Hybrid separation logic

A story of stars and labels

Armaël Guéneau
under the direction of Jules Villard

Contents

Introduction 1

1 Playing with separation logic: the `copytree` case study 2

- 1.1 Separation logic: a quick introduction . . . 2
- 1.2 Using separation logic to verify a simple program . . . 4
  - 1.2.1 The `copytree` program: a first specification . . . 5
  - 1.2.2 Using `copytree` on DAGs . . . 5

2 Extending separation logic: hybrid separation logic 8

- 2.1 Definitions . . . 8
- 2.2 The hybrid separation logic toolbox: useful remarks & lemmas . . . 9
  - 2.2.1 Hybrid formulæ . . . 9
  - 2.2.2 Hybrid formulæ in Hoare triples . . . 10

3 Hybrid separation logic in practice: case studies 12

- 3.1 `copytree` on DAGs, revamped . . . 12
- 3.2 Modifying the content of the input: `mark` . . . 13
- 3.3 Modifying the shape of the input: `spanning` . . . 15

4 Towards automatic reasoning on hybrid formulæ 19

- 4.1 Integration in `llStar`: proving `copytree` . . . 19
- 4.2 Thoughts about handling hybrid formulæ in tools . . . 20
  - 4.2.1 Expressive power of hybrid formulæ . . . 20
  - 4.2.2 Standard form . . . 21
  - 4.2.3 Canonical form . . . 22

Conclusion 24

Bibliography 24

A Examples codes and proofs 26

B Detailed proof of `spanning` specification 32

C Lemmas proofs 33

D `llStar` rules for proving `copytree` 34

Introduction

The general field of study of this internship at Imperial College of London, supervised by Jules Villard, was program verification. More precisely, writing specifications for imperative programs, and proving them. Such a specification being proved means that the program actually observes the specified behavior, and moreover does not do illegal operations, such as, eg. dereferencing a `NULL` pointer.

Our mindset during this internship was rather pragmatic, keeping proof automation in mind at all times. We wanted to verify programs, but we wanted to do it in the most simple, and the most easily automatable way possible. There is always some tension: if we decide to prove a specification describing the precise behavior of the program (*functional correctness*), this proof will likely not be automatable. On the other hand, proving only the *memory safety* of the program is too weak to prove recursive programs, for example. We will have to find the right balance, between the will to simplify a program specification to its essence, and the need for it to be still provable.

An example of such choice is, for example, when dealing with a program copying a tree, having to choose between a specification expressing that the new tree is identical to the input one, and an other one saying just that, after running the program, you now have two trees.

Our contributions consists of a new framework for writing program specifications. We first show that it leads to natural and very convenient specifications. We then identify various lemmas and use-cases, gathering a "toolbox" for our framework, illustrating different patterns via examples, where we are able to show simpler proofs than the state of the art. Finally, we discuss about automation of proofs in our framework.

1 Playing with separation logic: the `copytree` case study

There exists a well-known and well-used framework for the verification of imperative programs: separation logic [14], used in automated provers (Corestar [3], SLAyer [2], [6], [18], Infer [5], Hip/Sleek [10]), implemented as libraries (Ynot [7], MSL [1]), etc.

After a short introduction to the principles of separation logic, we will illustrate its use on a small example program. Despite being seemingly simple, it will shed light on some difficulties. These difficulties have led, in the literature, to complicated and hardly automatable solutions - thus motivating our contribution.

1.1 Separation logic: a quick introduction

Separation logic was designed as an extension of Hoare logic able to reason about imperative programs manipulating shared mutable data structures [14]. This means first that a specification for a program c is given as a Hoare triple, of the form $\{P\}$ c $\{Q\}$. Secondly, the extension is twofold: the logic of assertions (used to express the precondition P and the postcondition Q of a Hoare triple) is extended, and a new deduction rule for Hoare triples is added.

Assertion formulæ are written in a language derived from the logic of Boolean Bunched Implications (BBI) [12]. They describe properties of memory heaps: we want to verify programs manipulating on-heap data structures.

Syntactically, the language of assertion formulæ consists of the usual formulæ of propositional logic, plus three new forms that describe the heap:

Definition 1 (Separation logic formulæ syntax)

$$
\begin{array}{lll}
P, Q ::= & \text{propositional logic} & \\
\mid & \textbf{emp} & \text{(empty heap)} \\
\mid & x \mapsto x' & (x, x' \text{ are integer expressions}) \text{ (singleton heap)} \\
\mid & P * Q & \text{(separating conjunction)}
\end{array}
$$

The integer expressions occurring in singleton heaps ($x \mapsto x'$) may contain integer variables, interpreted as in standard Hoare logic, using some store associating values to variables (similar to the program stack).

Semantically, models of such logic need to describe memory heaps. We will not try to describe more precisely these models [4] [14], and just fix partial functions from addresses (integers) to values (integers) as representations for memory heaps.

Definition 2 (Models for separation logic)

$$
\begin{array}{lll}
h, h' \quad \text{(heaps)} & ::= & \text{Addresses} \rightharpoonup \text{Values} \\
\text{Addresses, Values} & ::= & \mathbb{N}
\end{array}
$$

We will write $\text{dom}(h)$ for the domain of the h partial function.

Then, we can define by induction the relation $s, h \models P$, meaning that the heap represented by h, associated to the store s, satisfies the assertion P:

Definition 3 (Formulæ semantics)

$$
\begin{array}{lcl}
s, h \models \top & & \text{always} \\
s, h \models \bot & & \text{never} \\
s, h \models \neg A & \Leftrightarrow & h \not\models_\rho A \\
s, h \models A_1 \wedge A_2 & \Leftrightarrow & h \models A_1 \text{ and } h \models A_2 \\
s, h \models A_1 \vee A_2 & \Leftrightarrow & h \models A_1 \text{ or } h \models A_2 \\
s, h \models A_1 \to A_2 & \Leftrightarrow & h \models A_1 \text{ implies } h \models A_2 \\
 & & \\
s, h \models \textbf{emp} & \Leftrightarrow & \text{dom}(h) = \varnothing \\
s, h \models x \mapsto x' & \Leftrightarrow & \text{dom}(h) = \{[\![x]\!]_s\} \text{ and } h([\![x]\!]_s) = [\![x']\!]_s \\
s, h \models A_1 * A_2 & \Leftrightarrow & \exists h_1, h_2 : \text{dom}(h_1) \cap \text{dom}(h_2) = \varnothing \\
 & & \text{and } h = h_1 \cup h_2 \\
 & & \text{and } h_1 \models A_1 \text{ and } h_2 \models A_2
\end{array}
$$

In the rest of this document, we will define:

Definition 4

$$h_1 \oplus h_2 \equiv h_1 \cup h_2 \text{ when } \mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \varnothing$$

so we can also write:

$$s, h \models A_1 * A_2 \quad \Leftrightarrow \quad \exists h_1, h_2 : h = h_1 \oplus h_2 \text{ and } h_1 \models A_1 \text{ and } h_2 \models A_2.$$

emp describes an empty heap, $x \mapsto x'$ a heap containing only one memory cell, at address x, containing x'. $A_1 * A_2$ describes the conjunction of two separated heaps, one satisfying A_1, the other satisfying A_2.

Separation logics are usually extended with a few more operators, such as $-\!*$, $\circledast$, etc. We will introduce them later when needed. We will also allow the - quite common - use of existential quantifiers on stack variables, for the purpose of introducing new names.

Separation logic is an extension of Hoare logic, which aim is to derive Hoare triples, of the form $\{P\}\ \mathsf{c}\ \{Q\}$. In includes all the usual rules of Hoare logic, and adds a new more: the *frame rule*, allowing to use the "separated" fact expressed by $*$. That is, if we act on some part of the heap, separated parts are not modified:

Definition 5 (Frame rule)

$$\text{Frame}\ \frac{\{P\}\ \mathsf{c}\ \{Q\}}{\{P * R\}\ \mathsf{c}\ \{Q * R\}}$$

when no variable free in R is modified by c

Using the frame rule, one can extend a local specification to a more global one, adding arbitrary predicates describing parts of the heap not accessed by c. Thinking backwards, this means that given some complicated description of the heap, we will be able to temporarily "frame out" the not interesting bits, and prove some specification on the smallest footprint needed.

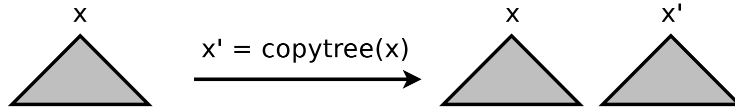
1.2 Using separation logic to verify a simple program

Following the path of Reynolds [14], we will now illustrate how separation logic can be used to write and prove a specification for some (not so) simple program: `copytree`.

1.2.1 The `copytree` program: a first specification

The `copytree` program (in Figure 14), given a tree as input, explores it recursively and copies it, finally returning the copy.

Because `copytree` does not reuse memory cells from the original tree, the new tree is totally disjoint from the first one. Thus, starting with a heap holding some tree rooted in `x`, after running `copytree` we obtain a heap holding two separated trees, rooted in `x` and `x'`.



Thus, given some "tree x" predicate describing a tree-shaped heap, we can write a specification for `copytree`, thanks to the separating conjuction ($*$):

$$\{\text{tree x}\}\ \texttt{x' = copytree(x)}\ \{\text{tree x} * \text{tree x'}\}$$

To complete the specification, the "tree x" predicate must be defined. This can be done inductively ($x \mapsto a, b, c$ being a shorthand for $(x \mapsto a) * (x+1 \mapsto b) * (x+2 \mapsto c)$):

$$\begin{aligned} \text{tree } x \quad \equiv \quad & (\exists val, l, r : x \mapsto val, l, r * \text{tree } l * \text{tree } r) \\ & \vee (x = 0 \wedge \texttt{emp}) \end{aligned}$$

In the non-trivial case, the two $*$ express two different things: first, the root node is separated from the subtrees, so there is no cycles, the subtrees cannot point back to the root. Thinking backwards, if there was a cycle, then x would be in a subtree (pointers of a tree only point to elements of itself) - but this is not possible as x is separated from the subtrees.

Secondly, the subtrees are separated from each other, so there is no sharing between them, by the same kind of reasoning.

The proof of the specification that follows from here is relatively simple, and consists only of applications of the frame rule, the consequence rule and variable-handling rules. To be completely formal, proofs of specifications should be presented as derivation trees of Hoare triples. However, we choose a lighter (and less precise) presentation, by interleaving assertions ($\{P\}$) and program instructions. More details about the specification of one instructions are given when needed.

The proof of the previous specification can be found in Figure 15, $\lightning$ denoting the use of the frame rule.

A more detailed, similar proof can be found in [14], §6.

1.2.2 Using `copytree` on DAGs

The specification we just gave is not as general as possible: because `copytree` just explores recursively its input, it also runs on a tree with sharing: a DAG (directed

acyclic graph) (however, it does not on graphs, because cycles would make him loop while allocating more and more memory).

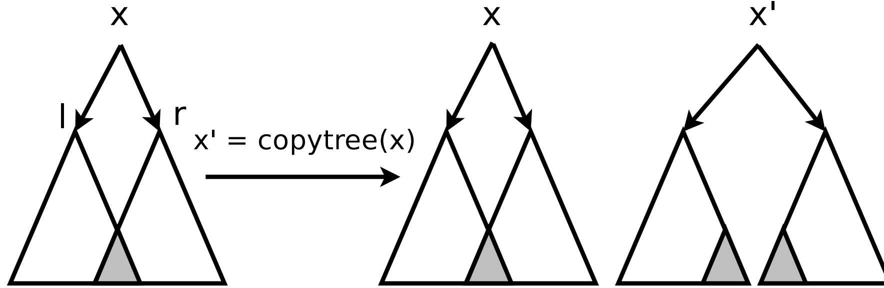


Figure 1: Calling `copytree` on a DAG.

The copied tree is indeed a plain tree, as `copytree` does not observe - nor preserve - sharing. As illustrated on figure 1, the shared parts (in grey) are duplicated in the output tree.

To write a specification accounting for this behavior, one need to define a "dag x" predicate. We can try to mimic the one for trees: similarly to trees, there are no cycles in a dag, so we can keep the first $*$: the root node shall be separated from the subdags. However, the second $*$ cannot be reused: we need a new operator to express the fact that the two subdags share *some* part of the heap (in grey), which was not the case with tree.

Separations logic often introduce, to this purpose, an operator called *overlapping conjunction*, $\circledast$. An formula "$P_1 \circledast P_2$" describes a heap composed of two subheaps, sharing some memory cells, one satisfying P_1, the other satisfying P_2 - as illustrated in Figure 2.

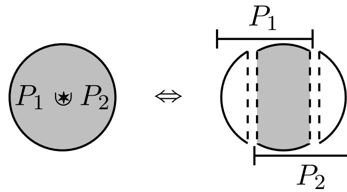
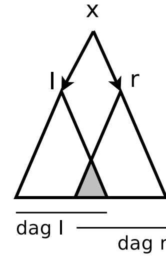


Figure 2: Overlapping conjunction.

The $s, h \models P$ relation is extended with the following rule, h_2 corresponding to the middle grey subheap of figure 2:

$$s, h \models P_1 \circledast P_2 \quad \Leftrightarrow \quad \exists h_1, h_2, h_3 : s, h_1 \oplus h_2 \models P_1 \text{ and } s, h_2 \oplus h_3 \models P_2$$

Then, "dag x" can be inductively defined by:



$$\text{dag } x \equiv \exists val, l, r : x \mapsto val, l, r * (\text{dag } l \uplus \text{dag } r)$$
$$\vee (x = 0 \wedge \texttt{emp})$$

A first attempt to design a new specification using dag, by mimicking the previous one, would give something similar to:

$$\{\text{dag } x\} \; \texttt{x' = copytree(x)} \; \{\text{dag } x * \text{tree } x'\}$$

However, if we try to prove this specification, we encounter a problem. When dealing with the first recursive call, we would like to be able to prove the following Hoare triple, to get an easy proof, similar to the previous one:

$$\{x \mapsto l, r * (\text{dag } l \uplus \text{dag } r)\}$$
$$\texttt{tree* l' = copytree(x->l)}$$
$$\{x \mapsto l, r * (\text{dag } l \uplus \text{dag } r) * \text{tree } l'\}$$

In the previous cases, all the predicates were separated by $*$, so we were able to apply the frame rule. Here, we would like to use some kind of frame rule to isolate only the dag l bit we want. However, we are not able to use the frame rule here, because of the presence of $\uplus$ instead of $*$. This is for good reasons: the induction hypothesis (our specification), is not strong enough - it does not imply that the recursive call preserves the subdag it is called on.

Some `evil_copytree` function could indeed erase the input dag and satisfy the same specification (an empty dag is still a valid dag):

$$\{\text{dag } x\} \; \texttt{x' = evil_copytree(x)} \; \{\text{dag } x * \text{tree } x'\}$$

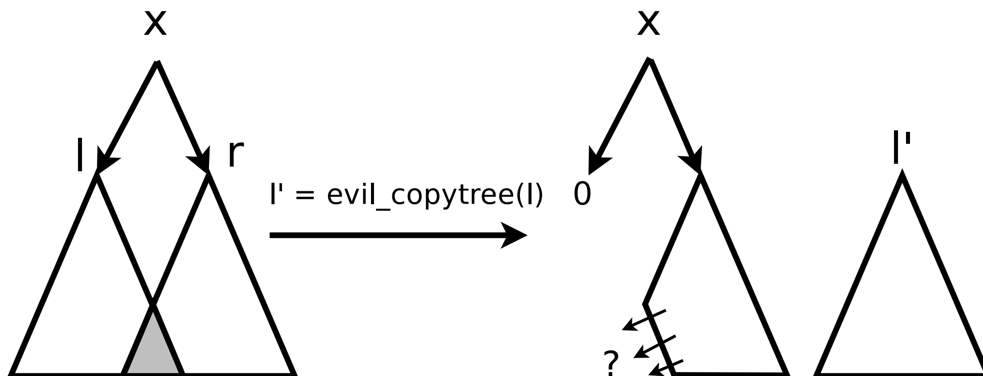


Figure 3: Heap after the first recursive call of `evil_copytree`.

In the case of `evil_copytree`, after the first recursive call, the structure is no longer a valid structure, because of the dangling pointers from the old right subdag to the - now deleted - shared part (Figure 3).

More generally, this specification is too weak to imply that, after the first recursive call on "dag l", we still have "dag r", and that we can still fold back the structure to a dag.

State of the art: In the same paper where he describes these difficulties ([14]), Reynolds gives some idea for a solution: use an *assertion variable* that quantifies over properties of the heap.

$$\{p \wedge \text{dag}\ \tau\ x\}\ \texttt{x' = copytree(x)}\ \{p * \text{tree}\ \tau\ x\}$$

Indeed, as the initial heap is not modified by `copytree`, every property p holding for this heap remains true in the postcondition - separated from the copied tree. This solution has a taste of second order logic, which is hard to handle for an automatic tool, and seems a bit overkill for such a simple program.

Another solution comes from Hobor and Villard [11]: it requires using a much more precise dag predicate, and then proving functional correctness of the program. They use an alternative rule, the *ramification rule*, to obtain multiples lemmas to be proved in the metatheory (using semantic reasoning).

Both solutions parametrize the `tree` and dag predicates by mathematical structures representing the content of the in-memory structure (the τ parameters). This is quite useful when doing manual proofs, as it lifts some reasonings to the metatheory where doing proofs is easier. However, manipulating them using an automatic tool is not always that easy: we would like to not use them, and have less precise, more simple predicates.

The aim of this internship is to try to answer the question: can we do better, by modifying our separation logic? We will need to find something in the middle: as seen before, the "most simple" specification does not work, so we still need a more expressive logic.
This would mean modify the assertions logic, and/or modify the set of Hoare rules.

A first grasp of what will be the main idea: as we want to talk about preserving parts of the heap, we will name them, using *labels* (think *heap variables*).

2 Extending separation logic: hybrid separation logic

2.1 Definitions

Following the ideas of HyBBI [4], hybrid separation logic extends usual separation logic by adding the possibility to name memory heaps in formulas. Each name (or **label**) will describe only one precise heap.

Formally, the syntax of assertion formulæ is extended:

Definition 6 (Hybrid separation logic formulæ syntax)

$$
\begin{array}{rlr}
P, Q \text{ (hybrid formulæ) } ::= & \text{separation logic formulæ} & \\
\mid & \ell & (\textit{heap variables} \text{ or } \textit{labels}) \\
\mid & @_\ell A & (@\text{-modality}) \\
\mid & \exists \ell : A & (\exists\text{-quantified labels})
\end{array}
$$

Semantically, the $\models$ relation is now also parametrized by a valuation ρ : Labels $\rightharpoonup$ Heaps, and extended with:

Definition 7 (Hybrid formulæ semantics)

$$
\begin{array}{lcl}
s, h \models_\rho \ell & \Leftrightarrow & h = \rho(\ell) \\
s, h \models_\rho @_\ell A & \Leftrightarrow & s, \rho(\ell) \models_\rho A \\
s, h \models_\rho \exists \ell : A & \Leftrightarrow & \text{exists } h_\ell \text{ heap st. } s, h \models_{\rho[\ell \to h_\ell]} A
\end{array}
$$

The semantics of standard separation logic formulæ are not affected by ρ, which is just recursively propagated.

A label ℓ is valid for only one heap, $\rho(\ell)$; $@_\ell A$ forgets about the current heap and queries for validity of A on the heap corresponding to ℓ. We also allow existential quantified labels, for the purpose of introducing fresh names.

The deduction rules for Hoare triples are the same as in standard separation logic. No additional rule was needed in our use cases, and we believe that it should be the case for other examples as well.

2.2 The hybrid separation logic toolbox: useful remarks & lemmas

Let us now explore the possibilities of hybrid separation logic by going through various lemmas that will prove useful, and illustrative examples.

2.2.1 Hybrid formulæ

Characterization of $\circledast$ In hybrid separation logic, the overlapping conjunction "$P \circledast Q$" can be characterized using a formula of the logic that does not use $\circledast$:

$$P \circledast Q \Leftrightarrow \exists d, a, b, u, v : (a * d * b) \wedge @_u(a * d) \wedge @_v(d * b) \wedge @_u P \wedge @_v Q$$

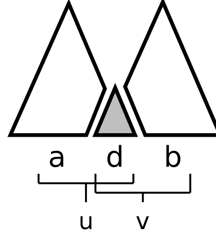


Figure 4: Characterizing $\circledast$ using $*$ and labels.

Fresh labels are introduced to name the different parts of the heap, then grouped accordingly and associated to the predicates P and Q using @-modalities.

Such characterization is quite useful: it allows us to express specific reasonings about $\circledast$ as a combination of more general lemmas talking only about labels and @s.

Interactions between ℓ and $@_\ell$ We have this rather intuitive lemma, that we will use extensively:

> **Lemma 1**
> $$\ell \wedge @_\ell P \Rightarrow P$$

The reverse implication is not always true: multiple heaps may satisfy P, while u denotes one precise heap.

$$P \wedge @_u P \not\Rightarrow u$$

Thus, the reverse holds only when P is a *strictly exact* assertion (Reynolds [14], Yang [17]). P is strictly exact iff for all s, h, h', ρ, $(s, h \models_\rho P)$ and $(s, h' \models_\rho P)$ implies $h = h'$. One example of strictly exact assertion is a single label, the $*$-conjunction of multiple labels, a memory cell $x \mapsto y$, or even `emp`.

On the opposite, a trivially non exact formula is $\top$, true for every heap, or $\exists y : x \mapsto y$ for example.

Consequently, we have the following lemma, that we will use in our future proofs:

> **Lemma 2** If P strictly exact,
> $$P \wedge @_\ell P \Rightarrow \ell$$

2.2.2 Hybrid formulæ in Hoare triples

Hoare triple validity As a remainder,

Definition 8

$\{P\}$ c $\{Q\}$ is valid if for all h, s, ρ such that $s, h \models_\rho P$, if the command c runs without errors, giving store and heap s', h', then $s', h' \models_\rho Q$.

Some examples

- $\{\ell\}$ `skip` $\{\ell\}$ is valid;
- $\{\ell \wedge @_\ell(y \mapsto v)\}$ `x = *y` $\{\ell\}$ is also valid (`x = foo` modifies the store s, not the heap);
- $\{\ell \wedge @_\ell(x \mapsto 4)\}$ `*x = 3` $\{\ell\}$ is not;
- $\{\ell\}$ `*x = 3` $\{\ell\}$ is not either: a free label is implicitly universally quantified.

What is interesting here is that we can reuse labels from the precondition in the postcondition, allowing to express the intuition that "this part of the heap hasn't changed". Moreover, free labels are implicitly universally quantified, because of the definition of validity of a Hoare triple, where ρ is (meta-)universally quantified.

$\exists$-quantifiers, in practice In practice, $\exists$-quantified labels are used for the only purpose of introducing fresh names: $\exists$ are prenex in this case, and can be easily eliminated using a standard Hoare rule.

$$\text{Exists}\ \frac{\{P\}\ \mathsf{c}\ \{Q\}}{\{\exists x.P\}\ \mathsf{c}\ \{\exists x.Q\}}$$

The consequence is that in practice, we never need to explicitly manipulate formulas with $\exists$.

The propagation lemma This lemma is rather intuitive: it lives on the idea that some $@_\ell P$ modality do not talk about the current heap, but about the one corresponding to ℓ. Thus, as long as the ρ valuation is the same, the $@_\ell P$ assertions holds. All this leads to the following lemma:

Lemma 3 (Propagation lemma) $\{A \wedge @_\ell P\}\ \mathsf{c}\ \{B\} \Rightarrow \{A \wedge @_\ell P\}\ \mathsf{c}\ \{B \wedge @_\ell P\}$

Consequently, as soon as we know some fact "$@_\ell P$", we can freely add it in the postcondition if needed: ρ, assigning heaps to labels is still the same after running c.

Using the propagation lemma in proofs Thanks to the propagation lemma, when writing assertions in proofs, we do not need to repeat the "$@_\ell P$" facts: they propagate.

Thus, we can draw a distinction between the components of an assertion formula: a part of it describes the current heap: we call it the *spacial part* of the formula. The other part is composed of the propagating @ modalities.

$$\underbrace{u * v * x \mapsto y}_{\text{spacial part}} * \underbrace{@_u \text{tree } x * @_v A}_{\text{pure facts (propagate)}}$$

Figure 5: The two parts of an example formula.

Then, when writing proofs using hybrid separation logic, we will usually only write the spacial part of the formula, and write the pure facts only when we introduce them.

Equipped with this "hybrid separation logic toolbox", we will now try to demonstrate the usefulness of such logic, by walking through different examples of programs to specify. We hope that the behaviors and difficulties they present and how we handle them will be representative and useful for other cases.

3 Hybrid separation logic in practice: case studies

3.1 `copytree` on DAGs, revamped

The problem with the previous specification was that it was too weak to imply that copytree doesn't modify the input dag. If we know that the input dag is left untouched, then we can, after the first recursive call, recover the shared part and make the second call.

We saw that a label can be used in the precondition and in the postcondition of a Hoare triple. This is a good way to express the fact that some heap is preserved: a label describes only one precise heap. Using this idea, we get the specification of Figure 6.

$$\{\ell \wedge @_\ell \text{dag } x\} \texttt{ x' = copytree(x) } \{\ell * \text{tree } x'\}$$

Figure 6: New specification for `copytree`.

Remark: we write only ℓ in the postcondition. ℓ by itself is of little use; however, if needed, we can use the propagation lemma to get $@_\ell \text{dag } x$ in the postcondition as well. We see here that another consequence to the propagation lemma is simpler specifications.

The main axes of the proof are drawn in Figure 16.

Figure 17 gives a detailed proof of the Hoare triple for the first recursive call. Successive preconditions then postconditions are implied (from top to bottom), this way denoting the use of the consequence rule of Hoare logic. The assertions are presented in two parts, the left part being the spatial description of the heap, the right part holding @-modalities, that propagate thanks to Lemma 3.

The proof for the second recursive call is similar.

3.2 Modifying the content of the input: `mark`

The example of `copytree` was actually a bit simple. As the `copytree` program does not modify its input, it was a perfect fit for labels, that easily state what does not change. However, many programs want to actually modify their input data structures.

Consequently, we study here the case of `mark`, a program that recursively marks the nodes of a dag, changing the val field of each node to 1 (code in Figure 19).

An important remark about `mark` is that it modifies the heap, but preserves its shape. The l and r pointers of each node are preserved, so the final heap will cover the same memory locations as the initial one.

We can reflect this kind of property to labels, by defining a new "same region" operator, "$\sim$":

Definition 9 ("Same region" operator)

For all labels ℓ, ℓ', heap h, store s, valuation ρ,

$$s, h \models_\rho \ell \sim \ell' \ \textbf{ if } \ \mathrm{dom}(\rho(\ell)) = \mathrm{dom}(\rho(\ell'))$$

For example, the following formulas are valid:

$$\begin{array}{lcl}
@_\ell(x \mapsto 3), @_{\ell'}(x \mapsto 4) & \Rightarrow & \ell \sim \ell' \\
@_\ell(x \mapsto 3) * (y \mapsto 4), @_{\ell'}(x \mapsto 3) & \Rightarrow & \ell \not\sim \ell' \\
 & & \ell \sim \ell \\
@_u a * b, @_v a' * b' & \Rightarrow & a \sim a' \wedge b \sim b' \Rightarrow u \sim v
\end{array}$$

We could extend this definition to allow $\sim$ to act on any strictly exact assertion (we just need a way to associate to an assertion **the** heap it describes). In practice, we just want to use $\sim$ on conjunctions of labels, so we will consider this as an abuse of notation (we could labels to name the conjunctions, using $\exists$...).

Using this "notation", we can now write for example:

$$a \sim a' \wedge b \sim b' \Rightarrow a * b \sim a' * b'$$

Remark 1

- $\sim$ is an equivalency relation.
- as a "$\ell \sim \ell'$" kind of assertion does not talk about the current heap, it'll count as a pure fact that propagates, just like @ modalities.

Thanks to this new definition, we can write a specification for `mark` in Figure 7.

$$\{\ell \wedge @_\ell \text{dag } x\}\ \texttt{mark(x)}\ \{\exists \ell' : \ell' \wedge @_{\ell'} \text{dag } x \wedge \ell \sim \ell'\}$$

Figure 7: Specification for `mark`.

As for `copytree`, we will give the proof outline in Figure 19, and then focus on the proof of the Hoare triple for the first recursive call.

What is happening here? The initial heap can be divided into three parts: the "left" and "shared" parts compose the left subdag, while the "shared" and "right" parts compose the right subdag. Then, each recursive call updates two parts of the heap: the first recursive call updates the left part and the shared part, $u = d_l * d$, giving $u' = d_l' * d'$. Because only the content of the nodes has changed, we have $u \sim u'$.

Then, the second recursive call updates the shared part and the right part, $v' = d' * d_r$, giving $v'' = d'' * d_r'$, with $v' \sim v''$.

Now, can we conclude, and obtain the postcondition $\{\exists \ell' : \ell' \wedge @_{\ell'} \text{dag } x \wedge l \sim l'\}$?

We know that $@_{u''}(d_l' * d'') \wedge @_{v''}(d'' * d_r') \wedge\ @_{u''}\text{dag } l \wedge @_{v''}\text{dag } r$, so by introducing a new name ℓ', representing the heap corresponding to $d_l' * d'' * d_r' * x \mapsto 1, l, r$, we have "$\exists \ell' : \ell' \wedge @_{\ell'} \text{dag } x$".

We also know that $u \sim u'$ and $v' \sim v''$, i.e. $d_l * d \sim d_l' * d'$ and $d' * d_r \sim d'' * d_r'$. Then,

$$d_l * d * d_r \sim d_l' * d' * d_r \sim d_l' * d'' * d_r'$$

Moreover, $x \mapsto val, l, r \sim x \mapsto 1, l, r$, so $l \sim l'$, which concludes the proof.

We can now focus on the first recursive call, `mark(l)`. Proving its Hoare triple will require an auxiliary lemma, whose purpose is to somehow recover the right subdag after modifying the left one.

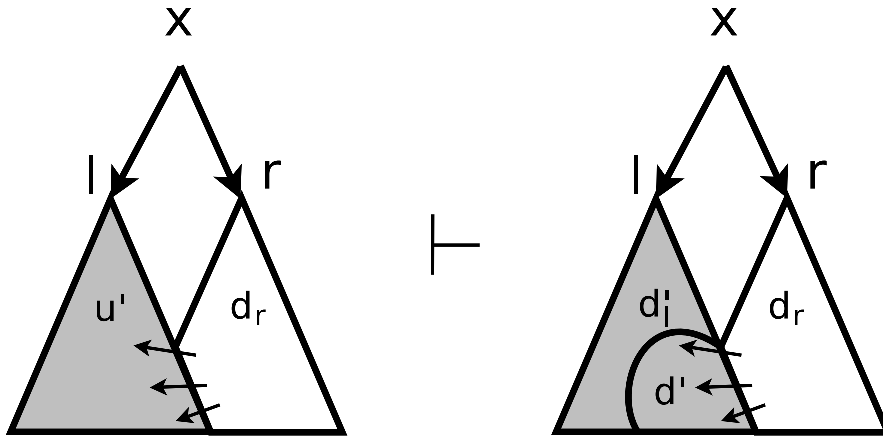


Figure 8: The auxiliary lemma, pictured.

The rightmost part, d_r, used to have pointers into the now-gone shared part. However, because the regions are preserved, we know they point somewhere into a dag (u'). If we name d' the part of the heap reachable from these pointers, it contains no cycles - thus $d_r * d'$ must be a dag.

The lemma is as follows:

Lemma 4 (Auxiliary lemma)

$$@_u(d_l * d) \wedge @_u \text{dag}\, l \wedge @_v(d * d_r) \wedge @_v \text{dag}\, r \wedge u \sim u' \wedge @_{u'} \text{dag}\, l$$
$$\vdash \exists d', d'_l, v' : @_{u'}(d'_l * d') \wedge @_{v'}(d' * d_r) \wedge @_{v'} \text{dag}\, r$$

This lemma describes more situations that what actually happens: it allows `mark` to change the structure of the dag laying in u', as long as it's still a dag. In practice, `mark` does not change dag pointers, but this lemma is enough for the induction to hold.

The detailed proof, using this lemma, follows in Figure 20.

3.3 Modifying the shape of the input: `spanning`

In this third example, we will proof a specification for a program that not only modifies the content of its input, but also its shape. Indeed, `spanning` computes the spanning tree of a dag, modifying it in place to remove superfluous edges.

Remark 2 We could have proved a version of spanning computing the spanning tree of a *graph*. However it would add a lot of bureaucracy to the proof in order to handle cycles, while not requiring new key ideas, so we settled on dags instead.

While walking through the dag and removing edges, `spanning` marks nodes it explores: at each moment of the computation, a part of the structure will be composed of unmarked, still unprocessed nodes; the other part, composed of marked nodes, has already been explored by the algorithm and contains tree fragments.

Initially, `spanning` receives as input a dag of unmarked nodes, and because it is a recursive program, some nodes of this dag may point to marked nodes. Then, `spanning` will mark and compute the spanning tree for the unmarked part, while preserving the marked part (as soon as `spanning` walks to a marked nodes, it stops and returns).

The result is a heap holding a marked tree. The code of `spanning` is in Figure 21.

A first difficulty with `spanning` is that the input structure is different from what we saw before. It is not a dag: the dag-like unmarked part has pointers to some marked nodes. Moreover, we want to be able to name the heap beyond these marked nodes, while not precisely knowing what its composed of (the algorithm will not need to explore it): we need to express the fact that it is preserved by the algorithm.

We cannot use two predicates, one for each part (unmarked, marked) of the heap: the unmarked part has pointers to the marked part, so we cannot describe it separately using an inductive predicate. Thus, a single inductive predicate will have to describe the whole heap.

The key idea is then to parametrize this predicate by two labels, in order to name the two parts. This way, these two parts can be defined depending of each other, but still named separately using the label parameters.

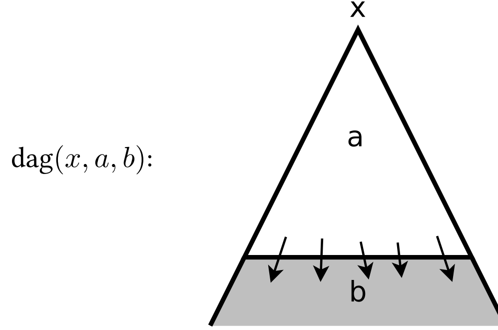


Figure 9: Shape of the input heap (marked nodes in grey).

As a trick to make the proof work nicely, we also allow the "b" part to contain arbitrary memory cells, as long as the nodes reachable from a are marked.

All of this gives us the following inductive predicate $\mathrm{dag}(x, a, b)$:

$$
\begin{aligned}
&\mathrm{dag}(x, a, b) \equiv \\
&\quad (x = 0 \wedge @_a \mathrm{emp} \wedge a \ast b) \vee \\
&\\
&\quad (\exists l, r, a', b', a'', b'' : \\
&\quad x \mapsto \mathbf{0}, l, r \ast \mathrm{dag}(l, a', b') \circledast \mathrm{dag}(r, a'', b'') \\
&\quad \wedge\ @_a x \mapsto 0, l, r \ast (a' \circledast a'') \\
&\quad \wedge\ @_b b' \circledast b'' \\
&\quad \wedge\ a \ast b) \vee \\
&\\
&\quad (\exists l, r : a \ast b \wedge @_a \mathrm{emp} \wedge @_b x \mapsto \mathbf{1}, l, r \ast \top)
\end{aligned}
$$

Figure 10: Inductive predicate for the `spanning` input structure.

The first case describes the situation where the dag is trivial; thus a is emp. In the second case - as illustrated in Figure 11 - the root is unmarked, and recursively points to two shared subdags. The a label describes the heap holding the root, and the unmarked parts of the subdags; while b describes the overlapping of the two marked parts of the subdags. In the third case, the root is marked: a is emp, and the root *plus arbitrary content* (described by $\top$) holds in b.

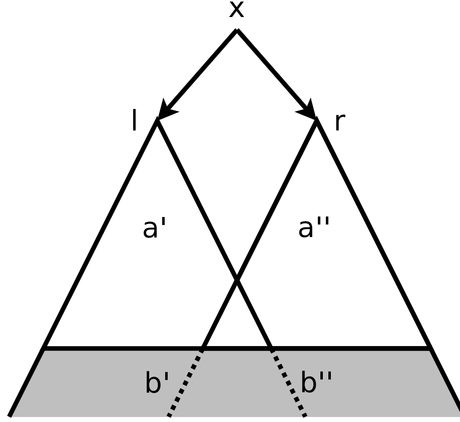


Figure 11: Shape of the heap in the recursive case of dag(x, a, b).

To get a provable specification, we need a last thing. Because in the proof we will care about whether a node is marked, we need to express that the final tree is marked. This is easy, defining a new "marked tree" inductive predicate:

$$\begin{aligned} \text{mtree } x \quad &\equiv \quad (\exists l, r : x \mapsto \mathbf{1}, l, r * \text{mtree } l * \text{mtree } r) \\ &\quad\ \vee (x = 0 \wedge \texttt{emp}) \end{aligned}$$

Then, the specification for `spanning` is as follows:

$$\{\text{dag}(x, a, b)\}\ \texttt{spanning(x)}\ \{\exists a' : a' * b \wedge @_{a'}\text{mtree } x \wedge a' \sim a\}$$

Let us now give the key ideas for the proof, leaving the details in Annex B.

First of all, the definition of our new dag trivially implies the following:

Lemma 5

$$\text{dag}(x, a, b) \Rightarrow a * b$$

The second idea is to, in some way, refactor the assertion obtained in the second inductive case of dag. It consists of applying Lemma 6, that can be found in the literature, associated to the notion of region [11], [15], [13].

Lemma 6

$$(a' * b') \circledast (a'' * b'') \Rightarrow (a' \circledast a'') * (b' \circledast b'')$$

if the heaps represented by a' and b'' are separated, and same for b' and a''.

In our case, we can apply it to the overlapping of the two subdags, as illustrated on Figure 12 (the red and green parts indicate the two assertions separated by an operator, on the left of the implication, $\circledast$, on the right, $*$).

Indeed, the definition of dag clearly states that a and b are separated $(\ldots \wedge a * b)$; then because a contains a' and b contains b'', a' and b'' are separated. Same goes for b' and a''.

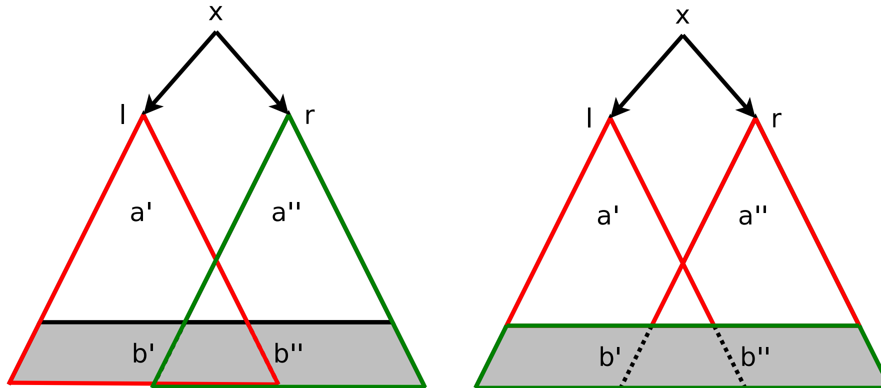


Figure 12: Two descriptions of a dag, thanks to Lemma 6.

After applying the lemma, the description of the heap we obtain is nicer: we can handle the unmarked $(a' \circledast a'')$ and the marked part $(b' \circledast b'')$ separately.

Then, the trick allowing us to continue after the first recursive call is to use the fact that the marked part of a dag$(\ldots)$ assertion is *extensible*. As illustrated in Figure 13, after calling `spanning` on the left subdag, we know that $@_{a'_1}$ mtree l, so we can just use dag$(r, d''_a, a'_1 * b'')$ as a precondition for `spanning(r)`. As the marked part is preserved, right after the second recursive call we will still have $a'_1 * b''$, plus the tree replacing d''_a.

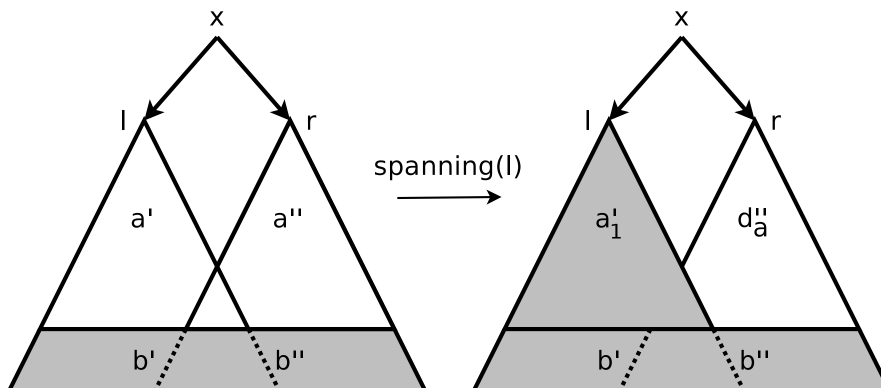


Figure 13: Recursive call of `spanning` on the left subdag.

Finally, a small auxiliary lemma is needed. Just like for `mark`, we will need an auxiliary lemma - this one being morally simpler that the one for `mark`. We need to express that, right after recursively calling `panning` on the left subdag, the remaining

unmarked bits of the right subdag, d''_a, still form a dag. Thus, we can obtain the required precondition for `spanning(r)`.

Lemma 7 (Auxiliary lemma)

$$@_{a'}d'_a * d_a \wedge @_{a''}d_a * d''_a \wedge @_v\text{dag}(x, a'', b'') \wedge a' \sim a'_1 \wedge @_{a'_1}\text{mtree}(l) \vdash$$
$$\exists w : @_w d''_a * a'_1 * b'' \wedge @_w\text{dag}(x, d''_a, b'' * a'_1)$$

Using all these four ideas, we can write a detailed proof of this specification for `spanning`. The proof in the non-trivial case can be found in Annex B.

4 Towards automatic reasoning on hybrid formulæ

Our general goal and motivator is the automation of program proofs. It is thus natural to wonder how hybrid separation logic can be dealt with using automatic tools. To this purpose, we tried different approaches; the two that proved somehow fruitful are detailed below.

4.1 Integration in `llStar`: proving `copytree`

The first approach consists of, quite pragmatically, trying to prove our specification with labels for `copytree` in an existing automatic prover, `llStar` [16]. The `llStar` prover is built on top of `coreStar` [3], and allow proving program specifications for programs that can be compiled to LLVM bitcode. In particular, we are able to write programs in C, just like in the examples, and compile them using `clang` before giving them to `llstar`.

The core logic of `coreStar` only knows about standard separation logic. However, `coreStar`'s flexible proof engine, which performs symbolic execution, allows the user to define new custom predicates and logic rules. We will use this feature to try to bake our reasonings on labels into custom rules.

The question is then to see, first, if we can prove some specification for `copytree` with this method, and then how hardcoded (or not) our custom rules will be.

After finally coming up with the set of rules written in Annex D (also available in `llStar` hg repository), the answer is "yes" and "not that much".

First, with this set of rules, `llstar` successfully proves the following specification for `copytree`: $\{\ell * @_\ell \text{dag}\ x\}$ `z = copytree(x)` $\{\ell * \text{tree}\ z\}$.

Remark 3 As it is in general difficult for automatic provers to handle both $*$ and $\wedge$ in assertions, we implicitly define that @-modalities always describe the empty heap.

This way, we can write $P * @_l Q \ldots$ instead of the $P \wedge @_l Q \ldots$ we are used to; which simplifies things a bit.

Let us now quickly step through the rules, without according too much time to their syntax.

The first group of rules encodes the definitions of tree and dag to rules unfolding the predicate to its definition (`tree_unfold_right`, `dag_unfold_under_at`)), or proving the predicate when given the unfolded bits (`dag_fold`). Only the rules that were needed for the proof are written here, but following the same idea more can be written, to cover more use cases.

Then, the second group contains rules following the ideas of Lemma 1 and Lemma 2: either inline the content of a $@_\ell$ if ℓ can be found, or prove ℓ from $@_\ell$ contents. The `inline_at_labels[2,4]` rules describe inlining of a conjunction of labels. Because `llStar` doesn't allow to talk about a n-ary conjunction of labels, we had to write a rule for each number of labels we needed; this may be improved by hacking into `llStar`. The `fold_at[2,4]` rules express the opposite operation, as in Lemma 2. However, written naively it would loop with `inline_at_labels`, so instead of replacing a conjunction of labels by a single label, we inline the content of the @, but in the right hand side of the sequent. The same idea holds for @s containing "pointsto".

Finally, we have some rules to do the basic reasonings, once everything possible is inlined thanks to the previous ones. A $@_a b$ is translated to a label equality; matching $@_l P$ with $@_u P$ tries to prove that $l = u$; and we instantiate fresh variables when needing them to equal something else.

What we can observe from these rules is that they are little specific to `copytree`: most of them handle labels and @-modalities in a generic way. It seems that such a set of rules can be a sane basis for proving other examples in `llStar`, and could be reused and extended quite easily.

4.2 Thoughts about handling hybrid formulæ in tools

The previously presented `llStar` rules (and even `llStar` rules in general), if sufficient for our example, cannot deal with arbitrarily complex formulæ (for example, multiple nested @s).

Consequently, we try here to encompass all the possibilities of hybrid assertions, and draw some ideas on how they could be handled by an automatic tool. We could not put these ideas into practice yet, but they could serve as a basis for a label-based solver.

Our general goal in this part is, given in hybrid formula, be able to process it in some way it structure becomes clearer, and know the shape of the heap it describes, the predicates it holds, if they share parts of the heap, etc.

4.2.1 Expressive power of hybrid formulæ

We can distinguish different possibilities offered by labels and @s, making the hybrid formulæ at the same time expressive and difficult to handle automatically.

- Sharing between predicates. We assume that our formulæ contain arbitrary predicates (it can be $x \mapsto \ldots$, tree x, etc. Then some predicates may share some part of the heap described by the formula, just as in $P \circledast Q$.

  Examples:

  - $(a * b * c) * @_u(a * b) * @_u P * @_v(b * c) * @_v Q$
  - $\ell * @_\ell P * @_\ell Q$

- Different descriptions of the heap. A formula holding $(a * b * c) \wedge \ldots$ describes the heap as being separated on three parts, at least. However, we can have multiple, different descriptions of the same heap, using different $@_\ell$ with the same ℓ.

  For example,

  $$g * @_g(a * b * c) * @_g(u * v) * \ldots$$

  Here, we know that the global heap can be separated in three parts, but also in two. Moreover, nothing requires these two descriptions to be compatible: they can really be two different visions of the same heap.

- Arbitrary number of indirections. It is possible to have an arbitrarily long chain of @s refining the knowledge on some label. For example, knowing that the toplevel holds the a label may not be the more precise description, if we know that $@_a(u * \ldots)$. But then, we can also have $@_u(w * \ldots)$, etc.

We would like to have a kind of *normal form* for hybrid formulæ, accounting of these specifics, and allowing to easily determine the shape of the heaps described by an assertion.

4.2.2 Standard form

We start by defining an intermediate form, the *standard form*.

> **Definition 10 (Standard form)** A hybrid formula in standard form is composed of:
>
> | | |
> |---|---|
> | $(u_1 * \ldots * u_n) *$ | A *toplevel* description of the heap |
> | $@_{a_1}(b_1^1 * \ldots * b_1^p) * \ldots * @_{a_m}(b_m^1 * \ldots * b_m^q) *$ | Various @s *between labels* |
> | $@_v P_1 * \ldots * @_w P_r$ | Labelled predicates, where every P_i does not contains any $*$ ("small" predicates) |

Putting an arbitrary formula into standard form is relatively simple. We do it by recursively walking through the formula and applying the following rewrite rules:

- Assign a label to every "small" predicate

$$\begin{array}{lll} P & \rightsquigarrow \ell * @_\ell P & l \text{ fresh variable} \\ P * Q & \rightsquigarrow (u * v) * @_u P * @_v Q & u, v \text{ fresh variables} \end{array}$$

- Simplify nested @s

$$@_\ell(\ldots * @_u(\ldots)) \quad \rightsquigarrow \quad @_\ell(\ldots) * @_u(\ldots)$$

An example of such rewriting is:

$$\ell * (x \mapsto y) * @_u(a * P) \quad \rightsquigarrow \quad \ell * v * @_v(x \mapsto y) * @_u(a * b) * @_b P$$

4.2.3 Canonical form

An assertion in standard form does not gives so much information. First, we do not know if the "toplevel" description of the heap is the more precise possible: some $@_a(b_1 \ldots b_n)$ may refine this description, if inlined into the "toplevel" part. Moreover, given a $@_u P$, because of all of these entangled @s in the second part, we cannot easily know if u can be found in the toplevel description, i.e. if P holds in the heap described by the formula.

Consequently, our goal is now to simplify the second part of the formula, to have the most precise heap description possible, and be able to easily lookup for satisfied predicates.

A first idea is to inline the @s on labels in the toplevel formula, following the rule:

$$(a * \ldots) * @_a(u * v) \quad \rightsquigarrow \quad (u * v * \ldots) * @_a(u * v)$$

However, such rewrite rule is ambiguous. For example, how should be handled the following assertion?

$$(a * \ldots) * @_a(u * v) * @_a(x * y * z) \quad \rightsquigarrow \quad ?$$

A solution is to simply compute each inlining possible, giving us, as a result, a list of "toplevel heap descriptions" - associated with the chosen @s. Eg, applied to our previous assertion, we get:

$$\begin{array}{lll} (a * \ldots) * @_a(u * v) * @_a(x * y * z) & \rightsquigarrow & [(u * v * \ldots) * @_a(u * v); \\ & & (x * y * z * \ldots) * @_a(x * y * z)] \end{array}$$

The list we get is a list of different "visions" of the heap: each formula describes the same heap, but in a different way - they will be combined by a $\wedge$. We call these *layers.*

However, this is not enough to have all the possibly useful heap descriptions. Formally, we just applied Lemma 1 everywhere, but sometimes we also want to use Lemma 2.

For example, consider the following assertion:

$$(a * b) * @_u(a * b) * @_u(c * d) * @_c P \dots$$

Applying the previous inlining procedure will not do anything; however, $(c * d)$ is also a valid description of the heap. Applying greedily Lemma 2 everywhere just like for Lemma 1 will not work, as it would undo all the inlinings of Lemma 1.

What we do is assume that what we want is, given some $@_\ell P$, know if ℓ can be found in some heap description. Then, for each $@_\ell P$, we follow @s indirections from ℓ to a toplevel description, and add all the intermediate toplevel descriptions found along the path.

Then, we obtain an assertion in canonical form:

Definition 11 (Canonical form) A formula $\mathcal{F}$ in canonical form is composed of:

$$\left(\bigwedge_i \text{layer}_i\right) * @_{\ell_1} P_1 * \dots * @_{\ell_n} P_n$$

where

$$\text{layer} ::= (u_1 * \dots * u_n) * @_{a_1}(b_1^1 * \dots * 1^p) * \dots * @_{a_m}(b_m^1 * \dots * b_m^q)$$

and we have the following properties:

- For each layer, all a_i are different
- For each layer, no @ can be inlined inside the toplevel description
- If $\mathcal{F} \Rightarrow P$, then exists i st. $\mathcal{F}$ contains $@_{\ell_i} P$, and exists some layer where: exists j st. $u_j = \ell_i$
  or exists $j_1 \dots j_k$ st. the layer contains $@_{\ell_i}(u_{i_1} * \dots * u_{i_k})$.

To wrap up, a formula in canonical form gives us easy access to the different descriptions of the heap, and allow to easily know the part of the heap, in term of labels, corresponds to some property. We believe that such canonical form can allow and simplify further reasonings and automation on hybrid formulæ, by removing the use to reason about Lemmas 1 and 2, as their use is baked into the canonizing algorithm.

A prototype implementation of the canonized algorithms informally described previously can be found at [9].

Conclusion

We have presented a new framework, as an extension of separation logic, for writing and proving program specifications. It introduces the notion of label, as a name allowing to talk about memory heaps in the logic of assertions. We gave a set of lemmas useful when manipulating assertions and specification using this new logic. We have demonstrated the applicability of our framework by proving specifications for various examples, presenting different programming patterns and difficulties. Finally, we successfully proved the specification of one of our previous examples using an automatic prover, and presented an algorithm canonizing the hybrid formulæ to a form easier to handle. This leads us to believe that hybrid separation logic constitutes an elegant extension of separation logic for reasonings about imperative programs and sharing in data structures.

References

[1] Andrew W. Appel, Robert Dockins, and Aquinas Hobor. Mechanized semantic library, 2009.

[2] Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In Gopalakrishnan and Qadeer [8], pages 178–183.

[3] Matko Botinčan, Dino Distefano, Mike Dodds, Radu Grigore, and Matthew J. Parkinson. coreStar: The core of jStar. In *BOOGIE*, pages 65–77, 2011.

[4] J. Brotherston and J. Villard. Parametric completeness for separation theories. In *POPL*, 2014.

[5] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *LNCS*, pages 459–465. Springer, 2011.

[6] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.

[7] Adam Chlipala, Paul Goverau, Gregory Malecha, Greg Morrisett, Aleks Nanevski, Avi Shinnar, Matthieu Sozeau, and Ryan Wisnesky. The ynot project.

[8] Ganesh Gopalakrishnan and Shaz Qadeer, editors. *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *LNCS*. Springer, 2011.

[9] Armaël Guéneau. Prototype source code. http://srv.isomorphis.me/proto.tar.gz.

[10] Guanhua He, Shengchao Qin, Chenguang Luo, and Wei-Ngan Chin. Memory usage verification using Hip/Sleek. In Zhiming Liu and Anders P. Ravn, editors, *ATVA*, volume 5799 of *LNCS*, pages 166–181. Springer, 2009.

[11] Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In *POPL*, pages 523–536. ACM, 2013.

[12] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.

[13] Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program analysis for overlaid data structures. In Gopalakrishnan and Qadeer [8], pages 592–608.

[14] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

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

[16] Jules Villard. `llStar` website. http://bitbucket.org/jvillard/llstar/.

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

[18] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *LNCS*, pages 385–398. Springer, 2008.

A Examples codes and proofs

```
typedef struct tree {
  int val;
  struct tree* l;
  struct tree* r;
} tree;

tree* copytree(tree* x) {
  if (x == NULL)
    return x;

  tree* l' = copytree(x->l);
  tree* r' = copytree(x->r);

  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
  x'->val = x->val;
  return x';
}
```

Figure 14: The copytree program.

```
// {ℓ ∧ @ℓ dag x}
tree* copytree(tree* x) {
  if (x == NULL)
    return x;
//  { ℓ  ∧ @ℓ(x ↦ val, l, r ∗ dℓ ∗ d ∗ dr)
//       ∧ @u(dℓ ∗ d) ∧ @u dag l
//       ∧ @v(d ∗ dr) ∧ @v dag r }
  tree* l' = copytree(x->l);
// {ℓ ∗ tree l'}

// {ℓ ∗ tree l'}
  tree* r' = copytree(x->r);
// {ℓ ∗ tree l' ∗ tree r'}

  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
  x'->val = x->val;
// {ℓ ∗ tree l' ∗ tree r' ∗ x' ↦ val, l', r'}
  return x';
}
// {ℓ ∗ tree x'}
```

Figure 16: Proof of `copytree` specification using dags.

```
// $\{\text{tree } x\}$
tree* copytree(tree* x) {
  if (x == NULL)
    return x;
// $\{x \mapsto val, l, r * \text{tree } l * \text{tree } r\}$
// $\lightning$ $\{\text{tree } l\}$
  tree* l' = copytree(x->l);
// $\lightning$ $\{\text{tree } l * \text{tree } l'\}$
// $\{x \mapsto val, l, r * \text{tree } l * \text{tree } r * \text{tree } l'\}$

// $\lightning$ $\{\text{tree } r\}$
  tree* r' = copytree(x->r);
// $\lightning$ $\{\text{tree } r * \text{tree } r'\}$
// $\{x \mapsto val, l, r * \text{tree } l * \text{tree } r * \text{tree } l' * \text{tree } r'\}$

  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
  x'->val = x->val;
// $\{x \mapsto val, l, r * \text{tree } l * \text{tree } r * x' \mapsto val, l', r' * \text{tree } l' * \text{tree } r'\}$
  return x';
}
// $\{\text{tree } x * \text{tree } x'\}$
```

Figure 15: Proof of `copytree` specification.

	spatial part	pure facts (propagate)
	ℓ	$\land @_l(x \mapsto l, r * d_\ell * d * d_r)$ $\land @_u(d_\ell * d) \land @_u \mathsf{dag}\ l$ $\land @_v(d * d_r) \land @_v \mathsf{dag}\ r$
Lemma 1 $\Rightarrow$	$x \mapsto l, r * d_\ell * d * d_r$	
Frame rule:	$\lightning\ d_\ell * d$	
Lemma 2 $\Rightarrow$	u	$\land @_u \mathsf{dag}\ l$
	`l' = copytree(x->l)`	
	$\lightning\ u * \mathtt{tree}\ l'$	
	$x \mapsto l, r * u * \mathtt{tree}\ l' * d_r$	
Lemma 1 $\Rightarrow$	$x \mapsto l, r * d_\ell * d * \mathtt{tree}\ l' * d_r$	
Lemma 2 $\Rightarrow$	$\ell * \mathtt{tree}\ l'$	

Figure 17: `copytree`: detailed proof for the first recursive call.

```
void mark(dag* x) {
  if (x == NULL)
    return;

  mark(x->l);
  mark(x->r);
  x->val = 1;
}
```

Figure 18: Code of the `mark` program.

```
// {ℓ ∧ @_ℓ dag x}
  void mark(dag* x) {
    if (x == NULL)
      return;
```

$$// \left\{ \begin{array}{ll} \ell & \wedge\ @_\ell(x \mapsto val, l, r * d_l * d * d_r) \\ & \wedge\ @_u(d_l * d) \wedge @_u \text{dag } l \\ & \wedge\ @_v(d * d_r) \wedge @_v \text{dag } r \end{array} \right\}$$

```
    mark(x->l);
```

$$// \left\{ \begin{array}{ll} \exists d_l', d', u', v' : d_l' * d' * d_r * x \mapsto val, l, r & \wedge\ @_{u'}(d_l' * d') \wedge @_{v'}(d' * d_r) \\ & \wedge\ @_{u'} \text{dag } l \wedge @_{v'} \text{dag } r \\ & \wedge\ u \sim u' \end{array} \right\}$$

```
    mark(x->r);
```

$$// \left\{ \begin{array}{ll} \exists d'', d_r', u'', v'' : d_l' * d'' * d_r' * x \mapsto val, l, r & \wedge\ @_{u''}(d_l' * d'') \wedge @_{v''}(d'' * d_r') \\ & \wedge\ @_{u''} \text{dag } l \wedge @_{v''} \text{dag } r \\ & \wedge\ v' \sim v'' \end{array} \right\}$$

```
    x->val = 1;
```

$$// \left\{ \exists d'', d_r', u'', v'' : d_l' * d'' * d_r' * x \mapsto 1, l, r \right\}$$

```
  }
```

Figure 19: Proof outline for `mark`.

	ℓ	$\wedge\ @_\ell(x \mapsto val, l, r * d_l * d * d_r)$ $\wedge\ @_u(d_l * d) \wedge @_u \text{dag}\ l$ $\wedge\ @_v(d * d_r) \wedge @_v \text{dag}\ r$
Lemma 1 $\Rightarrow$	$d_l * d * d_r * x \mapsto val, l, r$	
Frame rule:	$\lightning\ d_l * d$	
Lemma 2 $\Rightarrow$	$\quad u$	$\wedge\ @_u \text{dag}\ l$
	$\quad$ `mark(l)`	
	$\lightning\ \exists u' : u'$	$\wedge\ @_{u'} \text{dag}\ l \wedge u \sim u'$
	$\exists u' : u' * d_r * x \mapsto val, l, r$	
Aux. Lemma $\Rightarrow$	$\exists u', d', d'_l, v' :$ $u' * d_r * x \mapsto val, l, r$	 $\wedge\ @_{u'}(d'_l * d')$ $\wedge\ @_{v'}(d' * d_r) \wedge @_{v'} \text{dag}\ r$
Lemma 1 $\Rightarrow$	$\exists u', d', d'_l, v' :$ $d'_l * d' * d_r * x \mapsto val, l, r$	

Figure 20: `mark`: detailed proof for the first recursive call.

```
void spanning(dag* x) {
  if (x == NULL || x->val == 1)
    return;

  x->val = 1;

  if (x->l && !x->l->val)
    spanning(l);
  else
    x->l = NULL;

  if (x->r && !x->r->val)
    spanning(r);
  else
    x->r = NULL;
}
```

Figure 21: Code of the `spanning` program.

B Detailed proof of `spanning` specification

	$x \mapsto 1, l, r * \mathrm{dag}(l, a', b') \circledast \mathrm{dag}(r, a'', b'')$	$\wedge\, a * b \wedge @_a x \mapsto 0, l, r * (a' \circledast a'') \wedge @_b b' \circledast b''$
	$x \mapsto 1, l, r * u \circledast v$	$++\ @_u \mathrm{dag}(l, a', b') \wedge @_v \mathrm{dag}(r, a'', b'')$
Lemma 5 $\Rightarrow$	$x \mapsto 1, l, r * u \circledast v$	$++\ @_u a' * b' \wedge @_v a'' * b''$
Lemma 1 $\Rightarrow$	$x \mapsto 1, l, r * (a' * b') \circledast (a'' * b'')$	
Lemma 6 $\Rightarrow$	$x \mapsto 1, l, r * (a' \circledast a'') * (b' \circledast b'')$	
	$x \mapsto 1, l, r * d'_a * d_a * d''_a * d'_b * d_b * d''_b$	$++\ @_{a'}(d'_a * d_a) * @_{a''}(d_a * d''_a) *$ $@_{b'}(d'_b * d_b) * @_{b''}(d_b * d''_b)$
Lemma 2 $\Rightarrow$	$x \mapsto 1, l, r * a' * b' * d''_a * d''_b$	
Lemma 2 $\Rightarrow$	$x \mapsto 1, l, r * u * d''_a * d''_b$	
Lemma 1 $\Rightarrow$	$x \mapsto 1, l, r * \mathrm{dag}(l, a', b') * d''_a * d''_b$	
Frame rule :	$\lightning\ \mathrm{dag}(l, a', b')$ `spanning(l)` $\lightning\ \exists a'_1 : a'_1 * b' \wedge @_{a'_1} \mathrm{mtree}(l) \wedge a'_1 \sim a'$	
	$\exists a'_1 : x \mapsto 1, l, r * a'_1 * b' * d''_a * d''_b$	$++\ @_{a'_1} \mathrm{mtree}(l) \wedge a'_1 \sim a'$
Auxiliary lemma (7) $\Rightarrow$	$\exists a'_1, w : x \mapsto 1, l, r * a'_1 * b' * d''_a * d''_b$	$++\ @_w d''_a * a'_1 * b'' \wedge @_w \mathrm{dag}(x, d''_a, a'_1 * b'')$
Lemma 1 $\Rightarrow$	$\exists a'_1, w : x \mapsto 1, l, r * a'_1 * d'_b * d_b * d''_a * d''_b$	
Lemma 2 $\Rightarrow$	$\exists a'_1, w : x \mapsto 1, l, r * a'_1 * d'_b * b'' * d''_a$	
Lemma 2 $\Rightarrow$	$\exists a'_1, w : x \mapsto 1, l, r * w * d'_b$	
Lemma 1 $\Rightarrow$	$\exists a'_1, w : x \mapsto 1, l, r * \mathrm{dag}(r, d''_a, a'_1 * b'') * d'_b$	
Frame rule :	$\lightning\ \exists a'_1, w : \mathrm{dag}(r, d''_a, a'_1 * b'')$	
Exists rule :	$\mathrm{dag}(r, d''_a, a'_1 * b'')$ `spanning(r)` $\exists a''_2 : a''_2 * a'_1 * b'' \wedge @_{a''_2} \mathrm{mtree}(r) \wedge a''_2 \sim d''_a$	
	$\lightning\ \exists a'_1, w, a''_2 : a''_2 * a'_1 * b'' \wedge @_{a''_2} \mathrm{mtree}(r) \wedge a''_2 \sim d''_a$	
	$\exists a'_1, a''_2, w : x \mapsto 1, l, r * a''_2 * a'_1 * b'' * d'_b$	$++\ @_{a''_2} \mathrm{mtree}(r) \wedge a''_2 \sim d''_a$
Lemma 1 $\Rightarrow$	$\exists a'_1, a''_2, w : x \mapsto 1, l, r * a''_2 * a'_1 * d_b * d''_b * d'_b$	
	$\exists a'_1, a''_2, w : x \mapsto 1, l, r * a''_2 * a'_1 * (b' \circledast b'')$	
	$\exists a_1 : a_1 * (b' \circledast b'') \wedge @_{a_1} \mathrm{mtree}(x) \wedge a_1 \sim a \wedge @_b b' \circledast b''$	
	$\exists a_1 : a_1 * b \wedge @_{a_1} \mathrm{mtree}(x) \wedge a_1 \sim a$	

C Lemmas proofs

Lemma 1

$$\begin{array}{rl} & s, h \models_\rho \ell \wedge @_\ell P \\ \Rightarrow & h = \rho(l) \text{ and } s, \rho(l) \models_\rho P \\ \Rightarrow & s, h \models_\rho P \end{array}$$

Lemma 2

$$\begin{array}{rl} & s, h \models_\rho P \wedge @_\ell P \\ \Rightarrow & s, h \models_\rho P \text{ and } s, \rho(\ell) \models_\rho P \\ P \text{ strictly exact + definition} \Rightarrow & h = \rho(\ell) \\ \Rightarrow & s, h \models_\rho \ell \end{array}$$

Lemma 3 (propagation lemma)

$$\dfrac{\dfrac{\dfrac{\{A \wedge @_l P\}\ \mathtt{c}\ \{B\}}{\{A * (\mathtt{emp} \wedge @_l P)\}\ \mathtt{c}\ \{B\}}}{\{A * (\mathtt{emp} \wedge @_l P) * (\mathtt{emp} \wedge @_l P)\}\ \mathtt{c}\ \{B * (\mathtt{emp} \wedge @_l P)\}}\ \text{Frame}}{\{A \wedge @_l P\}\ \mathtt{c}\ \{B \wedge @_l P\}}$$

Lemma 4 (`mark` auxiliary lemma) Let $h_{d'}$ the part of the heap in $\rho(u')$ reachable from pointers of $\rho(d_r)$. Let $h_{d'_l}$ the heap $\rho(u') \backslash h_{d'}$.

Thus, $\rho(u') = h_{d'_l} \cup h_{d'}$. Let $h_{v'} = h_{d'} \cup \rho(d_r)$. $h_{v'}$ is the union of a part of a dag, pointing to parts of another dag: it only contains dag nodes pointing to each other, and contains no cycles. Thus, $h_{v'}$ holds a dag.

Now, let d', d'_l, v' be the labels such that $h_{d'} = \rho(d')$, $h_{d'_l} = \rho(d'_l)$, $h_{v'} = \rho(v')$.

Then, $@_{u'}(d'_l * d') \wedge @_{v'}(d' * d_r) \wedge @_{v'}\text{dag } r$.

Lemma 5 Each case of the definition of dag trivially implies $a * b$.

Lemma 6 Assume $h \models_\rho (a' * b') \circledast (a'' * b'')$. Then $\rho(a')$ is separated from $\rho(b')$, but also from $\rho(b'')$ by hypothesis. Thus, $\rho(a')$ may share memory cells only with $\rho(a'')$.

By the same reasoning, $\rho(a'')$ is separated from $\rho(b')$, $\rho(b'')$, and shares memory only with $\rho(a')$. Consequently, the heap can be separated in two disjoint parts, one holding $\rho(a')$ and $\rho(a'')$, the other $\rho(b')$ and $\rho(b'')$.

We can then conclude, $h \models_\rho (a' \circledast a'') * (b' \circledast b'')$.

Lemma 7 (`spanning` auxiliary lemma) Let $h_w = \rho(d''_a) \cup \rho(a'_1) \cup \rho(b'')$. $\rho(a'')$ contains a dag of unmarked nodes, with pointers to the marked nodes of $\rho(b'')$. d''_a contains a subdag of $\rho(a'')$, with pointers to $\text{dom}(\rho(d_a))$; as this region is composed of marked nodes ($@_{a'_1}\text{mtree}(l)$), $\rho(d''_a)$ can be considered as a dag, with $\rho(a'_1) \cup \rho(b'')$ as "marked part".

Introducing w as the label such that $h_w = \rho(w)$, we have $@_w d''_a * a'_1 * b'' \wedge @_w\text{dag}(x, d''_a, b'' * a'_1)$.

D llStar rules for proving copytree

copytree.star

```
predicate dag(lltype, i64);
predicate dagnode(lltype, i64, i64, i64);
predicate tree(lltype, i64);
predicate treenode(lltype, i64, i64, i64);

predicate here(label);
predicate at(label, bool);

import "../rules/llvm.logic";
import "../specs/safe_stdlib.spec";

rewrite dagnode:
   dagnode(lltype ?t, i64 ?x, i64 ?l, i64 ?r)
   -> pointer(i64 ?x, named("struct.node") { i32 _c, i64 ?l, i64 ?r })
;

rewrite treenode:
   treenode(lltype ?t, i64 ?x, i64 ?l, i64 ?r)
   -> pointer(i64 ?x, named("struct.node") { i32 _c, i64 ?l, i64 ?r })
;

/*
 * Rules encoding definitions of tree/dag
 */

rewrite dag_nil:
  dag(lltype ?t, NULL()) -> emp
;

rewrite tree_nil:
  tree(lltype ?t, NULL()) -> emp
;

rule nobacktrack tree_unfold_right:
 bool ?f | bool ?fl * i64 ?x != NULL()
          |-
          bool ?fr * tree(lltype ?t, i64 ?x)
if
 bool ?f | bool ?fl
          |-
          bool ?fr *
          treenode(lltype ?t, i64 ?x, i64 _l, i64 _r) *
          tree(lltype ?t, i64 _l) *
          tree(lltype ?t, i64 _r)
;

rewrite dag_unfold_under_at:
  !at(label ?l, dag(lltype ?t, i64 ?x)) * i64 ?x != NULL() * bool ?f
  ->
     !at(label ?l, dag(lltype ?t, i64 ?x)) *
     !at(label ?l,
         here(label _node) *
         here(label _a) *
         here(label _b) *
         here(label _c)) *
     !at(label _node, dagnode(lltype ?t, i64 ?x, i64 _l, i64 _r)) *
     !at(label _u, here(label _a) * here(label _b)) *
     !at(label _v, here(label _b) * here(label _c)) *
     !at(label _u, dag(lltype ?t, i64 _l)) *
     !at(label _v, dag(lltype ?t, i64 _r)) *
     bool ?f
```

```
;

rule nobacktrack dag_fold:
 bool ?f | here(label ?node) *
           here(label ?a) *
           here(label ?b) *
           here(label ?c) *
           !at(label ?node, dagnode(lltype ?t, i64 ?x, i64 ?l, i64 ?r)) *
           !at(label ?u, here(label ?a) * here(label ?b)) *
           !at(label ?v, here(label ?b) * here(label ?c)) *
           !at(label ?u, dag(lltype ?t, i64 ?l)) *
           !at(label ?v, dag(lltype ?t, i64 ?r)) *
           bool ?fl
           |-
           bool ?fr *
           dag(lltype ?t, i64 ?x)
if
 bool ?f * dag(lltype ?t, i64 ?x) | bool ?fl |- bool ?fr
;


/*
 * Reasonings on @s and their contents: make the proof go forward.
 * Unfold/Fold @s.
 */

rule nobacktrack inline_at_labels2:
 bool ?f | here(label ?l) *
           !at(label ?l, here(label ?a) * here(label ?b)) * bool ?fl
           |-
           bool ?fr
if
 bool ?f | here(label ?a) *
           here(label ?b) *
           !at(label ?l, here(label ?a) * here(label ?b)) *
           bool ?fl
           |-
           bool ?fr
;

rule nobacktrack inline_at_labels4:
 bool ?f | here(label ?l) *
           !at(label ?l, here(label ?a) *
                         here(label ?b) *
                         here(label ?c) *
                         here(label ?d)) *
           bool ?fl
           |-
           bool ?fr
if
 bool ?f | here(label ?a) *
           here(label ?b) *
           here(label ?c) *
           here(label ?d) *
           !at(label ?l, here(label ?a) *
                         here(label ?b) *
                         here(label ?c) *
                         here(label ?d)) *
           bool ?fl
           |-
           bool ?fr
;

rule nobacktrack fold_at2:
 bool ?f | !at(label ^l, here(label ^l1) * here(label ^l2)) * bool ?fl
```

```
           |- here(label ^l) * bool ?fr
if
 bool ?f | !at(label ^l, here(label ^l1) * here(label ^l2)) * bool ?fl
           |- here(label ^l1) * here(label ^l2) * bool ?fr
;

rule nobacktrack fold_at4:
 bool ?f | !at(label ^l, here(label ^l1) *
                          here(label ^l2) *
                          here(label ^l3) *
                          here(label ^l4)) *
           bool ?fl
           |-
           here(label ^l) *
           bool ?fr
if
 bool ?f | !at(label ^l, here(label ^l1) *
                          here(label ^l2) *
                          here(label ^l3) *
                          here(label ^l4)) *
           bool ?fl
           |-
           here(label ^l1) *
           here(label ^l2) *
           here(label ^l3) *
           here(label ^l4) *
           bool ?fr
;

rule nobacktrack pointsto_in_at:
 bool ?f | here(label ?l) *
           !at(label ?l, pointer(i64 ?x, 'a ?y) * bool ?p) *
           bool ?fl
           |-
           pointer(i64 ?x, 'a ?z) *
           bool ?fr
if
 bool ?f * pointer(i64 ?x, 'a ?y) |
           here(label _ll) *
           !at(label _ll, bool ?p) *
           !at(label ?l, pointer(i64 ?x, 'a ?y) * here(label _ll)) *
           bool ?fl
           |-
           'a ?y = 'a ?z *
           bool ?fr
;

rule nobacktrack pointsto_outside_at:
 bool ?f | pointer(i64 ?x, 'a ?y) *
           !at(label ?l, pointer(i64 ?x, 'a ?y) * bool ?p) *
           bool ?fl
           |-
           here(label ?l) *
           bool ?fr
if
 bool ?f | bool ?fl *
           !at(label _lll, bool ?p)
           |-
           here(label _lll) *
           bool ?fr
;

rule nobacktrack label_emp:
  bool ?f | !at(label ^l, emp) * bool ?fl |- here(label ^l) * bool ?fr
if
```

```
 bool ?f * here(label ^l) | !at(label ^l, emp) * bool ?fl |- bool ?fr
;

/*
 * Basic reasonings on lemmas/@s
 */

rewrite at_label_eq:
 !at(label ?a, here(label ?b)) -> label ?a = label ?b
;

rule match_at:
 bool ?f | !at(label ^l, bool ?p) * bool ?fl |- !at(label ^u, bool ?p) * bool ?fr
if
 bool ?f | !at(label ^l, bool ?p) * bool ?fl |- label ^u = label ^l * bool ?fr
;

rule nobacktrack equal_fresh:
 bool ?f | bool ?fl |- ('a ?a = 'a ^u) * bool ?fr
with
 fresh 'a ^u in bool ?fl;
 fresh 'a ^u in bool ?f
if
 bool ?f | bool ?fl * 'a ?a = 'a ^u |- bool ?fr
;

/*
 * Bureaucraty
 */

rule nobacktrack remove_label:
 bool ?f | here(label ?l) * bool ?fl |- here(label ?l) * bool ?fr
if
 bool ?f * here(label ?l) | bool ?fl |- bool ?fr
;

rule nobacktrack remove_tree:
  bool ?f | bool ?fl * tree(lltype ?t, i64 ?x)
            |-
            bool ?fr * tree(lltype ?t, i64 ?x)
if
 bool ?f * tree(lltype ?t, i64 ?x) | bool ?fl |- bool ?fr
;

/*
 * The spec for copytree
 */

procedure copytree(i64 %x) returns (i64 %z)
  {here(label _l) * !at(label _l, dag(lltype "struct.node",i64 %x))}
  {here(label _l) * tree(lltype "struct.node", i64 %z)}
;
```