, x \mapsto m, l, r * \mathsf{tree}(l) * \mathsf{tree}(r)\end{aligned}$$

The definition of tree uses the standard *classical* separation logic operators. A heaplet h satisfies the points-to predicate $x \mapsto y$ when h contains *only* the location x, whose value is y, and the separating conjunction $P * Q$ asserts that P and Q hold on disjoint subheaps. We use $x \mapsto m, l, r$ as a shorthand for $(x + 0) \mapsto m * (x + 1) \mapsto l * (x + 2) \mapsto r$ (simplifying the memory model so that *e.g.*, each datum occupies one unit of space).

It is well-known how to use separation logic to prove the `mark` procedure memory safe for trees. Moreover, the separation logic proof mirrors the programmer's intuitions beautifully. The crux of the verification is to handle the recursive calls via the frame rule, *e.g.*, at line 5, taking the spec of `mark` as a premise:

$$\cfrac{\{\mathsf{tree}(\mathtt{l})\}\ \mathtt{mark(l)}\ \{\mathsf{tree}(\mathtt{l})\}}{\begin{array}{l}\{\mathtt{t} \mapsto 1, \mathtt{l}, \mathtt{r} * \mathsf{tree}(\mathtt{l}) * \mathsf{tree}(\mathtt{r})\}\\ \mathtt{mark(l)}\\ \{\mathtt{t} \mapsto 1, \mathtt{l}, \mathtt{r} * \mathsf{tree}(\mathtt{l}) * \mathsf{tree}(\mathtt{r})\}\end{array}}\ \text{FRAME} \qquad (1)$$

This is a canonical example of how inductive predicates, the separating conjunction, and the frame rule fit together to produce concise proofs. Unrolling the tree predicate yields $*$-conjoined formulas, so the proof system, via its frame rule, is able to perform surgery on the symbolic state and work on each substate independently.

3. Ramifications for Sharing

We now turn to the case of data structures with sharing, and introduce our RAMIFY rule. We begin by defining inductive predicates for dags and graphs before presenting the proof sketch that we aspire to for the `mark` procedure when applied to dags.

3.1 Dag and Graph Predicates

Our first task is to define a dag predicate. Since the separating conjunction $*$ prevents sharing, our first attempt updates tree to utilize regular conjunction $\wedge$ between the children instead:

$$\mathsf{dag}_0(x) \overset{\text{def}}{=} (x = 0 \wedge \mathsf{emp}) \vee \exists l, r.\, x \mapsto l, r * (\mathsf{dag}_0(l) \wedge \mathsf{dag}_0(r))$$

Unfortunately, in *classical* separation logic, $\mathsf{dag}_0(x)$ actually describes a linked list because the conjunction forces the two sub-dags to occupy *exactly* the same space in memory ($h \models P \wedge Q$ if $h \models P$ and $h \models Q$). However, Reynolds points out that dag_0 is correct in *intuitionistic* separation logic, in which $x \mapsto y$ holds on any heap that contains *at least* x, rather than *only* x [Rey02, §6]. Translated into our classical setting this is equivalent to defining dags as follows:

$$\begin{aligned}&\mathsf{dag}_1(x) \overset{\text{def}}{=} (x = 0 \wedge \mathsf{emp}) \vee\\ &\quad \exists l, r.\, x \mapsto l, r * ((\mathsf{dag}_1(l) * \mathsf{true}) \wedge (\mathsf{dag}_1(r) * \mathsf{true}))\end{aligned}$$

If our first attempt was in some sense "too small", then our second is "too big": $\mathsf{dag}_1(x)$ holds on any heap that *at least* contains a dag rooted at x. As usual in intuitionistic separation logic, it is impossible to verify certain algorithms (*e.g.*, disposal) using dag_1.

What we want is a way to get the overlapping features of the intuitionistic conjunction without actually becoming intuitionistic. We turn to another connective, scarcely studied in the published literature, which we dub the *overlapping conjunction* and write $\circledast$, and which precisely characterizes the desired sharing:

$$\begin{aligned} h \models P \circledast Q \overset{\text{def}}{=}\ & \exists h_1, h_2, h_3. \\ & (h_1 \oplus h_2 \oplus h_3 = h) \wedge (h_1 \oplus h_2 \models P) \wedge (h_2 \oplus h_3 \models Q) \end{aligned}$$

The $\oplus$ is the combination operator on the underlying separation algebra [COY07] (often some kind of disjoint union). Contrast the definiton of $\circledast$ with the standard definition of $*$:

$$h \models P * Q \overset{\text{def}}{=} \exists h_1, h_2. (h_1 \oplus h_2 = h) \wedge (h_1 \models P) \wedge (h_2 \models Q)$$

The key point to $\circledast$ is that we can use it in exactly the same places that feature the kinds of sharing that the intuitionistic $\wedge$ captures, but does not "over-approximate" the resulting structure. That is, it allows us to define a classical dag (with a marking field) as:

$$\begin{aligned} \mathsf{dag}(x) \overset{\text{def}}{=}\ & (x = 0 \wedge \mathsf{emp}) \vee \\ & \exists m, l, r.\, x \mapsto m, l, r * (\mathsf{dag}(l) \circledast \mathsf{dag}(r)) \end{aligned} \tag{2}$$

The separating conjunction $*$ between the root x and its children prevents cycles in the data structure. Pleasingly, the definition for graphs simply replaces this remaining $*$ with another $\circledast$:

$$\begin{aligned} \mathsf{graph}(x) \overset{\text{def}}{=}\ & (x = 0 \wedge \mathsf{emp}) \vee \\ & \exists m, l, r.\, x \mapsto m, l, r \circledast (\mathsf{graph}(l) \circledast \mathsf{graph}(r)) \end{aligned} \tag{3}$$

We will equip dag and graph with mathematical dags δ and graphs γ to enable proofs of functional correctness, writing $\mathsf{dag}(x, \delta)$ and $\mathsf{graph}(x, \gamma)$ respectively. We defer the associated formal definitions until §4.2; one key notation is $\delta(x) = (d, l, r)$, which indicates that mathematical node x is associated with data d and successors l and r.

Unspecified Sharing. Observe that $\circledast$ models *unspecified sharing*: *i.e.*, the dag predicate does not say which parts of a dag are shared. In contrast, *specified sharing* requires the precise identification of the shared part, *e.g.* on a dag identifying which nodes are shared between the left and right children; often this is very difficult.

On the other hand, sometimes specified sharing is exactly what the doctor ordered. Although the overlapping conjunction is extremely useful, our framework is not based around it, and one of our key contributions is that RAMIFY can handle both specified and unspecified sharing. For an example of specified sharing, see §7, which uses $\wedge$ instead of $\circledast$; moreover, see §9.3 for how we can use the explicit overlapping conjunction of Cherini and Blanco [CB09].

3.2 Ramifications of Manipulating Dags

Fig. 1 presents the annotated[1] proof sketch of the functional correctness of `mark` when applied to dags using the small spec $\{\mathsf{dag}(\mathtt{l}, \delta)\}$ `mark(l)` $\{\mathsf{dag}(\mathtt{l}, m(\delta, \mathtt{x}))\}$. The function $m(\delta, x)$, whose formal definition is deferred until §5, indicates the mathematical dag derived from δ via marking starting from node x. Notice that this specification immediately implies that if the initial dag is unmarked then the final dag is completely marked.

As is the case for many recursive programs on graph-like data structures, part of the state tracking the recursive exploration of the graph resides in the call stack, which remembers which states have been only partially processed. Our spec accounts for this complexity while remaining local (*i.e.*, it only describes the portion of memory accessed by `mark`), enabling compositional reasoning. Moreover, we enjoy straightforward invariants at each program point.

[1] We often write *e.g.* $\delta(\mathtt{x}) = \ldots$ when what we really mean is $\mathtt{x} \Downarrow v \wedge \delta(v) = \ldots$, where $\mathtt{x} \Downarrow v$ means that the variable $\mathtt{x}$ evaluates to the value v in the current state, because mathematical graphs take values rather than variables. We elide these kinds of details for the presentation.

```
1  void mark(struct node *x) { // {dag(x, δ)}
2    struct node *l,*r;
3    if (x == 0 || x->m == 1) return;
4    l = x->l; r = x->r;
5  // {x ↦ 0,l,r * (dag(l,δ) ⊛ dag(r,δ)) ∧ δ(x) = (0,l,r)}
6    x->m = 1;
7  // {x ↦ 1,l,r * (dag(l,δ) ⊛ dag(r,δ)) ∧ δ(x) = (0,l,r)}
8    mark(l);
9  //↯(15) { x ↦ 1,l,r * (dag(l,m(δ,l)) ⊛ dag(r,m(δ,l))) ∧
           δ(x) = (0,l,r) }
10   mark(r);
11 //↯(16) { x ↦ 1,l,r * (dag(l,δ') ⊛ dag(r,δ')) ∧
           δ(x) = (0,l,r) ∧ δ' = m(m(δ,l),r) }
12 } // {dag(x, m(δ,x))}
```

Figure 1: Proof sketch for marking a binary dag. The steps that induce ramifications are indicated with $\lightning_i$, where the associated ramification entailment is equation number i.

Although the invariants are natural, the proof in separation logic is far from obvious. Things are straightforward enough until we reach the first recursive call at line 8. For tree we applied the frame rule in equation 1, which worked very well. While we can easily frame away the $\mathtt{x} \mapsto 1, \mathtt{l}, \mathtt{r}$ from the precondition (line 7), disentangling the two dag predicates into $\mathsf{dag}(\mathtt{l}, \delta)$ on the one hand and a $*$-disjoint frame on the other would necessitate describing the shape of the right child once everything that is shared with the left child has been removed, which is exactly what we wish to avoid. The second recursive call, in line 10, presents exactly the same problem: we wish to frame but cannot. These two recursive calls require a new proof pattern we call ramification.

3.3 The RAMIFY Rule

While the proof outline of Fig. 1 provides all the invariants needed to prove `mark` on dags, FRAME cannot be applied directly to reason about the effect of applying the `mark` spec on the left child because the left and right child are not disjoint. To solve this issue, we introduce the *ramification* rule, which allows the reasoning to progress through commands that have indirect global effects:

$$\text{RAMIFY}\quad \frac{\{P\}\, c\, \{Q\} \qquad R \vdash P * (Q \mathrel{-\!\!*} R')}{\{R\}\, c\, \{R'\}}\quad \begin{array}{l} \mathit{fv}(Q \mathrel{-\!\!*} R') \cap \\ \mathit{modif}(c) = \varnothing \end{array}$$

RAMIFY isolates the complicated leap in reasoning at each recursive call site so that the assertions at each program point remain natural, such as in Fig. 1 (*e.g.*, the assertions are free from $\mathrel{-\!\!*}$). No free variables of $Q \mathrel{-\!\!*} R'$ may be modified by c. As usual, magic wand (separating implication) is the adjunct[2] of $*$:

$$\sigma \models P \mathrel{-\!\!*} Q \ \overset{\text{def}}{=}\ \forall \sigma'.\, \sigma \perp \sigma' \ \Rightarrow\ \sigma' \models P \Rightarrow \sigma \oplus \sigma' \models Q$$

Here $\sigma \perp \sigma'$ asserts the *compatibility* of σ and σ' ($\exists \sigma''.\, \sigma \oplus \sigma' = \sigma''$). Informally, ramify can be read as "the result of applying c in a state R is R' if replacing P inside R with Q yields R'". Magic wand binds *more loosely* than any other operator.

Ramify is sufficiently abstract that it can be hard to appreciate. As an initial demonstration of its power, observe that the frame rule (modulo some restrictions on free variables as discussed below) is a direct consequence because $P * F \vdash P * (Q \mathrel{-\!\!*} Q * F)$.

Next, let us apply ramification to verify the following spec, in which $\mathtt{x} \mapsto -$ is the standard notation for $\exists x'.\, \mathtt{x} \mapsto x'$.

$$\{\mathtt{x} \mapsto - \circledast \mathtt{y} \mapsto -\}\ \mathtt{*x = a}\ \{\mathtt{x} \mapsto \mathtt{a} \circledast \mathtt{y} \mapsto -\}$$

RAMIFY emits two subgoals. The first precisely matches the standard small axiom for store update in separation logic:

$$\{\mathtt{x} \mapsto -\}\ \mathtt{*x = a}\ \{\mathtt{x} \mapsto \mathtt{a}\}$$

[2] That is, $*$ and $\mathrel{-\!\!*}$ are related by $P * Q \vdash R \Leftrightarrow P \vdash Q \mathrel{-\!\!*} R$.

The second is the following ramification entailment, whose proof is direct from the associated definitions:

$$x \mapsto - \uplus y \mapsto - \vdash x \mapsto - * (x \mapsto a -\!\!* x \mapsto a \uplus y \mapsto -) \quad (4)$$

Free variables. Notice that RAMIFY has a side condition to handle the usual free variable bugaboo. Usually this is no big deal, but it causes trouble when we want to use ramification to verify commands of the form `x =`$f(\ldots)$, since `x` *is* modified and we may want to refer to it in the postcondition. One sufficient solution, which pleasingly removes all free variable side conditions, is to use variables as resource [BCY06], but this introduces other complications. Another solution is to use the following variant of the ramify rule:

RAMIFYASSIGN

$$\frac{\{P\}\ x' = f(\ldots)\ \{Q\} \qquad R \vdash P * (Q -\!\!* [x \mapsto x']R')}{\{R\}\ x = f(\ldots)\ \{R'\}}\ \dagger$$

$$^{\dagger}x' \notin fv(R, R', P) \cup vars(f) \ \wedge\ fv(Q -\!\!* R') \cap modif(f) = \varnothing$$

Observe that RAMIFYASSIGN is a consequence of RAMIFY and the usual rules for assignment and sequence if we are allowed to make the local program transformation from `x =`$f(\ldots)$ to x'`=`$f(\ldots)$`; x =` x' in which x' is always chosen fresh. From now on we will sweep free variable issues under the rug, silently using RAMIFYASSIGN when needed in the verifications.

Lookup. Because points-to facts may be buried inside shared parts of the state, we find it convenient to use the *global* rule for lookup [Rey02] instead of the standard *local* one of separation logic:

LOOKUP

$$\frac{}{\{\exists x'.\, (y \mapsto x' * \mathsf{true}) \wedge [x \mapsto x']P\}\ \mathtt{x\ =\ *y}\ \{P\}}\ x' \notin fv(P, y)$$

In fact, RAMIFYASSIGN is able to derive LOOKUP from the standard local separation logic axiom.

4. Reasoning about Ramifications

To set the stage for the verification of our examples, we now present techniques for general reasoning about ramifications and link abstract mathematical reasoning about graphs to spatial ramifications.

4.1 Ramification Library

Our *ramification library* is a collection of lemmas that help reduce complicated ramifications and related entailments. Some of the more general-purpose lemmas, which can handle simplifications such as removing frames that occur within ramifications, are grouped in Fig. 2. Other lemmas in our library are specific to certain data structures such as graphs; we will meet some of these in §4.3.

Some of the lemmas in Fig. 2 require that various predicates be *precise*, which means that whenever P is satisfied on a substate ($\sigma_1 \leqslant \sigma_3 \overset{\text{def}}{=} \exists \sigma_2.\, \sigma_1 \oplus \sigma_2 = \sigma_3$), that substate must be unique:

$$\begin{aligned} \mathit{precise}(P) \ \overset{\text{def}}{=}\ & \forall \sigma_1, \sigma_2, \sigma_3.\ \sigma_1 \leqslant \sigma_3 \Rightarrow \sigma_2 \leqslant \sigma_3 \Rightarrow \\ & \sigma_1 \models P \Rightarrow \sigma_2 \models P \Rightarrow \sigma_1 = \sigma_2 \end{aligned}$$

Lemmas 4.2 and 4.5 use $-\!\circledast$, the existential magic wand:

$$h \models P -\!\circledast Q \ \overset{\text{def}}{=}\ \exists h'.\, h \perp h' \ \wedge\ h' \models P \ \wedge\ h \oplus h' \models Q$$

$-\!\circledast$, *a.k.a.* "septraction", can be tricky because one does not know *which* copy of P has been pulled out of Q, but is handy sometimes.

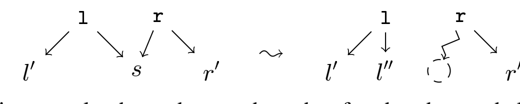
4.2 Exact Graph and Dag Predicates

In this section, we define mathematical graphs γ and dags δ; we will provide ways to reason about ramifications which involve them in the next section. Before we do so, however, let us consider whether our job would be any easier if we were only worried about shape instead of functional correctness, *e.g.* if we tried to verify `mark` with the spec $\{\mathsf{dag}(\mathtt{x})\}$ `mark(x)` $\{\mathsf{dag}(\mathtt{x})\}$.

As in Fig. 1, the proof is straightforward until the first recursive call on line 8. After framing away the root pointer $\mathrm{x} \mapsto 1, \mathrm{l}, \mathrm{r}$, we apply RAMIFY, which emits the following entailment, in which A and B are the pre- and postconditions from the recursive call:

$$\mathsf{dag}(\mathtt{l}) \uplus \mathsf{dag}(\mathtt{r}) \vdash \overbrace{\mathsf{dag}(\mathtt{l})}^{A} * (\overbrace{\mathsf{dag}(\mathtt{l})}^{B} -\!\!* \overbrace{\mathsf{dag}(\mathtt{l}) \uplus \mathsf{dag}(\mathtt{r})}^{C})$$

Unfortunately, this entailment turns out to be invalid. Recall that our ramification $P * (Q -\!\!* R')$ idiom represents a state update, in which P (here, one $\mathsf{dag}(\mathtt{l})$, marked A) is substituted for Q (another $\mathsf{dag}(\mathtt{l})$, marked B). The problem is that the "ramified away" state $(Q -\!\!* R')$ can have dangling pointers into the "local" state P; if P is mangled too badly as it is transformed into Q then those pointers break in the recombined state R' (here $\mathsf{dag}(\mathtt{l}) \uplus \mathsf{dag}(\mathtt{r})$, marked C):



In this example, the update on dag `l` has freed node s and allocated a fresh node l'' instead. Although `l` is still a dag afterwards, `r` is not, so we will not be able to prove $\mathsf{dag}(\mathtt{l}) \uplus \mathsf{dag}(\mathtt{r})$.

This is not an artificial problem stemming from our approach. In fact, the failure of the ramification entailment indicates that $\{\mathsf{dag}(\mathtt{l})\}$ `mark(l)` $\{\mathsf{dag}(\mathtt{l})\}$ is too weak a specification: overly aggressive changes to the pointer structure of the left sub-dag could make the recursive call to the right sub-dag crash, and we must reflect that reality in the specification for `mark`.

There are several solutions to this problem, but for this paper choose the most powerful: proving functional correctness. In §9.3 we will discuss some other possibilities that can yield more lightweight shape proofs at the cost of some additional formalism.

Mathematical graphs. We define the mathematical representation of a directed binary graph as a quadruple (V, D, L, E), where V is a finite set of vertices, D is some set of data, $L : V \to D$ is a labeling function associating each vertex v with some data d, and $E : V \to (V \uplus \{0\}) \times (V \uplus \{0\})$ associates each vertex with up to two successors. To ease the matching between a mathematical graph and its heap representation we usually take $V \subset \mathsf{Loc}$ and $D \subseteq \mathsf{Val}$.

Given a mathematical graph $\gamma = (V, D, L, E)$, we often write $x \in \gamma$ for $x \in V \uplus \{0\}$, $S \subseteq \gamma$ for $S \subseteq V \uplus \{0\}$, and $\gamma(x)$ for $(L(x), E(x).1, E(x).2)$. We define the update of γ at node v, written $[v \mapsto (d, l, r)]\gamma$, where $l, r \in V \cup \{v\} \uplus \{0\}$ and $d \in D$, as:

$$\begin{aligned} & [v \mapsto (d, l, r)](V, D, L, E) \ \overset{\text{def}}{=} \\ & \quad (V \cup \{v\}, D, [v \mapsto d]L, [v \mapsto (l, r)]V) \end{aligned}$$

A node y is the *successor* of a node $x \in \gamma$, written $x \overset{\gamma}{\rightsquigarrow} y$, or simply $x \rightsquigarrow y$ when γ is clear from context, if either $E(x) = (y, z)$ or $E(x) = (z, y)$ for some z. A node y is *reachable* from x, written $x \overset{\gamma}{\rightsquigarrow}^* y$ or $x \rightsquigarrow^* y$, if (x, y) is in the reflexive transitive closure $\rightsquigarrow$. The reachability set of $x \in \gamma$, written $\mathit{reach}(\gamma, x)$, is defined as:

$$\mathit{reach}(\gamma, x) \overset{\text{def}}{=} \{y \mid x \overset{\gamma}{\rightsquigarrow}^* y\}$$

We also lift reachability to sets of vertices $S = \{v_1, \ldots, v_n\} \subseteq V$:

$$\mathit{reach}(\gamma, S) \overset{\text{def}}{=} \mathit{reach}(\gamma, v_1) \cup \cdots \cup \mathit{reach}(\gamma, v_n)$$

Given a graph $\gamma = (V, D, L, E)$ and a set of vertices $S \subseteq V$, it is often useful to restrict γ to those vertices *reachable from* (respectively *not* reachable from) the vertices in S, written $\gamma \downarrow S$ (and respectively $\gamma \uparrow S$). Accordingly, we define (where $f \downharpoonright S$ is the function obtained from f by restricting the domain to the set S):

$$\begin{aligned} \gamma \downarrow S &\overset{\text{def}}{=} (V' = \mathit{reach}(\gamma, S) \qquad, D, L \downharpoonright V', E \downharpoonright V') \\ \gamma \uparrow S &\overset{\text{def}}{=} (V' = V \backslash (\mathit{reach}(\gamma, S)), D, L \downharpoonright V', E \downharpoonright V') \end{aligned}$$

The quadruple $(V', D', L', E') = \gamma \uparrow S$ is not necessarily a graph, since the edge function E' may point outside of the new set of edges V'. However, $\gamma \downarrow S$ is always a graph: the subgraph of γ reachable from S. We sometimes write $\gamma \downarrow x$ and $\gamma \uparrow x$ for $\gamma \downarrow \{x\}$ and $\gamma \uparrow \{x\}$. If $S \nsubseteq V$ then $(V, D, L, E) \downarrow S$ is the empty graph.

4.1: Frame within $\wedge$ Ramification 1

$$\frac{\mathit{precise}(P) \qquad R \vdash P \ast (Q \mathrel{-\!\!\ast} R')}{(P \ast F) \wedge R \vdash P \ast (Q \mathrel{-\!\!\ast} (Q \ast F) \wedge R')}$$

4.2: Frame within $\wedge$ Ramification 2

$$\frac{P \mathrel{-\!\!\circledast} R \vdash Q \mathrel{-\!\!\ast} R'}{(P \ast F) \wedge R \vdash P \ast (Q \mathrel{-\!\!\ast} (Q \ast F) \wedge R')}$$

4.3: Frame within $\cup\!\!\!\!\ast$ Ramification

$$\frac{\mathit{precise}(P, Q) \qquad P \cup\!\!\!\!\ast R \vdash P \ast (Q \mathrel{-\!\!\ast} Q \cup\!\!\!\!\ast R')}{(P \ast F) \cup\!\!\!\!\ast R \vdash P \ast (Q \mathrel{-\!\!\ast} (Q \ast F) \cup\!\!\!\!\ast R')}$$

4.4: Disjoint Ramification

$$\frac{R \vdash P \ast (P' \mathrel{-\!\!\ast} R') \qquad S \vdash Q \ast (Q' \mathrel{-\!\!\ast} S')}{R \ast S \vdash P \ast Q \ast (P' \ast Q' \mathrel{-\!\!\ast} R' \ast S')}$$

4.5: Exact Frame within Ramification

$$\frac{\begin{array}{c}\mathit{precise}(P)\\ R \vdash P \ast F \ast \mathsf{true} \qquad F \mathrel{-\!\!\circledast} R' \vdash F \mathrel{-\!\!\ast} R' \qquad R \vdash P \ast (Q \mathrel{-\!\!\ast} R')\end{array}}{R \vdash P \ast F \ast (Q \ast F \mathrel{-\!\!\ast} R')}$$

4.6: $\cup\!\!\!\!\ast$-piecewise Ramification

$$\frac{\mathit{precise}(P, P') \qquad \forall i.\, P \cup\!\!\!\!\ast Q_i \vdash P \ast (P' \mathrel{-\!\!\ast} P' \cup\!\!\!\!\ast Q'_i)}{P \cup\!\!\!\!\ast Q_1 \cup\!\!\!\!\ast Q_2 \vdash P \ast (P' \mathrel{-\!\!\ast} P' \cup\!\!\!\!\ast Q'_1 \cup\!\!\!\!\ast Q'_2)}$$

Figure 2: Some general-purpose lemmas from our ramification library.

Spatial graphs. We tie a mathematical graph γ to a spatial (in-heap) graph by adding γ as a parameter to graph:

$$\mathsf{graph}(x, \gamma) \stackrel{\text{def}}{=} \begin{array}{l}(x = 0 \wedge \mathsf{emp}) \vee \exists m, l, r.\, \gamma(x) = (m, l, r) \wedge \\ x \mapsto m, l, r \cup\!\!\!\!\ast \mathsf{graph}(l, \gamma) \cup\!\!\!\!\ast \mathsf{graph}(r, \gamma)\end{array}$$

Note that $\mathsf{graph}(x, \gamma)$ "owns" only the spatial representation of the portion of γ that is reachable from x; γ may contain other nodes.

We likewise enrich dag with a mathematical graph δ:

$$\mathsf{dag}(x, \delta) \stackrel{\text{def}}{=} \begin{array}{l}(x = 0 \wedge \mathsf{emp}) \vee \exists m, l, r.\, \delta(x) = (m, l, r) \wedge \\ x \mapsto m, l, r \ast (\mathsf{dag}(l, \delta) \cup\!\!\!\!\ast \mathsf{dag}(r, \delta))\end{array}$$

Moreover, the predicate $\mathsf{dag}(x, \delta)$ is satisfiable *if and only if* $\delta \downarrow x$ is indeed a dag, as enforced by the $\ast$ in the spatial predicate:

Lemma 4.7 *For all graph δ and variable x,*

$$\mathsf{dag}(x, \delta) \dashv\vdash \mathsf{graph}(x, \delta) \wedge (\delta \downarrow x \textit{ is acyclic})$$

Finally, we define the following shorthand for describing multiple sub-graphs of the same graph from a root set $S = \{v_1, \ldots, v_n\}$:

$$\mathsf{graphs}(S, \gamma) \stackrel{\text{def}}{=} \mathsf{graph}(v_1, \gamma) \cup\!\!\!\!\ast \cdots \cup\!\!\!\!\ast \mathsf{graph}(v_n, \gamma)$$
$$\mathsf{dags}(S, \delta) \stackrel{\text{def}}{=} \mathsf{dag}(v_1, \delta) \cup\!\!\!\!\ast \cdots \cup\!\!\!\!\ast \mathsf{dag}(v_n, \delta)$$

If $S = \emptyset$ then both predicates denote emp.

4.3 Reasoning about Graph and Dag Ramifications

One advantage of proving functional correctness is that we can tightly connect our mathematical reasoning with our spatial reasoning. Here we state lemmas that do just that.

First, the spatial graph (and thus dag) predicates are precise.

Lemma 4.8 *For all S and γ, $\mathit{precise}(\mathsf{graphs}(S, \gamma))$.*

Our next lemma lets us *reroot* collections of sub-graphs provided that we preserve the set of reachable nodes:

Lemma 4.9 *If $\mathit{reach}(\gamma, S) = \mathit{reach}(\gamma, S')$, then*

$$\mathsf{graphs}(S, \gamma) \dashv\vdash \mathsf{graphs}(S', \gamma)$$

Our third lemma helps us extend a graph with fresh nodes.

Lemma 4.10 (Graph Growth)

$$x \mapsto d, x, x \vdash \mathsf{graph}(x, [x \mapsto (d, x, x)]) \quad (5)$$
$$x \mapsto d, x, r \ast \mathsf{graph}(r, \gamma) \vdash \mathsf{graph}(x, [x \mapsto (d, x, r)]\gamma) \quad (6)$$
$$x \mapsto d, l, r \ast \mathsf{graphs}(\{l, r\}, \gamma) \vdash \mathsf{graph}(x, [x \mapsto (d, l, r)]\gamma) \quad (7)$$
$$x \mapsto d, l, r \ast \mathsf{dags}(\{l, r\}, \delta) \vdash \mathsf{dag}(x, [x \mapsto (d, l, r)]\delta) \quad (8)$$

First, (5) a graph cell x whose successors are both itself corresponds to a singleton graph[3] $[x \mapsto (d, x, x)]$. Second, (6) if a node x has a loop to itself on the left and a pointer to an existing graph on the right,[4] then we can add x to the graph; not shown is the mirrored case when the loop is on the right. Third, (7) if x links to two (possibly equal) graph nodes then we can again add x to the graph. The first two cases need to be stated separately because $x \notin \gamma$ means that $\mathsf{graph}(x, \gamma)$ is $\bot$. Finally, (8) is the analog of (7) for dags; we do not need analogs for (5) and (6) because dags must be acyclic.

The frame rule, combined with the $\ast$ between a parent and its descendants and equation (8), is enough to mutate the root of a dag. However, an unrestricted graph has $\cup\!\!\!\!\ast$ between the parent and its successors, and so we need to use RAMIFY to update the root. The following lemma helps discharge the associated ramifications:

Lemma 4.11 (Single Graph Node Update)

$$\frac{\gamma(x) = (d, l, r) \qquad \gamma' = [x \mapsto (d', l', r')]\gamma}{\begin{array}{l}\mathsf{graphs}(\{x, l', r'\} \cup S, \gamma) \vdash x \mapsto d, l, r \ast \\ \quad (x \mapsto d', l', r' \mathrel{-\!\!\ast} \mathsf{graphs}(\{x, l, r\} \cup S, \gamma'))\end{array}} \quad (9)$$

$$\frac{\gamma(x) = (d, l, r) \qquad \gamma' = [x \mapsto (d', l, r)]\gamma}{\mathsf{graph}(x, \gamma) \vdash x \mapsto d, l, r \ast (x \mapsto d', l, r \mathrel{-\!\!\ast} \mathsf{graph}(x, \gamma'))} \quad (10)$$

Lemma 4.10 handles the cases in which we are adding a fresh node, so in (9)-(10) we need only consider the case in which $\{x, l, r, l', r'\} \subseteq \gamma$. The case of interest is (9), a full update to node x, where we are updating not only the data d to d' but also the pointers l and r to l' and r' respectively. The precondition is a $\cup\!\!\!\!\ast$-joined set of subgraphs of γ, including x, l', and r', as well as arbitrary others S. After the update, the state contains subgraphs at x (which now contains l' and r') and S, as well as the old l and r (previously contained in the old x), which may now be disconnected from $\{x\} \cup S$. In practice we often care about far simpler updates; (10) is a direct consequence of (9), and handles the case in which we only wish to update the data field.

Next, we observe that an update that preserves the set of reachable nodes cannot remove any overlapping points-to fact. The same remark is true of dags as well, replacing graphs with dags everywhere in the lemma below.

Lemma 4.12 (Points-to preservation)

$$\frac{\mathit{reach}(\gamma', S') \supseteq \mathit{reach}(\gamma, S)}{\begin{array}{l}\mathsf{graphs}(S, \gamma) \cup\!\!\!\!\ast x \mapsto - \vdash \mathsf{graphs}(S, \gamma) \ast \\ \quad (\mathsf{graphs}(S', \gamma') \mathrel{-\!\!\ast} \mathsf{graphs}(S', \gamma') \cup\!\!\!\!\ast x \mapsto -)\end{array}}$$

Our final lemma applies when we wish to update an entire subgraph (typically with a function call) rather than a single node.

Lemma 4.13 (Subgraph Update)

$$\frac{\mathit{reach}(\gamma', S'_1) \supseteq \mathit{reach}(\gamma, S_1) \qquad \gamma' \uparrow S'_1 = \gamma \uparrow S_1}{\begin{array}{l}\mathsf{graphs}(S_1, \gamma) \cup\!\!\!\!\ast \mathsf{graphs}(S_2, \gamma) \vdash \mathsf{graphs}(S_1, \gamma) \ast \\ \quad (\mathsf{graphs}(S'_1, \gamma') \mathrel{-\!\!\ast} \mathsf{graphs}(S'_1, \gamma') \cup\!\!\!\!\ast \mathsf{graphs}(S_2, \gamma'))\end{array}} \quad (11)$$

$$\frac{\mathit{reach}(\delta', S'_1) \supseteq \mathit{reach}(\delta, S_1) \qquad \delta' \uparrow S'_1 = \delta \uparrow S_1}{\begin{array}{l}\mathsf{dags}(S_1, \delta) \cup\!\!\!\!\ast \mathsf{dags}(S_2, \delta) \vdash \mathsf{dags}(S_1, \delta) \ast \\ \quad (\mathsf{dags}(S'_1, \delta') \mathrel{-\!\!\ast} \mathsf{dags}(S'_1, \delta') \cup\!\!\!\!\ast \mathsf{dags}(S_2, \delta'))\end{array}} \quad (12)$$

[3] $[x \mapsto (d, x, x)] \stackrel{\text{def}}{=} (\{x\}, D, [x \mapsto d], [x \mapsto (x, x)])$

[4] Observe that r can be equal to 0, in which case $\mathsf{graph}(r, \gamma)$ is just emp.

First, (11) lets us ramify an update to a subgraph (or set of subgraphs) as long as all previously reachable nodes are still reachable (to prevent *e.g.* the dangling pointer problem outlined in §3.3) and the mathematical update is local. Second, (12) gives us the same property for dags (if our newly substituted sub-dag does not contain a cycle than our whole dag will not suddenly become cyclic).

5. Proving `mark` on Dags

We are ready at last to polish off the proof of `mark` from Fig. 1.

Mathematical marking. One of our goals is to translate mathematical reasoning into spatial reasoning. Define the mathematical marking $m(\gamma, r)$ of a graph $\gamma = (V, D, L, E)$ starting from the vertex $r \in V$ as marking all nodes reachable via *unmarked nodes* from r. Formally, define a new relation $\overset{\gamma}{\rightsquigarrow}_0$ as follows:

$$x \overset{\gamma}{\rightsquigarrow}_0 y \text{ iff } \exists z.\, \gamma(x) = (0, y, z) \vee \gamma(x) = (0, z, y)$$

As before, we omit the subscript γ when it is clear from context and write $\overset{\gamma}{\rightsquigarrow}{}_0^*$ for the reflexive transitive closure. The marking $m(\gamma, r)$ of γ from r is then (V, D, L', E) where, for all $x \in V$,

$$L'(x) \quad = \quad \begin{cases} 1 & \text{if } r \rightsquigarrow_0^* x \\ L(x) & \text{otherwise} \end{cases}$$

We also need to describe the effect of marking a single node in γ, accomplished with $m_1(\gamma, x)$, that sets the marked bit of node x in γ to 1. The following lemma about mathematical markings now becomes crucial to prove the functional correctness of `mark`.

Lemma 5.1 *For all graph γ and nodes $x, y \in \gamma$,*

$$m(m(\gamma, x), y) \quad = \quad m(m(\gamma, y), x) \tag{13}$$

Moreover, if $\gamma(x) = (m, l, r)$, then

$$m(m(m_1(\gamma, x), l), r) = m(m_1(m(\gamma, l), x), r) =$$
$$m_1(m(m(\gamma, l), r), x) = m(m_1(m(\gamma, r), x), l) = m(\gamma, x) \tag{14}$$

That is, (13) we can swap the order of two mathematical markings, and (14) regardless of which order we mark the root and children (either child first by equation 13), at the end we are fully marked.

Spatial marking. Our first remaining tasks are the ramifications on lines 9 and 11. In both cases we frame away the root node and then apply RAMIFY, yielding the following entailments:

$$\begin{aligned} \mathsf{dag}(\mathtt{l}, \delta) \circledast \mathsf{dag}(\mathtt{r}, \delta) \vdash \mathsf{dag}(\mathtt{l}, \delta) * (&\mathsf{dag}(\mathtt{l}, m(\delta, \mathtt{l})) \mathrel{-\!\!\ast} \\ &\mathsf{dag}(\mathtt{l}, m(\delta, \mathtt{l})) \circledast \mathsf{dag}(\mathtt{r}, m(\delta, \mathtt{l}))) \end{aligned} \tag{15}$$

$$\begin{aligned} &\mathsf{dag}(\mathtt{l}, m(\delta, \mathtt{l})) \circledast \mathsf{dag}(\mathtt{r}, m(\delta, \mathtt{l})) \\ \vdash\ &\mathsf{dag}(\mathtt{r}, m(\delta, \mathtt{l})) * (\mathsf{dag}(\mathtt{r}, m(m(\delta, \mathtt{l}), \mathtt{r})) \mathrel{-\!\!\ast} \\ &\quad \mathsf{dag}(\mathtt{l}, m(m(\delta, \mathtt{l}), \mathtt{r})) \circledast \mathsf{dag}(\mathtt{r}, m(m(\delta, \mathtt{l}), \mathtt{r}))) \end{aligned} \tag{16}$$

Observe that the first ramification directly implies the second by instantiating δ with $m(\delta, \mathtt{l})$ in the first entailment and using the commutativity of $\circledast$ to swap the roles of `l` and `r`. Since marking a graph does not change either the set of nodes nor the edge function of a graph, Lemma 4.13 can easily prove (15).

Finally, to establish the postcondition in line 12 from line 11, apply Lemma 4.10 to derive $\mathsf{dag}(\mathtt{x}, m_1(m(m(\delta, \mathtt{l}), \mathtt{r}), \mathtt{x}))$, which by Lemma 5.1 is equivalent to our postcondition.

Observations. Our proof of `mark` (*i.e.*, Fig. 1 and §5) is short and our invariants are straightforward. We were able to reuse our initial ramification (15) to prove our second (16). Essentially all of the spatial difficulties were handled by our ramification library. Moreover, by Lemma 5.1 our proof is easy to modify to accommodate trivial changes in the program like moving the update in line 6 to after one or both of the recursive calls in lines 8 and 10, swapping the order of the recursive calls, and so forth. Our ability to accommodate these kinds of changes is an indication of the power of using ramification to separate the mathematical and spatial reasoning from each other. All of these desirable properties are in

```
struct node {struct node *c,*l,*r;};
struct node *
copy_dag(struct node *x) { // $\{\mathsf{icdag}(\mathtt{x}, \delta)\}$
  local l,r,ll,rr,y;
  if (!x) return 0;
  if (x->c) return x->c;
  l = x->l; r = x->r;
  y = malloc(sizeof(struct node));
  x->c = y;
// $\left\{ \begin{array}{l} \mathtt{x} \mapsto_\alpha \mathtt{y}, \mathtt{l}, \mathtt{r} * (\mathsf{icdag}(\mathtt{l}, \delta) \circledast \mathsf{icdag}(\mathtt{r}, \delta)) * \mathtt{y} \mapsto_\beta -, -, - \wedge \\ \delta(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \end{array} \right\}$
  ll = copy_dag(l);
//$\lightning$(20) $\left\{ \begin{array}{l} \mathtt{x} \mapsto_\alpha \mathtt{y}, \mathtt{l}, \mathtt{r} * (\mathsf{ddag}(\mathtt{l}, \mathtt{ll}, \delta') \circledast \mathsf{icdag}(\mathtt{r}, \delta')) * \\ \mathtt{y} \mapsto_\beta -, -, - \wedge \delta(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \delta' \geqslant_{\mathtt{l}} \delta \end{array} \right\}$
  rr = copy_dag(r);
//$\lightning$(21) $\left\{ \begin{array}{l} \mathtt{x} \mapsto_\alpha 0, \mathtt{l}, \mathtt{r} * (\mathsf{ddag}(\mathtt{l}, \mathtt{ll}, \delta'') \circledast \mathsf{ddag}(\mathtt{r}, \mathtt{rr}, \delta'')) * \\ \mathtt{y} \mapsto_\beta -, -, - \wedge \delta(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \delta' \geqslant_{\mathtt{l}} \delta \wedge \delta'' \geqslant_{\mathtt{r}} \delta' \end{array} \right\}$
  y->c = 0; y->l = ll; y->r = rr;
// $\left\{ \begin{array}{l} \mathtt{x} \mapsto_\alpha \mathtt{y}, \mathtt{l}, \mathtt{r} * \mathtt{y} \mapsto_\beta 0, \mathtt{ll}, \mathtt{rr} * \\ (\mathsf{ddag}(\mathtt{l}, \mathtt{ll}, \delta'') \circledast \mathsf{ddag}(\mathtt{r}, \mathtt{rr}, \delta'')) \wedge \\ \delta(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \delta' \geqslant_{\mathtt{l}} \delta \wedge \delta'' \geqslant_{\mathtt{r}} \delta' \end{array} \right\}$
  return y;
} // $\{\mathsf{ddag}(\mathtt{x}, \mathtt{y}, \delta''') \wedge \delta''' \geqslant_{\mathtt{x}} \delta\}$
```

Figure 3: Proof sketch of dag copy

contrast to previous work on verifying these kinds of algorithms (*e.g.*, [BCO04]), which utilized extremely complex and brittle invariants so that they could always apply the frame rule. Verifications utilizing ramifications are both more natural and more robust.

Marking possibly cyclic graphs. The `mark` function can also mark unrestricted graphs. Because Lemmas 4.13 and 5.1 both apply to graphs as well as dags, the only substantial change to the the proof in figure 1 is for line 6. Here dags only require the frame rule due to the $*$ between a parent and its children but unrestricted graphs require an additional ramification due to the additional $\circledast$:

$$\begin{aligned} \mathtt{x} \mapsto 0, \mathtt{l}, \mathtt{r} \circledast \mathsf{graphs}(\{\mathtt{l}, \mathtt{r}\}, \gamma) \vdash\ & \mathtt{x} \mapsto 0, \mathtt{l}, \mathtt{r} * \\ & (\mathtt{x} \mapsto 1, \mathtt{l}, \mathtt{r} \mathrel{-\!\!\ast} \mathtt{x} \mapsto 1, \mathtt{l}, \mathtt{r} \circledast \mathsf{graphs}(\{\mathtt{l}, \mathtt{r}\}, m_1(\gamma, \mathtt{x}))) \end{aligned}$$

This ramification follows directly from Lemma 4.11.

Termination. Our work here is primarily concerned with partial correctness, but suppose we were interested in total correctness as well. The dag argument is simpler: each recursive call is on a strictly smaller subheap thanks to the $*$ between a parent and its children; notice that this argument is valid regardless of whether we mark the root first, at line 6, or after one or both recursive calls. In contrast, the termination argument on unrestricted graphs is more complicated because the $\circledast$ between root and successors means that the subheap may not be any smaller at the recursive calls. Instead, each recursive call must be on a graph with fewer unmarked nodes; if we recurse before coloring the root then we may not terminate.

6. Copying Dags

Let us now demonstrate that ramification can apply equally well to programs that, unlike `mark`, mutate the link structure of the graph, and turn our attention to the more involved program in Fig. 3 that makes a deep (structure-preserving) copy of a dag. To keep things simple we will use the data field of each node in the original dag to record the location of its copy (or 0 if the node has not yet been copied). Initially, all the copy fields of $\mathsf{dag}(\mathtt{x})$ must be set to 0, and at the end all the nodes reachable from `x` will have been copied into a new dag whose root is returned by `copy_dag`. In the intermediate recursive calls, parts of the dag rooted at the argument will have already been copied.

Just as with `mark`, despite the intricate intermediate states of the algorithm, a straightforward recursive implementation is quite compact, and works as follows. If the dag is empty, or the root is already copied, then return immediately. Otherwise, recursively copy the left and right children of the root and allocate a fresh node to be the root's copy, setting the fields as appropriate.

Regions. Our proof of `copy_dag` requires the notion of *regions* [LG88]. Briefly, regions indicate disjoint zones in the heap, and each spatial predicate can be parameterized by a region identifier such as α or β. Predicates in different regions are always disjoint, which we use to keep the original dag disentangled from its copy.

Regions are useful when we are faced with the following problem, in which $\sharp$ is some sharing operator such as $\wedge$ or $\circledast$:

$$(P * Q)\sharp(R * S) \ \overset{?}{\dashv\vdash}\ (P\sharp R) * (Q\sharp S)$$

That is, we have some disjoint formulas P (say, a dag) and Q (its copy) which overlap with two additional disjoint formulas R (a second dag, overlapping with P) and S (the second dag's copy, overlapping with Q), and we wish to shuffle resources around until the originals P and Q are once again overlapping with each other and are disjoint from their associated overlapping copies Q and S.

Observe that the $\dashv$ direction is immediate. Unfortunately, the direction we will need to prove the entailment in equation 24 is $\vdash$, which is not true in general. Regions are exactly what we need because they can ensure that P does not have any overlap with S despite the intermediate sharing operator $\sharp$:

$$(P_\alpha * Q_\beta)\sharp(R_\alpha * S_\beta) \ \dashv\vdash\ (P\sharp R)_\alpha * (Q\sharp S)_\beta \qquad (17)$$

Others have run across the same problem in contexts including RGSep [Vaf07] and shape analysis for overlaid lists and trees [LYP11] and have turned to regions for similar reasons.

Describing completed and in-process dag copies. We represent an entirely copied dag $\delta = (V, D, L, E)$ rooted at x and its copy rooted at y by the predicate $\mathsf{ddag}(x, y, \delta)$ (or *double dag*):

$$\mathsf{ddag}(x, y, \delta) \ \overset{\text{def}}{=}\ \begin{array}{l}(x = y = 0 \wedge \mathsf{emp}) \vee (\mathsf{dag}_\alpha(x, \delta) * \\ \quad \mathsf{dag}_\beta(y, \mathit{copy}(\delta)) \wedge L(x) = y \wedge y \neq 0)\end{array}$$

The nodes in the first dag are described by the graph δ. Because we store the addresses of the copy in the data fields, δ is also enough to describe the copy via $\mathit{copy}(\delta) = (V', D, L', E')$, where

$$\begin{aligned} V' &= \{v' \mid \exists v \in V.\, L(v) = v' \wedge v' \neq 0\} \\ L'(v) &= 0 \\ E'(v) &= (l', r') \text{ if } \begin{cases} \exists v' \in V.\, \delta(v') = (v, l, r) \wedge \\ (l = l' = 0 \vee L(l) = l') \wedge \\ (r = r' = 0 \vee L(r) = r') \end{cases} \end{aligned}$$

The predicate ddag describes the postcondition for `copy_dag`; our next task is to define the precondition. Because some parts of the dag may have already been copied, the *in-copy dag* predicate $\mathsf{icdag}(x, \delta)$ describes a single dag in region α and a *set* of dags in region β corresponding to any previously copied sub-dags.

$$\mathsf{icdag}(x, \delta) \overset{\text{def}}{=} \mathsf{dag}_\alpha(x, \delta) * \mathsf{dags}_\beta(\mathit{croots}(x, \delta), \mathit{copy}(\delta))$$

The $\mathit{croots}(x, \delta)$ function finds the roots of the copied sub-dags:

$$\mathit{croots}(x, \delta) \overset{\text{def}}{=} \begin{cases} \varnothing & \text{if } x = 0 \\ \mathit{croots}(l, \delta) \cup \mathit{croots}(r, \delta) & \text{if } \delta(x) = (0, l, r) \\ \{x\} & \text{otherwise} \end{cases}$$

Observe that when x is copied, *i.e.* $\delta(x) = (y, l, r)$ and $y \neq 0$, then

$$\mathsf{icdags}(x, \delta) \dashv\vdash \mathsf{ddags}(x, y, \delta) \qquad (18)$$

We will use this equivalence to move between the precondition and the postcondition when we discover that the dag is already copied.

When we wish to reason entirely about the copies we write $\mathsf{cdags}(x, \delta)$ (*i.e., copy dags*) for $\mathsf{dags}(\mathit{croots}(x, \delta), \mathit{copy}(\delta))$. Note that if x is not yet copied, *i.e.* $\delta(x) = (0, l, r)$, then, using the second case in the definition of *croots*, we deduce that

$$\mathsf{cdags}(x, \delta) \dashv\vdash \mathsf{cdags}(l, \delta) \circledast \mathsf{cdags}(r, \delta), \text{ and thus}$$

$$\mathsf{icdags}(x, \delta) \dashv\vdash x \mapsto_\alpha 0, l, r * (\mathsf{icdags}(l, \delta) \circledast \mathsf{icdags}(r, \delta)) \ (19)$$

Finally, to reflect the fact that already copied parts of the dag will not be changed by `copy_dag`, we define the relation $\delta' \geqslant_x \delta$ between two dags $\delta = (V, D, L, E)$ and $\delta' = (V, D, L', E)$, true when $\delta' \uparrow x = \delta \uparrow x$ and $\delta' \downarrow x$ is "more copied" than $\delta \downarrow x$:

$$\forall v \in \mathit{reach}(\delta, x).\ \begin{array}{l}(L(v) \neq 0 \Rightarrow L'(v) = L(v)) \wedge \\ (L'(v) = 0 \Rightarrow L(v) = 0)\end{array}$$

We will write $\delta' \geqslant \delta$ when $\exists x.\, \delta' \geqslant_x \delta$.

Verification of `copy_dag`. Armed with these new predicates, we can annotate the program in Fig. 3 with assertions at key program points to prove the following specification:

$$\{\mathsf{icdag}(x, \delta)\}\ \texttt{y = copy\_dag(x)}\ \{\mathsf{ddag}(x, y, \delta') \wedge \delta' \geqslant_x \delta\}$$

If the dag is empty (line 5) then the postcondition is trivially satisfied. If the node has already been copied (line 6) then equation 18 yields the postcondition. The real meat of the algorithm is in the ramifications from the two recursive call sites and the entailment of the postcondition from line 16. The two ramifications are as follows:

$$\begin{array}{ll} & \mathsf{icdag}(\mathtt{l}, \delta) \circledast \mathsf{icdag}(\mathtt{r}, \delta) \\ \vdash & \mathsf{icdag}(\mathtt{l}, \delta) * (\mathsf{ddag}(\mathtt{l}, \mathtt{ll}, \delta') \wedge \delta' \geqslant_{\mathtt{l}} \delta \mathrel{-\!\!*} \\ & \quad \mathsf{ddag}(\mathtt{l}, \mathtt{ll}, \delta') \circledast \mathsf{icdag}(\mathtt{r}, \delta') \wedge \delta' \geqslant_{\mathtt{l}} \delta) \end{array} \qquad (20)$$

$$\begin{array}{ll} & \mathsf{ddag}(\mathtt{l}, \delta') \circledast \mathsf{icdag}(\mathtt{r}, \delta') \\ \vdash & \mathsf{icdag}(\mathtt{r}, \delta') * (\mathsf{ddag}(\mathtt{r}, \mathtt{rr}, \delta'') \wedge \delta'' \geqslant_{\mathtt{r}} \delta' \mathrel{-\!\!*} \\ & \quad \mathsf{ddag}(\mathtt{l}, \mathtt{ll}, \delta'') \circledast \mathsf{ddag}(\mathtt{r}, \mathtt{rr}, \delta'') \wedge \delta'' \geqslant_{\mathtt{r}} \delta') \end{array} \qquad (21)$$

As with `mark`, the second ramification follows from the first by swapping the roles of `r` and `l` and observing that when $\delta' \geqslant \delta$

$$\mathsf{ddag}(x, y, \delta) \vdash \mathsf{icdag}(x, \delta') \Leftrightarrow \mathsf{ddag}(x, y, \delta') \Leftrightarrow \mathsf{ddag}(x, y, \delta)$$

Regions let us split the first ramification (20) using the Lemma 4.4 from our ramification library, yielding two simpler ramifications in which $\delta' \geqslant_{\mathtt{l}} \delta$, and, by the definition of ddag, $\mathtt{l} = \mathtt{ll} = 0 \vee \delta'(\mathtt{l}) = (\mathtt{ll}, -, -)$. The first half of (20), in region α,

$$\begin{array}{l}\mathsf{dag}(\mathtt{l}, \delta) \circledast \mathsf{dag}(\mathtt{r}, \delta) \ \vdash \mathsf{dag}(\mathtt{l}, \delta) * (\mathsf{dag}(\mathtt{l}, \delta') \mathrel{-\!\!*} \\ \qquad\qquad \mathsf{dag}(\mathtt{l}, \delta') \circledast \mathsf{dag}(\mathtt{r}, \delta')),\end{array} \qquad (22)$$

is direct from Lemma 4.13. The second half of (20), in region β, is

$$\begin{array}{l}\mathsf{cdags}(\mathtt{l}, \delta) \circledast \mathsf{cdags}(\mathtt{r}, \delta) \ \vdash \mathsf{cdags}(\mathtt{l}, \delta) * (\mathsf{dag}(\mathtt{ll}, \delta'_c) \mathrel{-\!\!*} \\ \qquad\qquad \mathsf{dag}(\mathtt{ll}, \delta'_c) \circledast \mathsf{cdags}(\mathtt{r}, \delta'))\end{array} \qquad (23)$$

where $\delta'_c = \mathit{copy}(\delta')$. This ramification is more involved because the copied roots of δ' starting from `r` may differ from the previous ones in δ. Instantiating Lemma 4.13 with $S_1 = \mathit{croots}(\delta, \mathtt{l})$, $S_2 = \mathit{croots}(\delta, \mathtt{r})$ and $S'_1 = \{\mathtt{ll}\}$ yields this entailment, which is only halfway there, because it features the sub-dags rooted at $\mathit{croots}(\delta, \mathtt{r})$, whereas we want those rooted at $\mathit{croots}(\delta', \mathtt{r})$:

$$\begin{array}{l}\mathsf{cdags}(\mathtt{l}, \delta) \circledast \mathsf{cdags}(\mathtt{r}, \delta) \vdash \mathsf{cdags}(\mathtt{l}, \delta) * \\ \quad (\mathsf{dag}(\mathtt{ll}, \delta'_c) \mathrel{-\!\!*} \mathsf{dag}(\mathtt{ll}, \delta'_c) \circledast \mathsf{dags}(\mathit{croots}(\delta, \mathtt{r}), \delta'_c))\end{array}$$

To complete this proof, we remark that the copied roots of `r` in δ' and in δ satisfy the following relations, hence Lemma 4.9 applies:

$$\mathit{croots}(\delta, \mathtt{r}) \subseteq \mathit{reach}(\delta'_c, \mathit{croots}(\delta', \mathtt{r})) \qquad (\delta' \geqslant \delta)$$

$$\mathit{croots}(\delta', \mathtt{r}) \subseteq \mathit{croots}(\delta, \mathtt{r}) \cup \mathit{reach}(\delta'_c, \mathtt{ll}) \qquad (\delta' \geqslant_{\mathtt{l}} \delta)$$

To reach the postcondition from line 16, the sub-copies on each side of the overlapping conjunction need to be disentangled from the original sub-dags using regions and equation 17 in the following derivations, where $\delta(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r})$, $\delta' \geqslant_{\mathtt{l}} \delta$, and $\delta'' \geqslant_{\mathtt{r}} \delta'$:

$$\begin{array}{l} \quad \mathtt{x} \mapsto_\alpha \mathtt{y}, \mathtt{l}, \mathtt{r} * \mathtt{y} \mapsto_\beta 0, \mathtt{ll}, \mathtt{rr} * \\ \quad (\mathsf{dag}_\alpha(\mathtt{l}, \delta'') * \mathsf{dag}_\beta(\mathtt{ll}, \mathit{copy}(\delta''))) \circledast \\ \quad (\mathsf{dag}_\alpha(\mathtt{r}, \delta'') * \mathsf{dag}_\beta(\mathtt{rr}, \mathit{copy}(\delta''))) \qquad (24) \\ \vdash \mathsf{dag}_\alpha(\mathtt{x}, \delta''') * \mathsf{dag}_\beta(\mathtt{y}, \mathit{copy}(\delta''')) \wedge \delta''' = [\mathtt{x} \mapsto (\mathtt{y}, \mathtt{l}, \mathtt{r})]\delta'' \\ \vdash \mathsf{ddag}(\mathtt{x}, \mathtt{y}, \delta''') \wedge \delta''' \geqslant_{\mathtt{x}} \delta \end{array}$$

```
1  struct node { struct node *next,*l,*r; };
2  void pop(void) { //{list(s) ∧ tree(t)}
3    if (!s) return;
4    struct node *c = s;
5  //  {(∃n. s ↦ n, l, r ∗ list(n)) ∧ tree(t) ∧ c = s}
6    s = c->next;
7  //  {(c ↦ s, l, r ∗ list(s)) ∧ tree(t)}
8  //  {(c ↦ s, l, r ∗ list(s)) ∧ (sktree(t, π ⊎ {c}) ∗ ptrs(π ⊎ {c}))}
9    t = tree_del(t,c);
10 //↯{(c ↦ s, −, − ∗ list(s)) ∧ (sktree(t, π) ∗ c ↦ −, −, − ∗ ptrs(π))}
11 //  {(list(s) ∧ tree(t)) ∗ c ↦ −, −, −}
12   free(c);
13 } //  {list(s)) ∧ tree(t)}
```

Figure 4: Removal from a threaded tree.

The last deduction step uses this mathematical fact:

$$\delta(x) = (0, l, r) \wedge \delta'(x) = (y, l, r) \wedge \delta' \geqslant_l \delta \wedge \delta' \geqslant_r \delta \Rightarrow \delta' \geqslant_x \delta$$

Disposing a dag. We have also verified a program that disposes a dag [HV12]. The real effort is on the mathematical side; the spatial aspects of the verification are no more complicated than what we have seen so far, and do not require regions. Because our definition of dags uses $\uplus$, we are able to establish emp at the end.

7. Overlaid Data Structures

Reasoning about threaded trees. Our examples so far have focused on graph manipulations. Ramification is also applicable in other interesting contexts, including overlaid data structures. Here we focus on one kind of overlaid structure: *threaded trees*, which overlay lists and trees. Each node has *three* links to other nodes of the data structure: a "next" pointer of a singly-linked list, and the "left" and "right" fields of a binary tree. This is a popular type of overlaid data structure: the linked list may record the set of elements some order of particular interest (*e.g.*, first-inserted to most recent), while the tree provides efficient out-of-order lookup.

Our case study is a procedure that removes the first element of the linked list from the data structure, inspired by what can be found in the Linux deadline I/O scheduler [LYP11]. The code and annotations are shown in Fig. 4. It assumes two global variables s and t that point respectively to the head of the linked list and the root of the tree. The precondition states that the two shapes span *exactly* the same memory cells, enforced by the conjunction $\wedge$. Removing from the list (line 6) merely advances the head pointer, but we cannot stop there because it leaves the overlaid structure in an inconsistent state (the items in the list and the tree must be identical).

Removing from the tree is likely to be operationally complex, potentially involving operations to rebalance, reroot, or otherwise rotate parts of the tree. Thus, we abstract this operation and assume that it is performed by a function `tree_del(t,c)`. Its spec has to express two particular facts to ensure that it is well-behaved w.r.t. the overlaid list structure: it must not tamper with the list fields, and the resulting new tree should cover the same nodes as before except for `c`. We enforce the first constraint by not giving any access rights on the list fields to the procedure, *i.e.* by restricting its precondition to the "skeleton" of the tree, and the second constraint by recording the set of nodes encompassed in the tree shape. We therefore define the following predicate that skips the list fields of each node:

$$\begin{aligned}\mathsf{sktree}(x, \pi) \overset{\text{def}}{=} {} & (x = 0 \wedge \mathsf{emp} \wedge \pi = \varnothing) \vee \exists l, r, \pi_l, \pi_r.\\ & x + 1 \mapsto l, r \ast \mathsf{sktree}(l, \pi_l) \ast \mathsf{sktree}(r, \pi_r) \wedge\\ & \pi = \{x\} \uplus \pi_l \uplus \pi_r\end{aligned}$$

The tree predicate can be split into a skeleton and a bag of points-to predicates, using the *pointers* predicate ptrs:

$$\mathsf{ptrs}(\{x_1, \ldots, x_n\}) \overset{\text{def}}{=} x_1 \mapsto - \ast \cdots \ast x_n \mapsto -$$

$$\mathsf{tree}(t) \Leftrightarrow \exists \pi.\, \mathsf{sktree}(x, \pi) \ast \mathsf{ptrs}(\pi) \qquad (25)$$

The list predicate is defined in the standard way for nil-terminated acyclic lists with two data fields:

$$\mathsf{list}(l) \overset{\text{def}}{=} (l = 0 \wedge \mathsf{emp}) \vee \exists l', x, y.\, l \mapsto l', x, y \ast \mathsf{list}(l')$$

We moreover assume that each address is aligned as a multiple of 3, to prevent *skewing*, in which a node in the tree might overlap two nodes in the list in a state satisfying $\mathsf{list}(s) \wedge \mathsf{tree}(t)$.

A general observation about how overlaid data structures are manipulated is that changes to fields of only one structure do not affect the other, *e.g.*, list induction easily proves that

$$x \mapsto n, l, r \mathbin{-\!\circledast} \mathsf{list}(s) \vdash x \mapsto n, l', r' \mathbin{-\!\!\ast} \mathsf{list}(s)$$

This reads as: if a state may be completed by a node to form a linked list, then completing it by any other node at the same location and with the same next field also yields a list. The same property for *skeleton* trees follows by induction on the size of the tree:

$$\mathsf{sktree}(t, \pi) \mathbin{-\!\circledast} \mathsf{list}(s) \vdash \mathsf{sktree}(t', \pi) \mathbin{-\!\!\ast} \mathsf{list}(s) \qquad (26)$$

Verification. The spec of `tree_del` follows the discussion above:

$\{\mathsf{sktree}(t, \pi \uplus \{c\})\}$ `u=tree_del(t,c)` $\{\mathsf{sktree}(u, \pi) \ast c{+}1 \mapsto -, -\}$

The proof sketched in Fig. 4 is mostly straightforward: if `s` is nil then the list is empty, hence so is the tree and the postcondition is trivially satisfied; otherwise, we unfold the list predicate, which enables the lookup at line 6. After that, we split the tree according to (25) and apply the following ramification:

$$\begin{aligned}& (\mathsf{c} \mapsto \mathsf{s}, l, r \ast \mathsf{list}(\mathsf{s})) \wedge (\mathsf{sktree}(\mathsf{t}, \pi \uplus \{\mathsf{c}\}) \ast \mathsf{ptrs}(\pi \uplus \{\mathsf{c}\}))\\ \vdash {} & \mathsf{sktree}(\mathsf{t}, \pi \uplus \{\mathsf{c}\}) \ast \big(\mathsf{c} + 1 \mapsto -, - \ast \mathsf{sktree}(t', \pi) \mathbin{-\!\!\ast}\\ & \qquad (\mathsf{c} \mapsto \mathsf{s}, -, - \ast \mathsf{list}(\mathsf{s})) \wedge\\ & \qquad (\mathsf{c} + 1 \mapsto -, - \ast \mathsf{sktree}(t', \pi) \ast \mathsf{ptrs}(\pi \uplus \{\mathsf{c}\}))\big)\end{aligned}$$

This ramification follows a general pattern, and we can reduce it to a much simpler one by noticing that the right-hand side conjunct is automatically handled by Lemma 4.2 from our ramification library, which can remove frames that occur within $\wedge$ ramifications. This yields the following simpler proof obligation:

$$\begin{aligned}& \mathsf{sktree}(t, \pi \uplus \{c\}) \mathbin{-\!\circledast} c \mapsto x, y, z \ast \mathsf{list}(s)\\ \vdash {} & c + 1 \mapsto -, - \ast \mathsf{sktree}(t', \pi) \mathbin{-\!\!\ast} c \mapsto x, -, - \ast \mathsf{list}(s)\end{aligned}$$

This entailment is similar to (26). The rest of the proof is immediate.

8. Cheney's Garbage Collector

It is time for the acid test: verifying the functional correctness of Cheney's garbage collector [Che70]. The general setting is as follows. There are two disjoint, equally large regions of memory, the *from-space* and the *to-space*, starting respectively at the address pointed to by `from` and `to`. Programs manipulate *objects* in the from-space. When the program wishes to allocate but the from-space has run out of room, we garbage collect by copying the entire graph of reachable objects into the to-space before swapping `from` and `to` and resuming normal execution. If the former from-space had any unreachable objects then the new from-space has some free space.

In the tradition of previous work, we make a number of simplifications. We assume that there is a single root from which all *active* objects are reachable, *i.e.* any object *not* reachable from that root can be safely reclaimed. We also restrict our study to even-aligned two-field objects that contain only pointers (including the null pointer) rather than arbitrary integers. Our proof can be modified to verify the unsimplified algorithm; *e.g.*, we can allow data by the usual systems hack of requiring that data be odd and pointers be even.

Remarkably, Cheney's algorithm migrates the graph from one space to the other using only a *constant* amount of extra memory, which is in short supply during garbage collection. Contrast this with our dag-copying example of §6 that required *linear* additional space (in both the data fields and the function stack). The cost is that we mangle the original graph, which we can live with because afterwards it will be garbage. The trick is that Cheney rewires the first field in each already-copied object in the from-space to point to its copy in the to-space. The collector can determine whether an object has already been copied, and moreover discover the copy's address, by checking if its first field points into the to-space.

Following [Gas11], we implement the algorithm as two functions: `collect` and `copy_ref`, shown in Fig. 5. In addition to the `to` and `from` pointers (fixed for the duration of the collection), they maintain two additional pointers into the to-space. First, the `scan` pointer separates the fully-processed "scanned" objects, whose pointers point into the to-space, from the partially-processed "queued" objects, whose pointers point into the from-space. Second, the `free` pointer distinguishes the first unused address in the to-space.

Initially (line 3), `scan = free = to`, meaning that no objects have been copied and the entire to-space is free. The process is initiated by copying the object pointed to by the root `r` (line 4), which allocates two cells of memory at the beginning of the to-space by increasing `free` and fills them with the values in the original object, now enqueued. After that the program loops (lines 5–12) until no queued objects remain, calling `copy_ref` on both object fields (lines 8 and 11) before incrementing `scan` to indicate that the object has been scanned. Each call to `copy_ref` swings the from-space pointers into the to-space, queuing newly encountered nodes as necessary. Fig. 6 presents an intermediate state in the execution, with one node copied and scanned and one node queued for scanning.

Formal specification. To represent states of the execution we use the following definitions. Mathematical graphs are pairs (V, E), *i.e.* we remove D and L, and the spatial predicate is accordingly

$$\mathsf{graph}(x,\gamma) \overset{\text{def}}{=} \begin{array}{l}(x = 0 \wedge \mathsf{emp}) \vee \exists l, r.\, \gamma(x) = (l, r) \wedge \\ x \mapsto l, r \circledast \mathsf{graph}(l,\gamma) \circledast \mathsf{graph}(r,\gamma)\end{array}$$

We define shorthand to express whether a node is in the from- or to-space and whether it has been copied:

$$\begin{aligned} \mathit{from}(x) &\overset{\text{def}}{=} x = 0 \vee \texttt{from} \leqslant x < \texttt{from} + \texttt{size} \\ \mathit{to}(x) &\overset{\text{def}}{=} x = 0 \vee \texttt{to} \leqslant x < \texttt{to} + \texttt{size} \\ \mathit{copied}(\gamma, x) &\overset{\text{def}}{=} x \neq 0 \wedge \mathit{from}(x) \wedge \mathit{to}(\gamma(x).1) \end{aligned}$$

We write $\mathit{from}(\gamma)$ for $\forall v \in \gamma.\, \mathit{from}(v)$ and similarly for $\mathit{to}(\gamma)$. The memory also contains a *pool* of free addresses, starting at some x, and the whole from-space, which we use to collect nodes that the algorithm disconnects (*i.e.*, from-space objects that are no longer reachable from the to-space and are therefore fresh garbage):

$$\begin{aligned} \mathsf{pool}(x) &\overset{\text{def}}{=} \mathsf{ptrs}(\{x, \ldots, \texttt{to} + \texttt{size} - 1\}) \\ \mathsf{fromsp} &\overset{\text{def}}{=} \mathsf{ptrs}(\{\texttt{from}, \ldots, \texttt{from} + \texttt{size} - 1\}) \end{aligned}$$

The main end-to-end property of a garbage collector is that the final graph is *isomorphic* to the original one. In the middle of a collection, the loop invariant is more complex; for Cheney it is that the graph rooted at `to` is isomorphic to the original one *up-to* a canonicalization function, $\mathit{canon}(\gamma)$. The canonicalization of a graph $\gamma = (V, E)$ "skips" already copied nodes by following their first field (which points to their copies). Formally, $\mathit{canon}(\gamma)$ is the graph (V', E') where $V' = \{v \in V \mid \neg \mathit{copied}(\gamma, v)\}$ and, if $E(x) = (v_1, v_2)$, then $E'(x) = (v_1', v_2')$ with

$$v_i' = \begin{cases} E(v_i).1 & \text{if } \mathit{copied}(\gamma, v_i) \\ v_i & \text{otherwise} \end{cases}$$

We write $\gamma@x \approx \gamma'@x'$ to denote *graph isomorphism* between $\mathit{canon}(\gamma) \downarrow x$ and $\mathit{canon}(\gamma') \downarrow x'$. Both $\mathit{from}(\gamma)$ and $\mathit{to}(\gamma)$ imply $\mathit{canon}(\gamma) = \gamma$, so at the end of garbage collection, when the entire graph has been moved into the to-space, we will have standard isomorphism between the old graph and the new.

The main constraints satisfied by the graph are enforced in the mathematical world by the *cheney* predicate shown in Fig. 7. Additionally, the following invariant is implicit throughout the proof:

$$\texttt{to} \leqslant \texttt{scan} \leqslant \texttt{free} < \texttt{to}+\texttt{size} \wedge \mathit{even}(\texttt{from}, \texttt{to}, \texttt{scan}, \texttt{free}, \gamma)$$

```
1  void collect(void **r) {
2  //  ${(r \mapsto r_0 * \mathsf{graph}(r_0,\gamma_0) \circledast \mathsf{fromsp}) * \mathsf{pool}(to) \wedge \mathit{from}(\gamma_0)}$
3    scan = free = to;
4    copy_ref(r);
5    while (scan != free)
6  //  $\{r \mapsto to * (\mathsf{graph}(to,\gamma) \circledast \mathsf{fromsp}) * \mathsf{pool}(free) \wedge$
       $\gamma@to \approx \gamma_0@r_0 \wedge \mathit{cheney}(\gamma, scan, free)\}$
7  { //  $\{r \mapsto to * (\mathsf{graph}(to,\gamma) \circledast scan \mapsto q_0, q_1 \circledast \mathsf{graph}(q_0,\gamma) \circledast$
       $\mathsf{graph}(q_1,\gamma) \circledast \mathsf{fromsp}) * \mathsf{pool}(free) \wedge \gamma@to \approx \gamma_0@r_0 \wedge$
       $\mathit{cheney}(\gamma, scan, free) \wedge scan \leqslant free - 2\}$
8      copy_ref(scan);
9  //↯ $\{r \mapsto to * (\mathsf{graph}(to,\gamma') \circledast scan \mapsto q_0', q_1 \circledast \mathsf{graph}(q_0',\gamma') \circledast$
       $\mathsf{graph}(q_0,\gamma') \circledast \mathsf{graph}(q_1,\gamma') \circledast \mathsf{fromsp}) * \mathsf{pool}(free) \wedge$
       $\gamma@to \approx \gamma_0@r_0 \wedge scan \leqslant free - 2 \wedge$
       $\mathit{cheney}(\gamma', scan+1, free) \wedge \gamma'@to \approx \gamma@to\}$
10 //  $\{r \mapsto to * (\mathsf{graph}(to,\gamma') \circledast \mathsf{fromsp}) * \mathsf{pool}(free) \wedge$
       $\gamma'@to \approx \gamma_0@r_0 \wedge \mathit{cheney}(\gamma', scan+1, free) \wedge$
       $scan \leqslant free - 2\}$
11     copy_ref(scan + 1);
12     scan = scan + 2; }
13 } //  $\{r \mapsto to * \mathsf{graph}(to,\gamma) * \mathsf{fromsp} * \mathsf{pool}(free) \wedge$
       $\mathit{to}(\gamma) \wedge \gamma@to \approx \gamma_0@r_0\}$
14
15 void copy_ref(void **p) {
16 //  $\{(p \mapsto q \circledast \mathsf{graph}(q,\gamma)) * \mathsf{pool}(f) \wedge \mathit{cheney}(\gamma, p, f) \wedge free = f\}$
17  if (*p) {
18 //  $\{(p \mapsto q \circledast q \mapsto a, b \circledast \mathsf{graph}(a,\gamma) \circledast \mathsf{graph}(b,\gamma)) *$
       $\mathsf{pool}(f) \wedge \mathit{cheney}(\gamma, p, f) \wedge free = f\}$
19    void *obj = *p;
20    void *fwd = *obj;
21 //  $\{(p \mapsto obj \circledast obj \mapsto fwd, b \circledast \mathsf{graph}(fwd,\gamma) \circledast \mathsf{graph}(b,\gamma)) *$
       $\mathsf{pool}(f) \wedge \mathit{cheney}(\gamma, p, f) \wedge free = f\}$
22    if (to <= fwd && fwd < to + size){
23 //  $\{(p \mapsto obj \circledast obj \mapsto fwd, b \circledast \mathsf{graph}(fwd,\gamma) \circledast \mathsf{graph}(b,\gamma)) *$
       $\mathsf{pool}(f) \wedge \mathit{cheney}(\gamma, p, f) \wedge free = f \wedge \mathit{to}(fwd)\}$
24      *p = fwd;
25 //↯ $\{(p \mapsto fwd \circledast obj \mapsto fwd, b \circledast \mathsf{graph}(fwd,\gamma') \circledast \mathsf{graph}(b,\gamma')) *$
       $\mathsf{pool}(f) \wedge \mathit{cheney}(\gamma, p, f) \wedge free = f \wedge \mathit{to}(fwd) \wedge$
       $\mathit{cheney}(\gamma', p+1, f) \wedge \gamma' = [p \mapsto fwd]\gamma\}$
26    } else {
27 //  $\{(p \mapsto obj \circledast obj \mapsto fwd, b \circledast \mathsf{graph}(fwd,\gamma) \circledast \mathsf{graph}(b,\gamma)) *$
       $\mathsf{pool}(f) \wedge \mathit{cheney}(\gamma, p, f) \wedge free = f \wedge \mathit{from}(fwd)\}$
28      void *new = free;
29      free = free + 2;
30      *new = *obj;
31      *(new + 1) = *(obj + 1);
32 //  $\{(p \mapsto obj \circledast obj \mapsto fwd, b \circledast \mathsf{graph}(fwd,\gamma) \circledast \mathsf{graph}(b,\gamma)) *$
       $new \mapsto fwd, b * \mathsf{pool}(free) \wedge free = f + 2 \wedge$
       $\mathit{cheney}(\gamma, p, f) \wedge \mathit{from}(fwd) \wedge new = f\}$
33 //  $\{(p \mapsto obj \circledast obj \mapsto fwd, b \circledast new \mapsto fwd, b \circledast$
       $\mathsf{graph}(fwd,\gamma_1) \circledast \mathsf{graph}(b,\gamma_1)) * \mathsf{pool}(free) \wedge$
       $free = f + 2 \wedge \mathit{cheney}(\gamma, p, f) \wedge \mathit{from}(fwd) \wedge$
       $new = f \wedge \gamma_1 = [new \mapsto fwd, b]\gamma\}$
34      *obj = new;
35 //↯ $\{(p \mapsto obj \circledast obj \mapsto new, b \circledast new \mapsto fwd, b \circledast$
       $\mathsf{graph}(fwd,\gamma_2) \circledast \mathsf{graph}(b,\gamma_2)) * \mathsf{pool}(free) \wedge$
       $\mathit{cheney}(\gamma, p, f) \wedge free = f + 2 \wedge \mathit{from}(fwd) \wedge$
       $new = f \wedge \gamma_2 = [obj \mapsto new][new \mapsto fwd, b]\gamma\}$
36      *p = new;
37 //↯ $\{(p \mapsto new \circledast obj \mapsto new, b \circledast new \mapsto fwd, b \circledast$
       $\mathsf{graph}(fwd,\gamma') \circledast \mathsf{graph}(b,\gamma')) * \mathsf{pool}(free) \wedge$
       $\mathit{cheney}(\gamma, p, f) \wedge free = f + 2 \wedge \mathit{from}(fwd) \wedge$
       $new = f \wedge \gamma' = [p \mapsto new][obj \mapsto new][new \mapsto fwd, b]\gamma\}$
38 }}} //  $\{(p \mapsto q' \circledast \mathsf{graph}(q',\gamma') \circledast \mathsf{graph}(q,\gamma')) * \mathsf{pool}(free) \wedge$
       $\mathit{cheney}(\gamma', p+1, free) \wedge \gamma@to \approx \gamma'@to \wedge$
       $\gamma' \uparrow \{p, q, q'\} = \gamma \uparrow \{p, q\} \wedge free \geqslant f\}$
```

Figure 5: Proof sketch of Cheney's garbage collector.

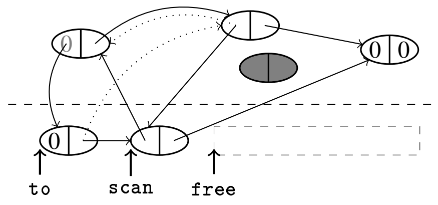


Figure 6: Transient state of the memory during garbage collection. Previous field values are indicated by 0 or dotted pointer arrows.

$$
\begin{aligned}
&\mathit{cheney}(\gamma, s, f) \overset{\text{def}}{=} \mathit{to}(s) \wedge \mathit{to}(f) \wedge \\
&|\{v \mid \mathit{copied}(\gamma, v)\}| = (f - \texttt{to})/2 \wedge && (27)\\
&\{\texttt{to}, \ldots, f-2\} \subseteq \gamma \downarrow \texttt{to} \wedge && (28)\\
&\forall v \in \gamma.\, \forall a, b.\, \gamma(v) = (a, b) \Rightarrow \\
&(\mathit{to}(v) \wedge ((v < s \wedge \mathit{to}(a)) \vee (v \geqslant s \wedge \mathit{from}(a))) \\
&\quad \wedge ((v{+}1 < s \wedge \mathit{to}(b)) \vee (v{+}1 \geqslant s \wedge \mathit{from}(b)))) \vee && (29)\\
&(\mathit{from}(v) \wedge \mathit{from}(b) \wedge (\mathit{to}(a) \Rightarrow \gamma@b \approx \gamma@(\gamma(a).2))) && (30)
\end{aligned}
$$

Figure 7: Cheney graphs. Parameter s is the first unscanned address and f is the beginning of the free space. There are as many nodes in the to-space as there are copied nodes (27), which ensures that we never exhaust our free space at line 29. Every cell in the to-space is reachable (28). For each object, either (29) it is in the to-space and either scanned with fields pointing to the to-space, or queued with fields pointing to the from-space; or (30) it is in the from-space, with fields either entirely pointing to the from-space or with first field pointing to its copy in the to-space, in which case the second fields of the object and its copy point to isomorphic sub-graphs.

Here *even* forces all objects and global pointers to be aligned on even boundaries. Notice that a graph entirely in the from-space is automatically a Cheney graph: $\mathit{from}(\gamma) \Rightarrow \mathit{cheney}(\gamma, \texttt{to}, \texttt{to})$. Similarly, if $\texttt{to} \in \gamma$ and $s = f$ then $\mathit{cheney}(\gamma, s, f) \Rightarrow \mathit{to}(\gamma)$. These observations are enough to go from the precondition to the loop invariant, and from the loop invariant to the postcondition.

Verification of `copy_ref`. We omit the case of the first call to `copy_ref` (line 4) and focus instead on the more interesting case of calls made from the main loop (lines 8 and 11). `copy_ref` swings the fields of a queued object from their original targets in the from-space to their targets' copies in the to-space. If the fields contain the null pointer then no action is required and the post is direct from the pre. Otherwise, we can unfold the graph (line 16) to expose the target object in the from-space. We then examine its first field, `fwd`.

If `fwd` is in the to-space, then the target object has a copy located there and we swing the pointer to it. The ramification immediately follows from Lemma 4.11, slightly modified to handle single-field updates, and updates the graph to $\gamma' = [\texttt{p} \mapsto \texttt{fwd}]\gamma$ (where a single field update $[x \mapsto y]\gamma$ corresponds to $[x \mapsto y, \gamma(x).2]\gamma$ if x is even and to $[x-1 \mapsto \gamma(x).1, y]\gamma$ if x is odd). The actual proof effort at that point is to mathematically establish $\mathit{cheney}(\gamma', \texttt{p}{+}1, \texttt{free})$ and $\gamma@\texttt{to} \approx \gamma'@\texttt{to}$. The former holds because the only update is that `p` changed from queued in γ to scanned in γ' ($\texttt{p} < \texttt{p}+1$) and from pointing to the from-space to the to-space. For the latter, notice that $\mathit{canon}(\gamma)(\texttt{p}) = \texttt{fwd}$, so swinging `p` to point to `fwd` gives the same canonical graph: $\mathit{canon}(\gamma) = \mathit{canon}(\gamma')$, hence $\gamma@\texttt{to} \approx \gamma'@\texttt{to}$.

If the object has not been copied yet, we reserve two units of space at the position of the `free` pointer (by advancing it, line 27), and fill them with the object's fields. Since the pool of free space is kept $\ast$-separated from the current graph of objects, FRAME is able to deal nicely with the heap mutations up to the assignment at line 35. Now we rewrite the state to integrate the new object into the main graph (Lemma 4.10), then swing both the current field `p` and the first field of the target object `obj` to point to the copy `new`, yielding two successive ramifications that update the global graph accordingly, which we can discharge with Lemma 4.11. Once again, RAMIFY and our library allow us to progress past updates to the shared state; the actual complexity resides in establishing mathematical facts about graphs in the postcondition. Their proof is similar to the case in which `fwd` was in the to-space to begin with. We have to prove that `new` is reachable from `to`, as required by 28, which holds because `p` is reachable from `to` and points to it. The isomorphism holds because `new` and `obj` have identical contents.

Verification of `collect`. The main function first copies the root node in the graph using an alternative (simpler) spec for `copy_ref` to establish the loop invariant (line 6, in which we leave out the case $r_0 = 0$ of an empty graph). It then enters a loop that updates both fields of the first unscanned object in succession (which may queue up new objects), repeating until all objects have been scanned. The looping condition allow us to go from the invariant at line 6 to the assertion at line 7 (in particular, $\texttt{to} \rightsquigarrow^* \texttt{scan}$ by (28) so Lemma 4.9 applies). The ramification at line 9 makes interesting use of our ramification library. Lemma 4.12 tells us that each individual pointer in fromsp (as well as the other field of `scan`) is preserved. Combining this with Lemma 4.6 yields that the whole of fromsp is preserved. The graphs are updated thanks to Lemma 4.13. We finally combine both our conclusions with another application of Lemma 4.6. To deduce line 10, we fold back the sub-graph rooted at `scan` into the main one rooted at `to`, which leaves the following spatial deduction, which holds because, together, $\mathsf{graph}(\texttt{to}, \gamma')$ and fromsp contain the whole allocated heap:

$$\mathsf{graph}(\texttt{to}, \gamma') \uplus \mathsf{graph}(q_0, \gamma') \uplus \mathsf{fromsp} \vdash \mathsf{graph}(\texttt{to}, \gamma') \uplus \mathsf{fromsp}$$

The second call to `copy_ref` is analogous to the first, and after we advance `scan` we reach the loop invariant.[5]

Related work. Cheney's garbage collector has been a benchmark of sorts for heap-aware verification, especially in separation logic [MAY06, TSBR08, Gas11]. Previous verifications worked by exploding the spatial graph into its individual nodes, and grouping those into several *disjoint* groups corresponding to the intersections of various heap regions (from and to-space, scanned and unscanned, etc.). Our approach uses a single, *generic* inductive graph predicate, and the intricacies of reasoning about those regions is handled at the level of mathematical graphs. This division of labor yields, in our opinion, a much more pleasant and concise proof, which enjoys relatively intuitive and natural invariants. It also compares favorably with respect to proofs in other formalisms, *e.g.* Boogie [HP09].

9. Universality, Strongest Posts, and Extensions

Here we discuss the general applicability of the ramify rule as well as an alternative form of the rule. We also discuss a number of extensions to apply ramifications to more examples, including the overlapping conjunction $\uplus$, regions, and higher-order settings.

9.1 Universality of Ramification

In §3.3 we showed that the frame rule was a consequence of the ramify rule. Somewhat surprisingly, the converse is also true.

Theorem 9.1 (RAMIFY)

$$\frac{\{P\}\ c\ \{Q\} \quad R \vdash P \ast (Q \mathrel{-\!\!\ast} R')}{\{R\}\ c\ \{R'\}} \quad \begin{array}{l} \mathit{fv}(Q \mathrel{-\!\!\ast} R') \cap \\ \mathit{modif}(c) = \varnothing \end{array}$$

Proof By the short derivation given in Fig. 8. ▫

[5] In the above proof, the global variable `free` is modified by `copy_ref`, but appears in our ramified assertions. We circumvent this issue by treating `free` as a resource: we remove our knowledge about `free` when `copy_ref` is called, and only get to assume what is in `copy_ref`'s post-condition in the post-ramified state (*e.g.* line 9).

$$\dfrac{\dfrac{}{R \vdash P \star (Q \mathrel{-\!\!\ast} R')}\,\text{Hyp.} \qquad \dfrac{\dfrac{}{\{P\}\;c\;\{Q\}}\,\text{Hyp.} \qquad \dfrac{}{\mathit{modif}(c) \cap \mathit{fv}(Q \mathrel{-\!\!\ast} R') = \varnothing}\,\text{Hyp.}}{\{P \star (Q \mathrel{-\!\!\ast} R')\}\;c\;\{Q \star (Q \mathrel{-\!\!\ast} R')\}}\,\text{Frame} \qquad \dfrac{}{Q \star (Q \mathrel{-\!\!\ast} R') \vdash R'}\,\text{Modus Ponens}}{\{R\}\;c\;\{R'\}}\,\text{Consequence}$$

Figure 8: Proof of RAMIFY.

Because theorem 9.1 only requires frame and consequence, ramify is valid in any separation logic. This is very handy, because it means that we do not need to modify the numerous flavors of separation logic in previous work to incorporate ramification: it has been there all along, just waiting for its importance to be recognized.

9.2 Weakest preconditions and strongest postconditions

In fact, our ramify rule appears in the separation logic folklore as a weakest precondition rule, codified as follows:

Lemma 9.1 (Weakest Pre) *Given a postcondition R' and a specification $\{P\}\;c\;\{Q\}$, then $P \star (Q \mathrel{-\!\!\ast} R')$ is the weakest precondition,* i.e.*, given any specification $\{R\}\;c\;\{R'\}$, then $R \vdash P \star (Q \mathrel{-\!\!\ast} R')$.*

Our examples demonstrate that we can successfully ramify with weakest precondition. Can we also succeed with strongest postcondition, *i.e.*, with the following "forward ramify" rule:

FWRAMIFY
$$\dfrac{\{P\}\;c\;\{Q\} \qquad R \vdash P \star \mathsf{true} \qquad (P \mathrel{-\!\circledast} R) \star Q \vdash R'}{\{R\}\;c\;\{R'\}}$$

The $(P \mathrel{-\!\circledast} R) \star Q \vdash R'$ pattern is reminiscent of a pattern used in RGSep [VP07] to characterize *stability* by setting R' to R. In RGSep the focus is on concurrency, and a thread's collaborators may take an unknown number of actions. In our setting we know that a given specification will execute exactly once, which we leverage by allowing the consequent to be the more general R' rather than R. When P is precise, FWRAMIFY gives the strongest postcondition:

Lemma 9.2 (Strongest Post) *Given precondition R and $\{P\}\;c\;\{Q\}$, if P is precise then $(P \mathrel{-\!\circledast} R) \star Q$ is the strongest postcondition.*

As it happens, whenever P is precise, RAMIFY and FWRAMIFY are each derivable from the other. However, precision is actually only needed when starting from FWRAMIFY, and so we consider RAMIFY to be fundamental. Moreover, although we were able to prove some of the examples using FWRAMIFY, we found its $\mathrel{-\!\circledast}$ idiom to be harder to reason about than the $\mathrel{-\!\!\ast}$ idiom in RAMIFY.

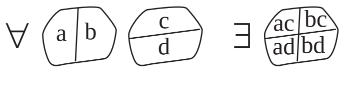
9.3 Extensions supporting ramification

We can ramify in any separation logic, but verifying certain programs can require various extensions, such as regions in §6. Here we detail other extensions, starting with a more careful look at $\cup\!\!\!\ast$. Additional comments on these extensions may be found in [HV12].

The overlapping conjunction $\cup\!\!\!\ast$. Although the overlapping conjunction $\cup\!\!\!\ast$ appears occasionally in the literature (under such names as "fusion", "relevance conjunction", and "sepish"), its properties are not well-understood for abstract separation logics.

A separation algebra [COY07] is a partial commutative monoid with cancellation $(S, \oplus)$ that provides an abstract model for separation logic. Although the overlapping conjunction $\cup\!\!\!\ast$ can be defined in any separation algebra, it is not necessarily easy to use: in fact, several critical properties require stronger separation algebra axioms. We propose using a variant described by Dockins *et al.* [DHA09] that has multiple units, *disjointness* (*i.e.*, $x \oplus x = y \Rightarrow x = y$), and a kind of distributivity property called "cross split":

$a \oplus b = z \wedge c \oplus d = z \Rightarrow \exists ac, ad, bc, bd.$
$ac \oplus ad = a \wedge bc \oplus bd = b \wedge ac \oplus bc = c \wedge ad \oplus bd = d$



That is, if an element (*e.g.*, a heaplet) can be split in two different ways, then there are *four* subobjects which partition the original and respect the original splittings. Cross split is not discussed much in the literature, but we discovered that it is vital for reasoning about the overlapping conjunction $\cup\!\!\!\ast$, which is not even associative without it. In fact, virtually all of our proofs that use $\cup\!\!\!\ast$ assume cross split.

Many—but by no means all—separation algebras used in practice satisfy cross split, including the canonical model of heaplets as partial maps from addresses to values (quarters are found by set intersection on the domain). Users of our theory must therefore verify that the separation algebras they care about satisfy cross split.

Explicit overlapping conjunction. Cherini and Blanco proposed a generalization of $P \cup\!\!\!\ast Q$ that tagged the shared core with an explicit description C [CB09], defined as follows:

$$\sigma \models P\langle \cup\!\!\!\ast : C\rangle Q \;\overset{\text{def}}{=}\; \exists \sigma_1, \sigma_2, \sigma_3.\; (\sigma_1 \oplus \sigma_2 \oplus \sigma_3 = s) \wedge (\sigma_1 \oplus \sigma_2 \models P) \;\wedge\; (\sigma_2 \models C) \;\wedge\; (\sigma_2 \oplus \sigma_3 \models Q)$$

This *explicit overlapping conjunction* is more expressive than $\cup\!\!\!\ast$:

$$P\langle \cup\!\!\!\ast : \mathsf{true}\rangle Q \quad \dashv\vdash \quad P \cup\!\!\!\ast Q$$

Moreover, Cherini and Blanco developed the following proof rule:

EXPRAMIFY
$$\dfrac{\{P\}\;c\;\{Q\} \qquad (C \mathrel{-\!\circledast} R) \star C' \vdash R' \qquad Q \vdash C' \star \mathsf{true}}{\{P\langle \cup\!\!\!\ast : C\rangle R\}\;c\;\{Q\langle \cup\!\!\!\ast : C'\rangle R'\}}$$

Unfortunately, EXPRAMIFY is not useful to verify any of our examples because we focus on *unspecified sharing*—that is, we do not know exactly what the overlap is (*e.g.*, the precise nodes shared between the children of a dag node), and hence cannot pick C or C' other than true. In general unspecified sharing sharing is more difficult to verify than specified sharing, which is apparent when one tries to apply EXPRAMIFY:

$$(\mathsf{true} \mathrel{-\!\circledast} R) \star \mathsf{true} \vdash R'$$

In other words, start from R, remove an unrestricted subheap, replace it with a second unrestricted heap, and now prove R'. Yikes!

Conversely, EXPRAMIFY cannot verify our overlaid example (§7) because instead of C and C' being too weak, they represent the entire structure (*i.e.*, $P = C = R$ and $Q = C' = R'$). Applying EXPRAMIFY then makes no progress because the "simpler" Hoare subproof $\{P\}\;c\;\{Q\}$ is actually identical to the goal.

All of that said, Cherini and Blanco demonstrate how to use EXPRAMIFY to verify programs that operate in the special case of *specified partial sharing*—*i.e.*, when nontrivial C and C' are known and not the entire P, Q, R, or R'. Happily, EXPRAMIFY is derivable from RAMIFY, so we can reuse all of their verifications.

Fractional shares, actions, and tight regions. In §4.2 we pointed out that naïve attempts to verify `mark` using the shape-only $\mathsf{dag}(x)$ predicate were unsound. In this paper we focused on functional correctness instead, but we also experimented various other methods for guaranteeing that the graph is not overly mangled, including fractional shares [DHA09], actions in the style of RGSep [Vaf07], and a variant of regions that could prevent memory deallocation. Each method had some benefits but also required some additional formalism; the tradeoffs were unclear.

Higher-order settings. In recent years there have been several flavors of separation logic to reason about higher-order state such as the resource invariants of concurrent separation logic with first-class locks [HAZ08]. Although we did not do any ramifications for genuine higher-order settings (which are often very complicated in ways unrelated to their higher-orderness), we did check a few of the ramifications from this paper in Coq within approximating separation algebras [HDA10], and believe that the higher-orderness by itself poses no fundamental difficulties.

10. Related Work and Conclusion

There is a large body of work, orthogonal to ours, tackling the design and proof of algorithms for data structures with sharing. Its counterpart in program verification spans a range of domains, and we begin this section with other separation logic based analyses.

Our reasoning about graphs owes a lot to the overlapping conjunction $\uplus$, which has roots in relevant logic [Urq72]. Many people have rediscovered it in the context of separation logic [Rey03, GMS12], who defined inductive graphs and dags as we did, but did not provide a means to reason about them. Cherini and Blanco were able to reason about a *specified* version of $\uplus$ using a more domain-specific framework than ours, as discussed in §9.3.

More recently, Mehnert *et al.* and Krishnaswami *et al.* have used some form of ramification to verify respectively implementations of snapshottable trees [MSBS12] and programs that follow the subject-observer pattern [KBA10], both of which involved unspecified sharing. Their ramifications are restricted to ad-hoc "ramification operators" tailored for each example, and the logic itself is domain-specific and done modulo a predicate on the global heap. It would be interesting to try and recast their proofs in our setting. Lee *et al.* devised an automatic analysis for threaded trees that instruments the results of separate analyses for lists and for trees [LYP11].

Moreover, several works have dealt with *definite* sharing in separation logic, *e.g.* doubly linked lists [Rey00], trees with parent pointers, skip or cyclic lists, etc. In these cases, one always knows *what* is shared and *by whom*. On the other hand, handling *indefinite* sharing, such as in this paper, was achieved only by resorting to tricks that specified or avoided the sharing. Yang's proof of the Schorr-Waite graph marking algorithm [Yan01, §7] (later mechanized in Isabelle/HOL [MN05]) does not define a spatial graph predicate, but rather describes the graph by its spanning tree. Attempts to lift this kind of reasoning to other algorithms on dags and graphs has led to convoluted predicates that explicitly deal with sharing and hack data structures into $*$-conjoined pieces, often in ways tied to the behavior of the program at hand [BCO04].

Several other frameworks have dealt with sharing in programs. In shape analysis, Hob can prove data structure consistency when one can expose a *backbone* into which objects ultimately point [WKL$^+$06], and TVLA has been used to prove partial correctness of a mark-and-sweep garbage collector and the Schorr-Waite algorithm [MSRF04]. Hawblitzel and Petrank have used Boogie to automatically verify garbage collectors [HP09]. However, these works do not provide *compositional* reasoning for sharing.

It would be interesting to see if we can import ramification into other frameworks, such as Dafny [Lei10], whose reasoning about the heap is based on dynamic frames (a cousin of separation logic).

Conclusion. We have presented a new paradigm, ramification, valid in any separation logic, for the compositional verification of programs that manipulate data structures with both specified and unspecified sharing. We gave a ramification library that helps simplify ramification entailments in general and reduces local spatial updates to abstract mathematical reasoning. We have demonstrated the applicability of our framework by providing concise, local specifications for a range of examples and data structures, including Cheney's garbage collector. These initial successes lead us to believe that ramification provides a robust basis for elegant, compositional reasoning about sharing in data structures.

References

[BCO04] R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation and aliasing. In *SPACE*, 2004.

[BCY06] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. *ENTCS*, 155, 2006.

[Bor00] R. Bornat. Proving pointer programs in hoare logic. In *MPC*, 2000.

[CB09] R. Cherini and J. O. Blanco. Local reasoning for abstraction and sharing. In *SAC*, 2009.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *C. ACM*, 13(11), 1970.

[COY07] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, 2007.

[DHA09] R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, 2009.

[Fin87] J. Finger. *Exploiting constraints in design synthesis*. PhD thesis, Stanford University, 1987.

[Gas11] H. Gast. Developer-oriented correctness proofs - a case study of Cheney's algorithm. In *ICFEM*, 2011.

[GMS12] P. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for JavaScript. In *POPL*, 2012.

[HAZ08] A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.

[HDA10] A. Hobor, R. Dockins, and A. W. Appel. A logical mix of approximation and separation. In *APLAS*, ENTCS, 2010.

[HP09] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *POPL*, 2009.

[HV12] A. Hobor and J. Villard. The ramifications of sharing in data structures, 2012. `http://www.cs.ucl.ac.uk/staff/J.Villard/popl13/ramifications-long.pdf`.

[IO01] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.

[KBA10] N. Krishnaswami, L. Birkedal, and J. Aldrich. Verifying event-driven programs using ramified frame properties. In *TLDI*, 2010.

[Lei10] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, 1988.

[LYP11] O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *CAV*, 2011.

[MAY06] N. Marti, R. Affeldt, and A. Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In *ICFEM*, 2006.

[MN05] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Inf. Comput.*, 199(1-2), 2005.

[MSBS12] H. Mehnert, F. Sieczkowski, L. Birkedal, and P. Sestoft. Formalized verification of snapshotable trees: Separation and sharing. In *VSTTE*, 2012.

[MSRF04] R. Manevich, S. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS*, 2004.

[Rey00] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, Cornerstones of Computing, 2000.

[Rey02] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[Rey03] J. C. Reynolds. A short course on separation logic. 2003.

[Thi01] M. Thielscher. The qualification problem: A solution to the problem of anomalous models. *Artificial Intelligence*, 131(1), 2001.

[TSBR08] N. Torp-Smith, L. Birkedal, and J. C. Reynolds. Local reasoning about a copying garbage collector. *ACM TOPLAS*, 30(4), 2008.

[Urq72] A. Urquhart. Semantics for relevant logics. *J. Symb. Log.*, 37(1), 1972.

[Vaf07] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

[VP07] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.

[WKL$^+$06] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. C. Rinard. Field constraint analysis. In *VMCAI*, 2006.

[Yan01] H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, 2001.