Labelled Proofs for Separation Logic with Arbitrary Inductive Predicates

Didier Galmiche[1] and Daniel Méry[2]

1,2 Université de Lorraine - LORIA, Campus Scientifique BP 239, Vandœuvre-lès-Nancy, France

Abstract

Separation Logic (SL) is a logical formalism for reasoning about programs that use pointers to mutate data structures. SL has proven itself successful in the field of program verification over the past fifteen years as an assertion language to state properties about memory heaps using Hoare triples. Since the full logic is not recursively enumerable, most of the proof-systems and verification tools for SL focus on the decidable but rather restricted symbolic heaps fragment. Moreover, recent proof-systems that go beyond symbolic heaps allow either the full set of connectives, or the definition of arbitrary predicates, but not both. In this work, we present a labelled proof-system called $\mathrm{GM}_{\mathrm{SL}}$ that allows both the definition of arbitrary inductive predicates and the full set of SL connectives.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases separation logic, inductive predicates, labelled proofs, cyclic proofs

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Separation Logic (SL) is a logic for reasoning about programs that use pointers to manipulate and mutate (possibily shared) data structures [10, 14]. It was mainly designed to be used in the field of program verification as an assertion language to state properties (invariants, pre- and post-conditions) about memory heaps using Hoare triples. Some problems about pointer management, such as aliasing, are notoriously difficult to deal with and SL has proven successful on that matter over the past fifteen years. Building upon the Logic of Bunched Implications (BI) [13], from which it borrows its spatial connectives $*$ ("star") and $-\!\!*$ ("magic wand"), SL adds the $\mapsto$ predicate ("points-to"), with $x \mapsto y$ meaning that y is the content of the memory cell located at address x. One of the most interesting feature of SL (and a significant part of its success) is its built-in ability for *local reasoning*, which allows program specifications to be kept tighter as they need not consider (or worry about) memory cells that are outside the scope of the program.

However, being able to specify tight and concise properties about memory heaps more easily would remain of a somewhat limited interest if such specifications could not be verified or proved. It is therefore very important to provide (preferably efficient) proof-methods and automated verification tools for SL and much effort has been put on that subject recently. However, the task is not trivial because although the quantifier-free fragment of SL is decidable, full SL is not [5, 6, 11]. Full SL is not even recursively enumerable, so that no proof-system for SL can be finite, sound and complete at the same time. The undecidability of SL entails that most of the existing proof-systems and verification tools consider only restricted (but usually decidable) fragments of SL, of which the *symbolic heaps* fragment [2] is the most popular. Unfortunately, since the multiplicative implication $-\!\!*$ has been left

out, the symbolic heaps fragment cannot express the properties about heap extension that most of the induction hypotheses used in the literature for proving properties about pointer manipulating programs require. Even without considering such formulas, the symbolic heaps fragment cannot express many useful properties about heaps (such as cross-split or partial determinism for example) that are used is ASL to distinguish classes of models and variants of the logic.

In Section 2 we recall the basic notions about the syntax and semantics of SL and illustrate its use as an assertion language to specify properties about mutable data structures. In Section 3 we discuss our motivations and related works. In Section 4 we introduce $\mathrm{GM}_{\mathrm{SL}}$, our labelled proof-system which combines both Brotherston's cyclic-proofs [4] with Hou & Gore & Tiu's labelled proof-system $\mathrm{LS}_{\mathrm{SL}}$ [9]. Like $\mathrm{LS}_{\mathrm{SL}}$ and unlike $\mathrm{Cyclist}_{\mathrm{SL}}$, $\mathrm{GM}_{\mathrm{SL}}$ supports the full set of SL connectives. Like $\mathrm{Cyclist}_{\mathrm{SL}}$ and unlike $\mathrm{LS}_{\mathrm{SL}}$, it also allows arbitrarily defined inductive predicates. We finally show that all the dedicated rules for list segments in $\mathrm{LS}_{\mathrm{SL}}$ are derivable in $\mathrm{GM}_{\mathrm{SL}}$, thus proving that $\mathrm{GM}_{\mathrm{SL}}$ is strictly more expressive than $\mathrm{LS}_{\mathrm{SL}}$.

2 Separation Logic

Separation Logic (SL) is a concrete model of the boolean variant of BI called Boolean BI (BBI) [11] in which worlds are pairs of memory heaps and stacks called *states*. There are many variants of SL. In this section we follow Reynolds's original presentation of SL [14] (which was called "Pointer Logic" back then) without the machinery of pointer arithmetic.

In Reynolds's presentation, the set of *values* Val is the set of integers. Val contains two disjoint subsets Loc and $Atoms$. Loc contains an infinite number of *locations* (addresses of memory cells), while $Atoms$ denote constants such as nil (which is always assumed to be present). Besides values, we need an infinite and countable set Var of *program variables*.

A *stack* (or *store*) $s : Var \to_{fin} Val$ is a finite total function that associates values to program variables and a *heap* $h : Loc \rightharpoonup_{fin} Val \times Val$ is a finite partial function that associates pairs of values to locations[1]. The heap the domain of which is empty is called the *empty heap* and is denoted ϵ. We respectively denote $Heaps$ and $Stacks$ the sets of all heaps and all stacks. A *state* is a pair (s, h) where s is a stack and h is a heap.

We use the notation $h_1 \# h_2$ to denote that the heaps h_1 and h_2 have disjoint domains. Heap composition $h_1 \cdot h_2$ is only defined when $h_1 \# h_2$ and is then equal to the union of functions with disjoint domains. Heap composition extends to states as follows:

$$(s_1, h_1) \cdot (s_2, h_2) = (s_1, h_1 \cdot h_2) \text{ iff } s_1 = s_2 \text{ and } h_1 \# h_2.$$

An expression e can either be a value v or a program variable x and is interpreted w.r.t. a stack s so that $[\![x]\!]_s = s(x)$ and $[\![v]\!]_s = v$.

The language of SL contains equality, two "points-to" predicates $\overset{1}{\mapsto}$ and $\overset{2}{\mapsto}$ (we shall often drop the superscripts to improve readability), the connectives of BI and the existential quantifier. It is defined as follows:

- $P ::= e \overset{1}{\mapsto} e \mid e \overset{2}{\mapsto} e_1, e_2 \mid e_1 = e_2$ where e, e_1 and e_2 are expressions,
- $F ::= P \mid \mathrm{I} \mid F * F \mid F \mathrel{-\!\!*} F \mid \top \mid \bot \mid F \wedge F \mid F \to F \mid F \vee F \mid \exists u.F$

As usual, negation $\neg F$ can be defined as $(F \to \bot)$. One could also define $\top$ as $(\bot \to \bot)$ instead of having it as primitive.

[1] For convenience, in this paper, we also work with heaps of the form $h : Loc \rightharpoonup_{fin} Val$.

The semantics of the formulas is given by a forcing relation of the form $(s,h) \models F$ that asserts that the formula F is true in the state (s,h), where s is a stack and h is a heap. It is also required that the free variables of F are included in the domain of s.

▶ **Definition 1.** The semantics of the formulas is defined as follows:

- $(s,h) \models e_1 = e_2$ iff $[\![e_1]\!]_s = [\![e_2]\!]_s$
- $(s,h) \models e \mapsto e_1$ iff $dom(h) = \{[\![e]\!]_s\}$ and $h([\![e]\!]_s) = \langle\, [\![e_1]\!]_s \,\rangle$
- $(s,h) \models e \mapsto e_1, e_2$ iff $dom(h) = \{[\![e]\!]_s\}$ and $h([\![e]\!]_s) = \langle\, [\![e_1]\!]_s, [\![e_2]\!]_s \,\rangle$
- $(s,h) \models \top$ always
- $(s,h) \models \bot$ never
- $(s,h) \models A \wedge B$ iff $(s,h) \models A$ and $(s,h) \models B$
- $(s,h) \models A \vee B$ iff $(s,h) \models A$ or $(s,h) \models B$
- $(s,h) \models A \to B$ iff $(s,h) \models A$ implies $(s,h) \models B$
- $(s,h) \models \mathrm{I}$ iff $h = \epsilon$
- $(s,h) \models A * B$ iff $\exists h_1, h_2.\ h_1 \# h_2,\ h_1 \cdot h_2 = h,\ (s,h_1) \models A$ and $(s,h_2) \models B$
- $(s,h) \models A \mathrel{-\!\!*} B$ iff $\forall h_1.\ if\ h_1 \# h$ and $(s,h_1) \models A$ then $(s, h \cdot h_1) \models B$
- $(s,h) \models \exists u.A$ iff $\exists v \in Val.\ ([\, s \,|\, u \mapsto v \,], h) \models A$

In the previous definition, the notation $[\, s \,|\, u \mapsto v \,]$ denotes the stack s' such that

$$s'(u) = v \text{ and } s'(x) = s(x) \text{ if } x \neq u.$$

As usual, an *entailment* $F \models G$ between formulas holds if and only if for all states (s,h), if $(s,h) \models F$ then $(s,h) \models G$. The formula F is valid in SL, written $\models F$, if and only if $\top \models F$, *i.e.*, for all states (s,h), $(s,h) \models F$. By the semantics of $\to$, we can relate the notions of entailment and validity as follows: $F \models G$ if and only if $\models F \to G$.

The actual use of SL is as an assertion language to state invariants, pre- and post-condition in Hoare triples [13]. For example, one can define the command $dispose(e)$ that deallocates a location (thus creating dangling pointers) by the axiom $\{\, P * \exists\, u.\, e \mapsto u \,\}\, dispose(e)\, \{\, P \,\}$ where u is not free in e. The ability to state low-level properties about memory states (such as *dispose*) and Hoare Logic programming axioms using SL's assertion language is already very useful, mainly because SL has built-in facilities for *local reasoning* that allows a program specification to do without cumbersome conditions about memory cells that are outside the program's footprint [13]. However, SL only achieves its full potential w.r.t. program verification when moving to high-level properties about data structures that are mutated by pointer-manipulating programs. Most of these data structures are inductive and can be expressed using SL's assertion language enriched with inductive predicates. For example, one can define an acyclic singly-linked list segment $ls(e_1, e_2)$ that starts at address e_1 and ends with a memory cell containing e_2 as follows:

$$ls(e_1,e_2) \stackrel{\text{def}}{=} (\, e_1 = e_2 \,\wedge\, \mathrm{I}\,) \,\vee\, (\, e_1 \neq e_2 \,\wedge\, \exists u.(\, e_1 \mapsto u \,*\, ls(u,e_2))\,)$$

Such a formula states that a memory heap corresponds to an empty list (a list with identical starting and ending points) if it is empty and corresponds to a non-empty list segment (with distinct starting and ending points e_1 and e_2) if it can be split into two disjoints heaps, one being the first node of the list segment located at address e_1 and pointing to address u, the second one corresponding to a list segment that starts at address u and ends with a memory cell containing e_2.

A fairly standard example of a high-level property about list segments is a property stating that the combination of a heap that represents a list segment $ls(x,x')$ with a disjoint

heap that represents a list segment $ls(x', y)$ should result in a heap that represents a list segment $ls(x, y)$. The corresponding entailment is the following:

$$(LC) \stackrel{\text{def}}{=} \quad ls(x, x') * ls(x', y) \models ls(x, y)$$

However, as intuitive and reasonable as it might seem, such a property is not valid in SL when ls represents acyclic list segments. The invalidity of (LC) comes from the fact that two acyclic list segments $ls(x, x')$ and $ls(x', y)$ can give rise to what is often called a *panhandle list*, *i.e.*, a list that contains a cycle after a possibly empty acyclic initial segment. A panhandle list occurs whenever y in the second list segment points to an address occurring in the first list segment.

In order to obtain a valid high-level property about concatenation of acyclic list segments, one needs to strengthen (LC) so as to prevent panhandle lists which leads to the following entailment:

$$(ALC) \stackrel{\text{def}}{=} \quad (ls(x, x') \wedge \neg((ls(y, y') \wedge \neg\mathrm{I}) \mathrel{-\!\!*} \bot)) * ls(x', y) \models ls(x, y)$$

The subformula $\neg((ls(y, y') \wedge \neg\mathrm{I}) \mathrel{-\!\!*} \bot)$ ensures that it is not impossible to extend the heap representing the first list segment $ls(x, x')$ with a non-empty list segment starting at address y, which by the semantics of $\mathrel{-\!\!*}$ implies that y cannot be an address occurring in $ls(x, x')$. Let us note that the entailment would not remain valid without $\neg\mathrm{I}$ enforcing the non-emptiness of $ls(y, y')$ since the non-emptiness of $ls(y, y')$ is what ensures that y is an (allocated) address.

Another interesting entailment using $\mathrel{-\!\!*}$ is the following:

$$(ALH) \stackrel{\text{def}}{=} \quad ls(x, y) \mathrel{-\!\!*} ls(x, z) \models ls(y, z)$$

This entailment expresses the fact that if the current heap can be extended with an acyclic list segment $ls(x, y)$ so as to represent an acyclic list segment $ls(x, z)$ then it currently represents an acyclic list segment $ls(y, z)$.

3 Motivation and Related Work

Our motivation in this paper is to discuss how to obtain a proof-system for SL with both the full set of connectives and the ability to define reasonably general arbitrary inductive predicates. We do so by proposing a labelled proof-system with inductive rules sets and the notion of cyclic proofs.

Although a cyclic proof-system treating general SL formulas appears in [3], our work is original for several reasons we now discuss. The proof-system used in [3] is the purely syntactic bunched sequent calculus LBI devised for standard intuitionistic BI in [12]. However, the work in [3] considers a classical (point-wise) semantics for the additive connectives and is therefore actually a contribution to the proof-theory of Boolean BI, not BI (the title of the paper is thus a bit misleading on this point). Using an intuitionistic proof-system with a classical semantics does not impair soundness, as any theorem of BI is also a theorem of BBI, but it severely and unnecessarily impairs completeness since any theorem of BBI that is not also a theorem of BI cannot be proved with the proof-system described in [3]. The paper also considers an ordinary (non-inductive) points-to predicate[2] which is then

[2] Let us remark that this points-to predicate is never actually precisely defined.

used to define a binary inductive predicate $ls(x, y)$ that captures (possibly cyclic) segments of singly-linked lists. While this approach is suitable as an illustration of how the notions introduced for Boolean BI could be further developed to handle the case of SL, it still only remains a somewhat rough and too general approximation that does not take into account all the subtleties of SL as a very specific concrete model of Boolean BI admitting a handful of key properties. One such key property is disjointness: $(x \mapsto y) * (x \mapsto z)$ can never hold of any heap as it would require that two disjoint heaps share the location x. Therefore, $((x \mapsto y) * (x \mapsto z)) \to A$ is valid in SL for any formula A, but the proof-system in [3] cannot prove it. Moreover, SL also comes in two flavours: an intuitionistic and a more widely used classical one. However, as opposed to what happens for BI w.r.t. BBI, there are valid formulas of intuitionistic SL that are not valid in classical SL, for instance $((x \mapsto y) * A) \to (x \mapsto y)$ is valid for any A in intuitionistic SL, but not in classical SL. We could not find in [3] any discussion on whether, in the particular case of SL, the presented intuitionistic proof-system is actually sound w.r.t. classical SL.

For all the reasons discussed in the previous paragraph, although [3] remains a significant contribution to the proof-theory of Boolean BI, its stretched use of a too general points-to predicate leaves its application to (concrete classical) SL unclear and unsatisfactory. The case of classical SL is handled specifically in [4]. For example, the proof-system presented in [4], which is implemented in a tool called $\text{Cyclist}_{\text{SL}}$, has dedicated rules for disjointness ($\overset{1}{\mapsto}$ and $\overset{2}{\mapsto}$). However, the logical fragment addressed in [4] is a significant restriction of the one in [3] as additive conjunction, additive implication and multiplicative implication (magicwand) are discarded. Although sometimes claimed straightforward, extending [4] to the full set of SL connectives would actually be quite difficult. It is wisely and clearly pointed out in [3] that the use of an intuitionistic sequent calculus with a classical semantics is not guided by philosophical reasons, but by technical ones as the definition of a multi-conclusioned bunched sequent calculus for Boolean BI would require the formulation of an appropriate notion of disjunctive multiplication (dual to BI multiplicative conjunction), something which is far from trivial. Indeed, so far and to our current knowledge, there exists no true Gentzen style[3] purely syntactic multi-conclusioned bunched sequent calculus for Boolean BI. In our opinion, trying to handle the full set of SL connectives in a purely syntactic sequent calculus is not the way to go, which is why we introduce a label-based proof-system.

The first proof-system for SL supporting the full set of SL connectives was our labelled tableau system T_{SL} [8]. T_{SL} uses labels that represents heaps and captures the properties of the heap model inside a graphical structure called the *resource graph* induced by a closure operator on labels w.r.t. a labelling algebra. Provability in T_{SL} relies on the two notions of structural and logical consistency. Structural consistency captures the various properties of the heap model and involves notions such as points-to distributions and measures of the size of (the domain of) a heap. For simplicity, [8] also considers a fragment of SL where only locations can occur on the left-hand side of a points-to predicate. In [9], Hou & Goré & Tiu introduce a labelled sequent calculus LS_{SL} with various dedicated proof rules[4]for two kinds of data structures: acyclic singly-linked list segments and binary trees. Without the rules for inductive predicates, LS_{SL} can be seen as a sequent-style reformulation of T_{SL} where structural aspects (resource graphs operations and points-to distributions) are translated into explicit structural and pointer rules, with minor refinements to handle heap extension and values on the left-and side of points-to predicates. One valuable contribution of [9] is an

[3] We do not count display or deep sequent calculi as true Gentzen style sequent calculi.
[4] Eight rules for acyclic singly-linked list segments, six rules for binary trees.

$$\frac{h\epsilon \rhd h; \mathcal{G}; \Gamma \vdash \Delta}{\mathcal{G} \vdash \Delta}\ \mathrm{U} \qquad \frac{h_3h_5 \rhd h_0; h_2h_4 \rhd h_5; h_1h_2 \rhd h_0; h_3h_4 \rhd h_1; \mathcal{G}; \Gamma \vdash \Delta}{h_1h_2 \rhd h_0; h_3h_4 \rhd h_1; \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{A}$$

$$\frac{h_2h_1 \rhd h_0; h_1h_2 \rhd h_0; \mathcal{G}; \Gamma \vdash \Delta}{h_1h_2 \rhd h_0; \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{E} \qquad \frac{\epsilon\epsilon \rhd h_2; \mathcal{G}[\epsilon/h_1]; \Gamma[\epsilon/h_1] \vdash \Delta[\epsilon/h_1]}{h_1h_1 \rhd h_2; \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{D}$$

$$\frac{\epsilon h_2 \rhd h_2; \mathcal{G}[h_2/h_1]; \Gamma[h_2/h_1] \vdash \Delta[h_2/h_1]}{\epsilon h_1 \rhd h_2; \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{Eq}_1 \qquad \frac{\epsilon h_2 \rhd h_2; \mathcal{G}[h_1/h_2]; \Gamma[h_1/h_2] \vdash \Delta[h_1/h_2]}{\epsilon h_1 \rhd h_2; \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{Eq}_2$$

$$\frac{h_1h_2 \rhd h_0; \mathcal{G}[h_0/h_3]; \Gamma[h_0/h_3] \vdash \Delta[h_0/h_3]}{h_1h_2 \rhd h_0; h_1h_2 \rhd h_3; \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{P} \qquad \frac{h_1h_2 \rhd h_0; \mathcal{G}[h_2/h_3]; \Gamma[h_2/h_3] \vdash \Delta[h_2/h_3]}{h_1h_2 \rhd h_0; h_1h_3 \rhd h_0; \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{C}$$

$$\frac{h_5h_6 \rhd h_1; h_7h_8 \rhd h_2; h_5h_7 \rhd h_3; h_6h_8 \rhd h_4; h_1h_2 \rhd h_0; h_3h_4 \rhd h_0; \mathcal{G}; \Gamma \vdash \Delta}{h_1h_2 \rhd h_0; h_3h_4 \rhd h_0; \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{CS}$$

$$\frac{\mathcal{G}[\epsilon/h_1, \epsilon/h_2]; \Gamma[\epsilon/h_1, \epsilon/h_2] \vdash \Delta[\epsilon/h_1, \epsilon/h_2]}{h_1h_2 \rhd \epsilon; \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{IU}$$

Side conditions:
Each label being substituted cannot be ϵ.
In A, the label h_5 does not occur in the conclusion.
In CS, the labels h_5, h_6, h_7, h_8 do not occur in the conclusion.

Figure 1 Structural rules in $\mathrm{GM}_{\mathrm{SL}}$.

implementation of the proof-system in a tool called Separata+ with a proof-strategy which guarantees termination when restricted to the symbolic heaps fragment. However, in our opinion, devising dedicated sets of rules to handle inductive predicates is an approach that is bound to show scalability problems given the great variety of data structures encountered in high-level properties and given the fact that even the most simple inductive data structure, namely lists, admit a huge amount of variants: singly-linked, doubly-linked, with or without cycles, *etc.* For example, since ls represents acyclic list segments in $\mathrm{LS}_{\mathrm{SL}}$, $\mathrm{LS}_{\mathrm{SL}}$ can prove the (ALC) entailment but cannot prove (LC), as it would otherwise imply the unsoundness of the system. It also cannot prove (ALH). On the other hand, while (LC) is provable in $\mathrm{Cyclist}_{\mathrm{SL}}$, neither (ALC), nor (ALH) are expressible "as is" syntactically in $\mathrm{Cyclist}_{\mathrm{SL}}$ as it lacks multiplicative implication and additive conjunction (although there are other ways to express the content of (ALC) using a more general $ls(x, y, z)$ inductive predicate).

4 The $\mathrm{GM}_{\mathrm{SL}}$ Proof-System

In this section we introduce $\mathrm{GM}_{\mathrm{SL}}$, our labelled proof-system with arbitrarily defined inductive predicates. The core of the $\mathrm{GM}_{\mathrm{SL}}$ labelled proof-system consists of the structural, logical and pointer rules depicted in Figures 1, 2 and 3 and can be viewed as an extension of Hou & Goré & Tiu's $\mathrm{LS}_{\mathrm{SL}}$ proof-system [9] without the rules for data structures. $\mathrm{LS}_{\mathrm{SL}}$ incorporates the graph relations of T_{SL} [8] directly into the sequents instead of maintaining a separate resource graph. Therefore, the sequents take the form $\mathcal{G}; \Gamma \vdash \Delta$. The Γ part contains only labelled formula $h : A$, where h is a label representing a heap and A is a SL formula. The $\mathcal{G}$ part contains only ternary relations $h_1h_2 \rhd h_0$ meaning that the heap h_0 can be split into two disjoint subheaps h_1 and h_2 (or conversely that combining the two heaps h_1 and h_2 yields the heap h_0).

Like $\mathrm{LS}_{\mathrm{SL}}$, $\mathrm{GM}_{\mathrm{SL}}$ has label and expression substitutions. *Label substitutions* are written $[h_1/h'_1, \ldots, h_n/h'_n]$ meaning that h'_i gets replaced with h_i. *Expression substitutions* are

mappings $[\, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n \,]$ from program variables to expressions meaning that x_i gets replaced with e_i. The result of applying an expression substitution θ to the expression e is written $e\theta$. Equality betweens expressions is handled via standard syntactic unification as in logic programming. Therefore, given pairs of expressions $E = \{\, (e_1, e'_1), \ldots, (e_n, e'_n) \,\}$, a *unifier* is an expression substitution θ such that $e_i\theta = e'_i\theta$. The *most general unifier* of E is defined as usual and written $mgu(E)$ when it exists. In order to simplify comparisons with $\mathrm{LS_{SL}}$, we consider a fragment where *nil* is the only constant.

The two logical rules $\overset{2}{=}_{\mathrm{L}}$ and $\overset{2}{=}_{\mathrm{R}}$, as well as the structural rule IU and the two pointer rules $\mapsto_{\mathrm{L}_6}$ and $\mapsto_{\mathrm{L}_7}$ are specific to $\mathrm{GM_{SL}}$ and do not appear in $\mathrm{LS_{SL}}$. The rules $\overset{2}{=}_{\mathrm{L}}$ and $\overset{2}{=}_{\mathrm{R}}$ capture the fact that equality does not depend on heaps. The properties of heap composition in SL (unit, associativity, exchange, disjointness, equality, partial determinism, cancellativity and cross-split) are explicity captured into the remaining structural rules[5]. The structural rule IU explicitly captures the fact that the empty heap is an indivisible unit for heap composition in SL. The pointer rule $\mapsto_{\mathrm{L}_6}$ states that there is only one way to split a heap h_0 having the address e_1 in its domain so that the first component of the splitting is the singleton heap the domain of which is e_1. The pointer rule $\mapsto_{\mathrm{L}_7}$ is a form of cross-split that captures the fact that whenever a heap h_0 admits a first splitting $h_0 = h_1 \cdot h_2$ with h_1 being the singleton heap $e_1 \mapsto e_2$ and a second splitting $h_0 = h_3 \cdot h_4$ with h_3 being the singleton heap $e_3 \mapsto e_4$ then, provided that e_1 is not the same address as e_3, h_0 has at least the two distinct addresses e_1 and e_3 in its domain and can thus be rearranged so that $h_0 = h_1 \cdot h_3 \cdot h_5$ for some (possibly empty) heap h_5, from which it follows that $h_2 = h_3 \cdot h_5$ and $h_4 = h_1 \cdot h_5$.

4.1 Inductive Definitions

We now follow [3] to extend $\mathrm{GM_{SL}}$ with inductive definitions in the spirit of Martin-Löf productions. Our definition of SL already contains the ordinary (non-inductive) predicates $\top$, I, $\bot$, $\overset{1}{\mapsto}$ and $\overset{2}{\mapsto}$. The intepretation of these n-any predicates as subsets of $(Heaps \times Val^n)$ are as follows: $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = Heaps$, $[\![\mathrm{I}]\!] = \{h \mid dom(h) = \emptyset\}$, $[\![\overset{1}{\mapsto}]\!] = \{(h, v_1, v_2) \mid dom(h) = \{v_1\} \text{ and } h(v_1) = v_2\}$, $[\![\overset{2}{\mapsto}]\!] = \{(h, v_1, v_2, v_3) \mid dom(h) = \{v_1\} \text{ and } h(v_1) = \langle\, v_2, v_3 \,\rangle\}$.
We enrich the language of SL with a (fixed) finite set of inductive predicate symbols $P_1, \ldots, P_n$ with arities $a_1, \ldots, a_n$. We write x as a shorthand for tuples $(x_1, \ldots, x_n)$. and denote π_i^n the ith projection function on n-tuples such that $\pi_i^n(x_1, \ldots, x_n) = x_i$.

▶ **Definition 2** (Inductive definition)**.** An *inductive definition* of an inductive predicate P is a set of *production rules* $C_1(\mathrm{x}_1) \Rightarrow P(\mathrm{x}_1), \ldots, C_k(\mathrm{x}_k) \Rightarrow P(\mathrm{x}_k)$ where $k \in \mathbb{N}$, x_1,...,x_k are tuples of variables of appropriate length to match the arity of P and $C_1(\mathrm{x}_1)$,...,$C_k(\mathrm{x}_k)$ are *inductive clauses* given by the grammar $C(\mathrm{x}) ::= P(\mathrm{x}) \mid \hat{F}(\mathrm{x}) \mid C(\mathrm{x}) \wedge C(\mathrm{x}) \mid C(\mathrm{x}) * C(\mathrm{x}) \mid \hat{F}(\mathrm{x}) \rightarrow C(\mathrm{x}) \mid \hat{F}(\mathrm{x}) \mathrel{-\!\!*} C(\mathrm{x}) \mid \forall x C(\mathrm{x})$ with $\hat{F}(\mathrm{x})$ ranging over all formulas in which no inductive predicates occur and whose free variables are contained in $\{\mathrm{x}\}$.

Each production rule $C_i(\mathrm{x_i})$ is read as a disjunctive clause of the definition of the inductive predicate P. As in [3], the use of $\hat{F}(\mathrm{x})$ on the left of implications in production rules is designed to ensure monotonicity of the inductive definitions and we suppose that we have a unique inductive definition for each inductive predicate.

An annotated production rule $C \overset{\mathrm{z}}{\Rightarrow} P(\mathrm{x})$ is a production rule such that gathering all the free variables in C and in x exactly results in the tuple z. The left and right part of a production rule are respectively called its *body* and its *head*. An inductive definition is said

[5] Those properties are captured as a closure operator on labels in $\mathrm{T_{SL}}$.

$$\frac{}{\mathcal{G};\Gamma;h:A\vdash h:A;\Delta}\ \mathrm{id_a} \qquad \frac{\mathcal{G};\Gamma\vdash h:A;\Delta \qquad \mathcal{G};\Gamma;h:A\vdash\Delta}{\mathcal{G};\Gamma\vdash\Delta}\ \mathrm{cut_a}$$

$$\frac{\mathcal{G};\Gamma\vdash h:\bot;\Delta}{\mathcal{G};\Gamma\vdash\Delta}\ \bot_{\mathrm{R}} \qquad \frac{\mathcal{G};\Gamma;h:\top\vdash\Delta}{\mathcal{G};\Gamma\vdash\Delta}\ \top_{\mathrm{L}}$$

$$\frac{\mathcal{G};\Gamma\vdash h:A;\Delta}{\mathcal{G};\Gamma;h:\neg A\vdash\Delta}\ \neg_{\mathrm{L}} \qquad \frac{\mathcal{G};\Gamma;h:A\vdash\Delta}{\mathcal{G};\Gamma\vdash h:\neg A;\Delta}\ \neg_{\mathrm{R}}$$

$$\frac{}{\mathcal{G};\Gamma;h:\bot\vdash\Delta}\ \bot_{\mathrm{L}} \qquad \frac{\mathcal{G}[\epsilon/h];\Gamma[\epsilon/h]\vdash\Delta[\epsilon/h]}{\mathcal{G};\Gamma;h:\mathrm{I}\vdash\Delta}\ \mathrm{I_L} \qquad \frac{}{\mathcal{G};\Gamma\vdash\epsilon:\mathrm{I};\Delta}\ \mathrm{I_R}$$

$$\frac{\mathcal{G};\Gamma\vdash h:A;\Delta \qquad \mathcal{G};\Gamma;h:B\vdash\Delta}{\mathcal{G};\Gamma;h:A\to B\vdash\Delta}\ \to_{\mathrm{L}} \qquad \frac{\mathcal{G};\Gamma;h:A\vdash h:B;\Delta}{\mathcal{G};\Gamma\vdash h:A\to B;\Delta}\ \to_{\mathrm{R}}$$

$$\frac{\mathcal{G};\Gamma;h:A,h:B\vdash\Delta}{\mathcal{G};\Gamma;h:A\wedge B\vdash\Delta}\ \wedge_{\mathrm{L}} \qquad \frac{\mathcal{G};\Gamma\vdash h:A;\Delta \qquad \mathcal{G};\Gamma\vdash h:B;\Delta}{\mathcal{G};\Gamma\vdash h:A\wedge B;\Delta}\ \wedge_{\mathrm{R}}$$

$$\frac{\mathcal{G};\Gamma\vdash h:A,h:B;\Delta}{\mathcal{G};\Gamma;h:A\vee B\vdash\Delta}\ \vee_{\mathrm{R}} \qquad \frac{\mathcal{G};\Gamma;h:A\vdash\Delta \qquad \mathcal{G};\Gamma;h:B\vdash\Delta}{\mathcal{G};\Gamma;h:A\vee B\vdash\Delta}\ \vee_{\mathrm{L}}$$

$$\frac{h_1h_2\triangleright h_0;\mathcal{G};\Gamma;h_1:A;h_2:B\vdash\Delta}{\mathcal{G};\Gamma;h_0:A*B\vdash\Delta}\ *_{\mathrm{L}} \qquad \frac{h_1h_0\triangleright h_2;\mathcal{G};\Gamma;h_1:A\vdash h_2:B;\Delta}{\mathcal{G};\Gamma\vdash h_0:A\mathrel{-\!\!*}B;\Delta}\ \mathrel{-\!\!*}_{\mathrm{R}}$$

$$\frac{h_1h_2\triangleright h_0;\mathcal{G};\Gamma\vdash h_1:A;h_0:A*B;\Delta \qquad h_1h_2\triangleright h_0;\mathcal{G};\Gamma\vdash h_2:B;h_0:A*B;\Delta}{h_1h_2\triangleright h_0;\mathcal{G};\Gamma\vdash h_0:A*B;\Delta}\ *_{\mathrm{R}}$$

$$\frac{h_1h_0\triangleright h_2;\mathcal{G};\Gamma;h_0:A\mathrel{-\!\!*}B\vdash h_1:A;\Delta \qquad h_1h_0\triangleright h_2;\mathcal{G};\Gamma;h_0:A\mathrel{-\!\!*}B;h_2:B\vdash\Delta}{h_1h_0\triangleright h_2;\mathcal{G};\Gamma;h_0:A\mathrel{-\!\!*}B\vdash\Delta}\ \mathrel{-\!\!*}_{\mathrm{L}}$$

$$\frac{\mathcal{G};\Gamma;h:A[y/x]\vdash\Delta}{\mathcal{G};\Gamma;h:\exists x.A\vdash\Delta}\ \exists_{\mathrm{L}} \qquad \frac{\mathcal{G};\Gamma\vdash h:A[e/x];h:\exists x.A;\Delta}{\mathcal{G};\Gamma\vdash h:\exists x.A;\Delta}\ \exists_{\mathrm{R}}$$

$$\frac{\mathcal{G};\Gamma\theta\vdash\Delta\theta}{\mathcal{G};\Gamma;h:e_1=e_2\vdash\Delta}\ =_{\mathrm{L}} \qquad \frac{}{\mathcal{G};\Gamma\vdash h:e=e;\Delta}\ =_{\mathrm{R}}$$

$$\frac{\mathcal{G};\Gamma;h':e_1=e_2\vdash\Delta}{\mathcal{G};\Gamma;h:e_1=e_2\vdash\Delta}\ {}^{2}{=}_{\mathrm{L}} \qquad \frac{\mathcal{G};\Gamma\vdash h':e_1=e_2;\Delta}{\mathcal{G};\Gamma\vdash h:e_1=e_2;\Delta}\ {}^{2}{=}_{\mathrm{R}}$$

Side conditions:
Each label being substituted cannot be ϵ, each expression being substituted cannot be a constant.
In $=_{\mathrm{L}}$, $\theta = mgu(\{e_1, e_2\})$.
in $*_{\mathrm{L}}$, $\mathrel{-\!\!*}_{\mathrm{R}}$, the labels h_1 and h_2 do not occur in the conclusion.
In $\exists_{\mathrm{L}}$, y is not free in the conclusion.

Figure 2 Logical rules in $\mathrm{GM_{SL}}$.

$$\frac{}{\mathcal{G};\Gamma;\epsilon:e_1\mapsto e_2\vdash\Delta}\mapsto_{\mathrm{L}_1} \qquad \frac{h_1h_0\rhd h_2;\mathcal{G};\Gamma;h_1:e_1\mapsto e_2\vdash\Delta}{\mathcal{G};\Gamma\vdash\Delta}\,\mathrm{HE}$$

$$\frac{\begin{array}{l}\epsilon h_0\rhd h_0;\mathcal{G}[\epsilon/h_1,h_0/h_2];\Gamma[\epsilon/h_1,h_0/h_2];h_0:e_1\mapsto e_2\vdash\Delta[\epsilon/h_1,h_0/h_2]\\ h_0\epsilon\rhd h_0;\mathcal{G}[\epsilon/h_2,h_0/h_1];\Gamma[\epsilon/h_2,h_0/h_1];h_0:e_1\mapsto e_2\vdash\Delta[\epsilon/h_2,h_0/h_1]\end{array}}{h_1h_2\rhd h_0;\mathcal{G};\Gamma;h_0:e_1\mapsto e_2\vdash\Delta}\mapsto_{\mathrm{L}_2}$$

$$\frac{}{h_1h_2\rhd h_0;\mathcal{G};\Gamma;h_1:e\mapsto e_1;h_2:e\mapsto e_2\vdash\Delta}\mapsto_{\mathrm{L}_3} \qquad \frac{\mathcal{G};\Gamma\theta;h:e_1\theta\mapsto e_2\theta\vdash\Delta\theta}{\mathcal{G};\Gamma;h:e_1\mapsto e_2;h:e_3\mapsto e_4\vdash\Delta}\mapsto_{\mathrm{L}_4}$$

$$\frac{\mathcal{G}[h_1/h_2];\Gamma[h_1/h_2];h_1:e_1\mapsto e_2\vdash\Delta[h_1/h_2]}{\mathcal{G};\Gamma;h_1:e_1\mapsto e_2;h_2:e_1\mapsto e_2\vdash\Delta}\mapsto_{\mathrm{L}_5} \qquad \frac{}{\mathcal{G};\Gamma;h:nil\mapsto e\vdash\Delta}\,\mathrm{NIL}$$

$$\frac{h_3h_4\rhd h_1;h_5h_6\rhd h_2;\mathcal{G};\Gamma;h_3:e_1\mapsto e_2;h_5:e_1\mapsto e_3\vdash\Delta \qquad h_1h_2\rhd h_0;\mathcal{G};\Gamma\vdash\Delta}{\mathcal{G};\Gamma\vdash\Delta}\,\mathrm{HC}$$

$$\frac{h_1h_2\rhd h_0;\mathcal{G};\Gamma\theta[h_1/h_3,h_2/h_4];h_1:e_1\theta\mapsto e_2\theta\vdash\Delta\theta[h_1/h_3,h_2/h_4]}{h_1h_2\rhd h_0;h_3h_4\rhd h_0;\mathcal{G};\Gamma;h_1:e_1\mapsto e_2;h_3:e_1\mapsto e_3\vdash\Delta}\mapsto_{\mathrm{L}_6}$$

$$\frac{\begin{array}{ll}h_1h_5\rhd h_4;h_3h_5\rhd h_2;&\\ h_1h_2\rhd h_0;h_3h_4\rhd h_0;\mathcal{G};\Gamma;h_1:e_1\mapsto e_2;h_3:e_3\mapsto e_4 & \vdash h:e_1=e_3;\Delta\end{array}}{h_1h_2\rhd h_0;h_3h_4\rhd h_0;\mathcal{G};\Gamma;h_1:e_1\mapsto e_2;h_3:e_3\mapsto e_4 \qquad \vdash h:e_1=e_3;\Delta}\mapsto_{\mathrm{L}_7}$$

Side conditions:
Each label being substituted cannot be ϵ, each expression substituted cannot be a constant.
In $\mapsto_{\mathrm{L}_4}$, $\theta = mgu(\{(e_1,e_3),(e_2,e_4)\})$.
In $\mapsto_{\mathrm{L}_6}$, $\theta = mgu(\{e_2,e_3\})$.
In HE, h_0 occurs in conclusion, h_1,h_2,e_1 are fresh.
In HC, h_1,h_2 occur in the conclusion, $h_0,h_3,h_4,h_5,h_6,e_1,e_2,e_3$ are fresh in the premise.

Figure 3 Pointer rules in $\mathrm{GM}_{\mathrm{SL}}$.

to be in *normal form* (or *normal*) whenever all its production rules share the same head and the same annotation. It is straightforward to put any inductive definition into a normal form by adding equalities and existential quantifications over the free variables of an annotated production rule that occcur in its body but not in its head.

▶ **Definition 3** (Unfolding rules). Let $C_1 \overset{z}{\Rightarrow} P(\mathrm{x}),\ldots,C_k \overset{z}{\Rightarrow} P(\mathrm{x})$ be a normal inductive definition of the inductive predicate symbol P. Then each production rule gives rise to a right-unfolding rule and to one premiss of the single (multi-premiss) left-unfolding rule for P (also called case-split rule):

$$\frac{\mathcal{G};\Gamma\vdash h:C_i;\Delta}{\mathcal{G};\Gamma\vdash h:P(\mathrm{x});\Delta}\,\mathrm{P}_{\mathrm{R}_i} \qquad \frac{\mathcal{G};\Gamma;h:C_1\vdash\Delta \quad \ldots \quad \mathcal{G};\Gamma;h:C_n\vdash\Delta}{\mathcal{G};\Gamma;h:P(\mathrm{x})\vdash\Delta}\,\mathrm{P}_{\mathrm{L}}$$

Let us illustrate Definition 3 with list segments. From now on, we shall write $\ell\!s$ for arbitrary list segments and let ls denote only acyclic list segments. The inductive definition for $\ell\!s$ goes as follows:

$$\mathrm{I} \overset{x}{\Rightarrow} \ell\!s(x,x) \qquad\qquad x\mapsto z * \ell\!s(z,y) \overset{x,y,z}{\Rightarrow} \ell\!s(x,y)$$

which first gets normalized to obtain:

$$x=y\wedge \mathrm{I} \overset{x,y}{\Rightarrow} \ell\!s(x,y) \qquad\qquad \exists u.\, x\mapsto u * \ell\!s(u,y) \overset{x,y}{\Rightarrow} \ell\!s(x,y)$$

$$\frac{\mathcal{G};\Gamma \vdash h : e_1 = e_2 \wedge \mathrm{I};\Delta}{\mathcal{G};\Gamma \vdash h : \ell s(e_1,e_2);\Delta}\ \ell s_{\mathrm{R}_1} \qquad \frac{\mathcal{G};\Gamma \vdash h : \exists u.e_1 \mapsto u * \ell s(u,e_2);\Delta}{\mathcal{G};\Gamma \vdash h : \ell s(e_1,e_2);\Delta}\ \ell s_{\mathrm{R}_2}$$

$$\frac{\mathcal{G};\Gamma;h : e_1 = e_2;h : \mathrm{I} \vdash \Delta \qquad \mathcal{G};\Gamma;h : \exists u.e_1 \mapsto u * \ell s(u,e_2) \vdash \Delta}{\mathcal{G};\Gamma;h : \ell s(e_1,e_2) \vdash \Delta}\ \ell s_{\mathrm{L}}$$

$$\frac{\mathcal{G};\Gamma \vdash h : e_1 = e_2 \wedge \mathrm{I};\Delta}{\mathcal{G};\Gamma \vdash h : ls(e_1,e_2);\Delta}\ \mathrm{ls}_{\mathrm{R}_1} \qquad \frac{\mathcal{G};\Gamma \vdash h : e_1 \neq e_2 \wedge (\exists u.e_1 \mapsto u * ls(u,e_2));\Delta}{\mathcal{G};\Gamma \vdash h : ls(e_1,e_2);\Delta}\ \mathrm{ls}_{\mathrm{R}_2}$$

$$\frac{\mathcal{G};\Gamma;h : e_1 = e_2;h : \mathrm{I} \vdash \Delta \qquad \mathcal{G};\Gamma;h : e_1 \neq e_2;h : \exists u.e_1 \mapsto u * ls(u,e_2) \vdash \Delta}{\mathcal{G};\Gamma;h : ls(e_1,e_2) \vdash \Delta}\ \mathrm{ls}_{\mathrm{L}}$$

Figure 4 $\mathrm{GM}_{\mathrm{SL}}$ rules for list segments.

For ls, the inductive definition (where $x \neq y$ is syntactic sugar for $\neg(x = y)$):

$$\mathrm{I} \overset{x}{\Rightarrow} ls(x,x) \qquad\qquad x \neq y \wedge (x \mapsto z * ls(z,y)) \overset{x,y,z}{\Rightarrow} ls(x,y)$$

gets normalized to

$$x = y \wedge \mathrm{I} \overset{x,y}{\Rightarrow} ls(x,y) \qquad\qquad x \neq y \wedge \exists u.\, x \mapsto u * ls(u,y) \overset{x,y}{\Rightarrow} \ell s(x,y)$$

Generalizing from variables to expressions and expanding additive conjunctions on the left-hand side of sequents, we obtain the unfolding rules depicted in Figure 4 for list segments.

▶ **Definition 4.** For any inductive predicate symbol P_i with arity a_i defined by the production rules $C_1(\mathrm{x}_1) \Rightarrow P(\mathrm{x}_1), \ldots, C_k(\mathrm{x}_k) \Rightarrow P(\mathrm{x}_k)$ we obtain a corresponding n-ary function $\varphi_i : \wp(Heaps \times Val^{a_i}) \times \ldots \times \wp(Heaps \times Val^{a_n}) \to \wp(Heaps \times Val^{a_i})$ as follows:

$$\varphi_i(\mathrm{X}) = \bigcup_{1 \leq j \leq k} \{(h,\mathrm{v}) \mid (s[\mathrm{x}_j \mapsto \mathrm{v}],h) \models_{[\![\mathrm{P}]\!] \mapsto \mathrm{X}} C_j(\mathrm{x}_j)\}$$

where s is an arbitrary stack and $\models_{[\![\mathrm{P}]\!] \mapsto \mathrm{X}}$ is the satisfaction relation defined exactly as in Definition 1 except that $[\![P_i]\!] = \pi_i^n(\mathrm{X})$ for each $i \in \{1,\ldots,n\}$.

Any variables occurring in the right hand side but not the left hand side of the set comprehension in the definition of φ_i above are, implicitly, existentially quantified over the entire right hand side of the comprehension.

▶ **Definition 5.** The *definition set operator* for $P_1,\ldots,P_n$ is defined as the operator Φ_{P}, with domain and codomain $\wp(Heaps \times Val^{a_1}) \times \ldots \times \wp(Heaps \times Val^{a_n})$ such that $\Phi_{\mathrm{P}}(\mathrm{X}) = (\varphi_1(\mathrm{X}),\ldots,\varphi_n(\mathrm{X}))$.

It is proved in [3] that the operator generated from a set of inductive definitions by Definition 5 is monotone and therefore has a least fixed-point that can be iteratively approached by *approximants*. First define a chain of ordinal-indexed sets $(\Phi_{\mathrm{P}}^{\alpha})_{\alpha \geq 0}$ by transfinite induction: $\Phi_{\mathrm{P}}^{\alpha} = \bigcup_{\beta<\alpha} \Phi_{\mathrm{P}}(\Phi_{\mathrm{P}}^{\beta})$ (note that this implies $\Phi_{\mathrm{P}}^{\alpha} = (\emptyset,\ldots,\emptyset)$). Then for each $i \in \{1,\ldots,n\}$, the set $P_i^{\alpha} = \pi_i^n(\Phi_{\mathrm{P}}^{\alpha})$ is called the α-approximant of P_i. Finally, for each $i \in \{1,\ldots,n\}$, the standard interpretation of the inductive predicate P_i is given by $[\![P_i]\!] = \bigcup_{\alpha} P_i^{\alpha}$ and the forcing relation in Definition 1 is extended with the clause

$$(s,h) \models P_i(x_1,\ldots,x_n) \text{ iff } (h,[\![x_1]\!]_s,\ldots,[\![x_n]\!]_s) \in [\![P_i]\!].$$

4.2 Labelled Cyclic Proofs

$\mathrm{GM_{SL}}$ handles induction with the notion of *labelled cyclic proofs*. Although we reuse the terminology of buds, companions and pre-proofs used in [3, 4], we adapt it in the context of a labelled proof-system. Moreover, one key difference in that our soundness criterion to turn a pre-proof into a cyclic proof is based on an argument relying on a measure of the size of the heaps rather than on a global trace condition on traces following a path in a pre-proof.

▶ **Definition 6** (Pre-proof)**.** Let $\mathcal{D}$ be a derivation in $\mathrm{GM_{SL}}$ for a root sequent S. Each leaf sequent B in $\mathcal{D}$ which is not the conclusion of an inference rule is called a *bud*. A *pre-proof* of a sequent S is a pair $(\mathcal{D}, \mathcal{R})$ where $\mathcal{D}$ is a derivation the root of which is S and $\mathcal{R}$ is a function which assigns to every bud B in $\mathcal{D}$ a triple (C, θ, σ) such that $C\theta\sigma \subseteq B$[6], where C, called a *companion* for B, is a sequent occurring before B in the branch of $\mathcal{D}$ containing B, θ is an expression renaming substitution and σ is a label renaming substitution.

Let us consider a denumerable set $SVar = \{m_0, m_2, \ldots\}$ of *size variables* and a (fixed) injective function $|\cdot| : Heaps \to SVar$.

▶ **Definition 7** (Size constraints)**.** A *size constraint* is an expression of the form $s\ op\ s$, where $op \in \{=, \neq, \leq, <, \geq, >\}$ and s is a (non-empty) sum over $\mathbb{N} \cup SVar$. A set M of size constraints is *consistent* if it has a solution, *i.e.*, there exists a *measure* $\mu : SVar \to \mathbb{N}$ satisfying all the size constraints in M. Given two sets of size constraints M_1 and M_2, M_1 entails M_2, written $M_1 \models M_2$, if any solution of M_1 is also a solution of M_2.

A $\mathrm{GM_{SL}}$ sequent $S = \mathcal{G}; \Gamma \vdash \Delta$ induces a set $Size(S)$ defined as the smallest set of size constraints such that if $h_1 h_2 \triangleright h \in \mathcal{G}$ then $|h| = |h_2| + |h_1| \in Size(S)$, if $h : \mathrm{I} \in \Gamma$ then $|h| = 0 \in Size(S)$, if $h : \mathrm{I} \in \Delta$ then $|h| > 0 \in Size(S)$, and if $h : x \overset{1}{\mapsto} y \in \Gamma$ or $h : x \overset{2}{\mapsto} y, z \in \Gamma$ then $|h| = 1 \in Size(S)$.

▶ **Definition 8** (Cyclic proof)**.** A pre-proof $(\mathcal{D}, \mathcal{R})$ of a sequent S is a *(labelled) cyclic proof* if it satisfies the following condition: for each bud B in $\mathcal{D}$, the assigned companion $C = \mathcal{R}(B)$ contains at a least one inductive predicate symbol P such that $Size(B) \models \{|h_B| < |h_C|\}$, where h_B and h_C are the heaps labelling the same occurrence of P[7] in B and C respectively.

Figure 5 gives an example of a pre-proof for the (LC) entailment in $\mathrm{GM_{SL}}$. The bud B and companion C of this pre-proof are indicated by the (†) marks and respectively take the following forms:

$$B \overset{\text{def}}{=} \quad h_4 h_2 \triangleright h_5; \mathcal{G}_B; h_4 : \ell\!\!\ell(u, x'); h_2 : \ell\!\!\ell(x', y); \Gamma_B \vdash h_5 : \ell\!\!\ell(u, y)$$
$$C \overset{\text{def}}{=} \quad h_1 h_2 \triangleright h_0; h_1 : \ell\!\!\ell(x, x'); h_2 : \ell\!\!\ell(x', y) \vdash h_0 : \ell\!\!\ell(x, y)$$

Moreover, we have $C\theta\sigma \subseteq B$ with $\sigma = [\, h_4/h_1, h_5/h_0 \,]$ and $\theta = [\, x \mapsto u \,]$. This pre-proof is a cyclic proof because, in the bud B, $h_3 h_4 \triangleright h_1$ and $h_3 : x \mapsto u$ imply that $|h_1| = |h_4| + 1$ and thus $Size(B) \models \{|h_4| < |h_1|\}$ for the first occurrence of the $\ell\!\!\ell$ predicate in B and C.

Another example is the following entailment which states that if a heap represents a list segment ending with y, then y is not an address occurring in the heap and cannot point anywhere (*i.e.*, y is dangling):

$$(ALE) \overset{\text{def}}{=} \quad ls(x, y) \models \neg(y \mapsto z * \top)$$

[6] Using inclusion allows us to do without weakening rules in $\mathrm{GM_{SL}}$.
[7] Keeping track of the various occurrences of a predicate symbol can easily be done using indexes.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon h_2 \rhd h_0; h_2 : \ell\mathfrak{s}(x,y) \vdash h_2 : \ell\mathfrak{s}(x,y)}\,\mathrm{id_a}}{\epsilon h_2 \rhd h_0; h_2 : \ell\mathfrak{s}(x,y) \vdash h_0 : \ell\mathfrak{s}(x,y)}\,\mathrm{Eq_2}}{h_1 h_2 \rhd h_0; h_1 : \mathrm{I}; h_2 : \ell\mathfrak{s}(x,y) \vdash h_0 : \ell\mathfrak{s}(x,y)}\,\mathrm{I_L}}{h_1 h_2 \rhd h_0; h_1 : x = x'; h_1 : \mathrm{I}; h_2 : \ell\mathfrak{s}(x',y) \vdash h_0 : \ell\mathfrak{s}(x,y)}\,{=_\mathrm{L}}$$

Π_1

$$\dfrac{}{\begin{array}{l} h_3 h_5 \rhd h_0; h_2 h_4 \rhd h_5; \\ h_3 h_4 \rhd h_1; h_1 h_2 \rhd h_0; \\ h_3 : x \mapsto u; h_4 : \ell\mathfrak{s}(u,x'); h_2 : \ell\mathfrak{s}(x',y) \quad \vdash h_3 : x \mapsto u \end{array}}\,\mathrm{id_a}$$

Π_2

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_2}{\begin{array}{l} h_3 h_5 \rhd h_0; h_4 h_2 \rhd h_5; \\ h_3 h_4 \rhd h_1; h_1 h_2 \rhd h_0; \\ (\dagger)\ h_3 : x \mapsto u; h_4 : \underline{\ell\mathfrak{s}(u,x')}; h_2 : \ell\mathfrak{s}(x',y) \ \vdash h_5 : \ell\mathfrak{s}(u,y) \end{array}}\,\mathrm{E}}{\begin{array}{l} h_3 h_5 \rhd h_0; h_2 h_4 \rhd h_5; \\ h_3 h_4 \rhd h_1; h_1 h_2 \rhd h_0; \\ h_3 : x \mapsto u; h_4 : \underline{\ell\mathfrak{s}(u,x')}; h_2 : \ell\mathfrak{s}(x',y) \ \vdash h_5 : \ell\mathfrak{s}(u,y) \end{array}}\,{*_\mathrm{R}}}{\begin{array}{l} h_3 h_5 \rhd h_0; h_2 h_4 \rhd h_5; \\ h_3 h_4 \rhd h_1; h_1 h_2 \rhd h_0; \\ h_3 : x \mapsto u; h_4 : \underline{\ell\mathfrak{s}(u,x')}; h_2 : \ell\mathfrak{s}(x',y) \ \vdash h_0 : x \mapsto u * \ell\mathfrak{s}(u,y) \end{array}}\,\mathrm{A}}{\begin{array}{l} h_3 h_4 \rhd h_1; h_1 h_2 \rhd h_0; \\ h_3 : x \mapsto u; h_4 : \underline{\ell\mathfrak{s}(u,x')}; h_2 : \ell\mathfrak{s}(x',y) \ \vdash h_0 : x \mapsto u * \ell\mathfrak{s}(u,y) \end{array}}\,{\exists_\mathrm{R}}}{\begin{array}{l} h_3 h_4 \rhd h_1; h_1 h_2 \rhd h_0; \\ h_3 : x \mapsto u; h_4 : \underline{\ell\mathfrak{s}(u,x')}; h_2 : \ell\mathfrak{s}(x',y) \ \vdash h_0 : \exists u.x \mapsto u * \ell\mathfrak{s}(u,y) \end{array}}\,{\ell\mathfrak{s}_{\mathrm{R}_2}}}{\begin{array}{l} h_3 h_4 \rhd h_1; h_1 h_2 \rhd h_0; \\ h_3 : x \mapsto u; h_4 : \underline{\ell\mathfrak{s}(u,x')}; h_2 : \ell\mathfrak{s}(x',y) \ \vdash h_0 : \ell\mathfrak{s}(x,y) \end{array}}\,{*_\mathrm{L}}}{\begin{array}{l} h_1 h_2 \rhd h_0; h_1 : x \mapsto u * \underline{\ell\mathfrak{s}(u,x')}; h_2 : \ell\mathfrak{s}(x',y) \vdash h_0 : \ell\mathfrak{s}(x,y) \end{array}}\,{\exists_\mathrm{L}}}{\begin{array}{l} h_1 h_2 \rhd h_0; h_1 : \exists u.x \mapsto u * \underline{\ell\mathfrak{s}(u,x')}; h_2 : \ell\mathfrak{s}(x',y) \vdash h_0 : \ell\mathfrak{s}(x,y) \end{array}}\,{\ell\mathfrak{s}_\mathrm{L}}}{\begin{array}{l} (\dagger)\ h_1 h_2 \rhd h_0; h_1 : \underline{\ell\mathfrak{s}(x,x')}; h_2 : \ell\mathfrak{s}(x',y) \vdash h_0 : \ell\mathfrak{s}(x,y) \end{array}}\,{*_\mathrm{L}}}{\begin{array}{l} h_0 : \ell\mathfrak{s}(x,x') * \ell\mathfrak{s}(x',y) \vdash h_0 : \ell\mathfrak{s}(x,y) \end{array}}\,{\to_\mathrm{R}}}{\begin{array}{l} \vdash h_0 : (\ell\mathfrak{s}(x,x') * \ell\mathfrak{s}(x',y)) \to \ell\mathfrak{s}(x,y) \end{array}}$$

Π_1

Figure 5 Cyclic proof of $(\ell\mathfrak{s}(x,x') * \ell\mathfrak{s}(x',y)) \to \ell\mathfrak{s}(x,y)$ in $\mathrm{GM_{SL}}$.

$$\dfrac{\dfrac{\dfrac{}{\epsilon : x = y; \epsilon : y \mapsto z \vdash}\mapsto_{L_1}}{h_1h_2 \rhd \epsilon;\ \epsilon : x = y; h_1 : y \mapsto z \vdash}\mathrm{IU}}{h_1h_2 \rhd h_0; h_0 : \mathrm{I};\ h_0 : x = y; h_1 : y \mapsto z \vdash}\mathrm{I_L} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{l}h_1h_5 \rhd h_4; h_3h_5 \rhd h_2;\\ h_3h_4 \rhd h_0; h_1h_2 \rhd h_0;\\ (\dagger)\ h_0 : x \neq y; h_3 : x \mapsto u; h_4 : \underline{ls(u,y)}; h_1 : y \mapsto z \vdash\end{array}}\mapsto_{L_7}}{\begin{array}{l}h_3h_4 \rhd h_0; h_1h_2 \rhd h_0;\\ h_0 : x \neq y; h_3 : x \mapsto u; h_4 : \underline{ls(u,y)}; h_1 : y \mapsto z \vdash\end{array}}*_{\mathrm{L}}}{h_1h_2 \rhd h_0; h_0 : x \neq y; h_0 : x \mapsto u * \underline{ls(u,y)}; h_1 : y \mapsto z \vdash}\exists_{\mathrm{L}}}{h_1h_2 \rhd h_0; h_0 : x \neq y; h_0 : \exists u.x \mapsto u * \underline{ls(u,y)}; h_1 : y \mapsto z \vdash}\mathrm{ls_L}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(\dagger)\ h_1h_2 \rhd h_0; h_0 : \underline{ls(x,y)}; h_1 : y \mapsto z \vdash}\top_{\mathrm{L}}}{h_1h_2 \rhd h_0; h_0 : ls(x,y); h_1 : y \mapsto z; h_2 : \top \vdash}*_{\mathrm{L}}}{h_0 : ls(x,y); h_0 : (y \mapsto z * \top) \vdash}\neg_{\mathrm{R}}}{h_0 : ls(x,y) \vdash h_0 : \neg(y \mapsto z * \top)}\to_{\mathrm{R}}}{\vdash h_0 : ls(x,y) \to \neg(y \mapsto z * \top)}$$

Figure 6 Cyclic proof of $(ls(x,y) \to \neg(y \mapsto z * \top))$ in $\mathrm{GM_{SL}}$.

(ALE) is valid in Reynold's semantics if and only if for all states (s,h):

$$(s,h) \models ls(x,y) \text{ implies } (s,h) \not\models y \mapsto z * \top$$

(ALE) is obviously not valid for arbitrary list segments since a panhandle list needs to have y pointing back somewhere in the list. However, (ALE) is valid for acyclic list segments (the proof is in Appendix B).

A pre-proof of (ALE) in $\mathrm{GM_{SL}}$ is given in Figure 6. The bud B and companion C of this pre-proof are indicated by the $(\dagger)$ marks and respectively take the forms:

$$\begin{array}{lll} B & \stackrel{\text{def}}{=} & h_1h_5 \rhd h_4; \mathcal{G}_B; h_4 : ls(u,y); h_1 : y \mapsto z; \Gamma_B \vdash \Delta_B \\ C & \stackrel{\text{def}}{=} & h_1h_2 \rhd h_0; h_0 : ls(x,y); h_1 : y \mapsto z \vdash \Delta_B \end{array}$$

Moreover, we have $C\theta\sigma \subseteq B$ with $\sigma = [\,h_5/h_2, h_4/h_0\,]$ and $\theta = [\,x \mapsto u\,]$. This pre-proof is also a cyclic proof because, in the bud B, $h_3h_5 \rhd h_2$ and $h_3 : x \mapsto u$ imply that $|h_5| < |h_2|$ and it then follows from $h_1h_2 \rhd h_0$ and $h_1h_5 \rhd h_4$ that $Size(B) \models \{|h_4| < |h_0|\}$.

▶ **Theorem 9.** *If there is a cyclic proof of* $\vdash h_0 : F$ *in* $\mathrm{GM_{SL}}$*, then* F *is valid in* SL.

Proof. (Sketch) Proving the soundness of $\mathrm{GM_{SL}}$ requires two things: first proving the local soundness of the proof-rules and then proving the soundness of the cyclic mechanism.

A *label mapping* for a labelled sequent $S = \mathcal{G}; \Gamma \vdash \Delta$ is a function ρ mapping each heap label in the sequent to an actual heap of the heap model of SL and such that $\rho(\epsilon) = \epsilon$ and for all $h_ih_j \rhd h_k \in \mathcal{G}$, $\rho(h_i)\rho(h_j) = \rho(h_k)$. A *realization* for S is a pair (s,ρ) where s is a stack and ρ a label mapping for S such that for all $h_i : A \in \Gamma$, $(s,\rho(h_i)) \models A$ and for all $h_i : A \in \Delta$, $(s,\rho(h_i)) \not\models A$. S is *realizable* if there is a realization for S. Local soundness follows the standard pattern of proving that every proof-rule preserves realizability (*i.e.*, that the realizability of the conclusion of a proof-rule entails the realizability of at least one of its premisses) and has already been proven for the most part of the proof-rules in $\mathrm{T_{SL}}$ [8] and $\mathrm{LS_{SL}}$ [9]. The new proof-rules of $\mathrm{GM_{SL}}$ are easily proven along the lines of their intuitive justifications at the beginning of Section 4. The local soundness of the unfolding rules obtained by Definition 3 is an easy consequence of the production rules being read as a disjunction $\bigvee_i C_i$ of inductive clauses.

$$\frac{\mathcal{G}; \Gamma[e_2 \mapsto e_1] \vdash \Delta[e_2 \mapsto e_1]}{\mathcal{G}; \Gamma; \epsilon : ls(e_1, e_2) \vdash \Delta}\ \mathrm{LS}_1 \qquad \frac{}{\mathcal{G}; \Gamma \vdash \epsilon : ls(e, e); \Delta}\ \mathrm{LS}_2$$

$$\frac{\mathcal{G}; \Gamma; h : \mathrm{I} \vdash \Delta}{\mathcal{G}; \Gamma; h : ls(e, e) \vdash \Delta}\ \mathrm{LS}_3 \qquad \frac{\mathcal{G}; \Gamma[e \mapsto nil]; h : \mathrm{I} \vdash \Delta[e \mapsto nil]}{\mathcal{G}; \Gamma; h : ls(nil, e) \vdash \Delta}\ \mathrm{LS}_4$$

$$\frac{\mathcal{G}; \Gamma\theta_1; h : \mathrm{I} \vdash \Delta\theta_1 \qquad \mathcal{G}; \Gamma\theta_2; h : ls(e_1\theta_2, e_2\theta_2) \vdash \Delta\theta_2}{\mathcal{G}; \Gamma; h : ls(e_1, e_2); h : ls(e_3, e_4) \vdash \Delta}\ \mathrm{LS}_5$$

$$\frac{h_1h_2 \triangleright h_0; \mathcal{G}; \Gamma; h_1 : ds(e_1, e_2); h_0 : ls(e_1, e_3); h_2 : ls(e_2, e_3) \vdash \Delta}{h_1h_2 \triangleright h_0; \mathcal{G}; \Gamma; h_1 : ds(e_1, e_2); h_0 : ls(e_1, e_3) \vdash \Delta}\ \mathrm{LS}_6$$

$$\frac{h_1h_2 \triangleright h_0; \mathcal{G}; \Gamma; h_1 : ds(e_2, e_3); h_0 : ls(e_1, e_3); h_2 : ls(e_1, e_2) \vdash \Delta}{h_1h_2 \triangleright h_0; \mathcal{G}; \Gamma; h_1 : ds(e_2, e_3); h_0 : ls(e_1, e_3) \vdash \Delta}\ \mathrm{LS}_7$$

$$\frac{\begin{array}{l} h_1h_2 \triangleright h_0; h_1h_3 \triangleright h_4; \mathcal{G}; \\ \Gamma; h_1 : ds(e_1, e_2); h_3 : ad(e_3) \ \vdash h_2 : ls(e_2, e_3); h_0 : ls(e_1, e_3); h : G(ad(e_3)); \Delta \end{array}}{\begin{array}{l} h_1h_2 \triangleright h_0; h_1h_3 \triangleright h_4; \mathcal{G}; \\ \Gamma; h_1 : ds(e_1, e_2); h_3 : ad(e_3) \ \vdash h_0 : ls(e_1, e_3); h : G(ad(e_3)); \Delta \end{array}}\ \mathrm{LS}_8$$

$$\frac{}{h_1h_2 \triangleright h_0; \mathcal{G}; \Gamma; h_1 : ad(e_1); h_2 : ad(e_1)' \vdash h_3 : G(ad(e_1)); h_3 : G(ad(e_1)'); \Delta}\ \mathrm{IC}$$

Abbreviations and side conditions:
$ds(e, e')$ is either $(e \mapsto e')$ or $ls(e, e')$.
$ad(e)$ stands for one of $(e \mapsto e')$, $(e \mapsto e', e'')$, $ls(e, e')$, for some e', e''. Similarly for $ad(e)'$.
$G(ad(e))$ is defined as $G(e \mapsto e') \stackrel{\text{def}}{=} G(e \mapsto e', e'') \stackrel{\text{def}}{=} \bot$, $G(ls(e, e')) \stackrel{\text{def}}{=} (e = e')$.
In LS_5, $\theta_1 = mgu(\{(e_1, e_2), (e_3, e_4)\})$ and $\theta_2 = mgu(\{(e_1, e_3), (e_2, e_4)\})$.
In LS_8, if e_3 is nil, then $h_1h_3 \triangleright h_4$, $h_3 : ad(e_3)$ and $h : G(ad(e_3))$ in the conclusion are optional.
In LS_8, if $ds(e_1, e_2)$ is $(e_1 \mapsto e_2)$, then $h_1h_3 \triangleright h_4$, $h_3 : ad(e_3)$ and $h : G(ad(e_3))$ in the conclusion are optional, on the condition that $h' : (e_1 = e_3)$ occurs in the RHS of the conclusion, for some h'.

Figure 7 Rules for acyclic list segments in $\mathrm{LS}_{\mathrm{SL}}$.

Let us prove that the cyclic mechanism is globally sound. Suppose otherwise, then we have a cyclic proof $(\mathcal{D}, \mathcal{R})$ for a sequent $\vdash h_0 : F$ but F is not valid in SL. Then the root sequent S is not realizable and we have a branch in $\mathcal{D}$ with a bud B and a companion C satisfying the conditions of Definition 8. Travelling in $\mathcal{D}$ from S to B and then jumping back to C to cycle all over again between C and B, we can construct an infinite sequence of sequents $\mathcal{S} = (S_i)_{(i \in \mathbb{N})}$ with $S_0 = S$. It then follows from local soundness and the fact that S_0 is not realizable that no sequent in $(S_i)_{(i \in \mathbb{N})}$ can be realizable. Moreover, there exists a predicate symbol P such that each time $\mathcal{S}$ reaches an occurrence of the bud B, the heap associated with this occurrence of P in B has a size strictly lower than the size of the heap associated with the same occurrence of P in the previous occurrence of C and B in $\mathcal{S}$. Thus, we can construct an infinite sequence of heaps the size of which is strictly decreasing, which is impossible because, by definition, the size of a heap, which is the size of its (finite) domain, cannot be lower than 0. ◀

▶ **Theorem 10.** *The rules for lists in* $\mathrm{LS}_{\mathrm{SL}}$ *(given in Figure 7) are derivable in* $\mathrm{GM}_{\mathrm{SL}}$.

Proof. The detailed proof is in Appendix A. ◀

References

1. J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *4th Int. Symposium on Formal Methods for Components and Objects, FMCO'2005*, LNCS 4111, pages 115–137, Amsterdam, Netherlands, 2005.
2. J. Berdine, C. Calcagno, and P.W. O'Hearn. Symbolic execution with separation logic. In *3rd Asian Symposium on Programming Languages and Systems, APLAS'2005*, LNCS 3780, pages 52–68, Tsukuba, Japan, 2005.
3. J. Brotherston. Formalised inductive reasoning in the logic of bunched implications. In *14th Symposium on Static Analysis, SAS'2007*, LNCS 4634, pages 87–103, Kongens Lyngby, Denmark, 2007.
4. J. Brotherston, D. Distefano, and R.L. Petersen. Automatic cyclic entailment proofs in separation logic. In *23rd Int. Conference on Automated Deduction, CADE'2011*, LNCS 6803, pages 131–146, Wroclaw, Poland, 2011.
5. J. Brotherston and M. Kanovich. Undecidability of propositional separation logic and its neighbour. In *IEEE Symposium on Logic in Computer Science, LICS'2010*, pages 130–139, Edinburgh, Scotland, 2010.
6. C. Calcagno, H. Yang, and P.W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *20th Int. Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'2001*, LNCS 2245, pages 108–119, Bangalore, India, 2001.
7. S. Demri, D. Galmiche, D. Larchey-Wendling and D. Méry. Separation logic with one quantified variable. In *9th Int. Symposium on Computer Science in Russia, CSR'2014*, pages 125–138, Moscow, Russia, 2014.
8. D. Galmiche and D. Méry. Tableaux and resource graphs for separation logic. *Journal of Logic and Computation*, 20(1):189–231, 2010.
9. Z. Hou, R. Gore, and A. Tiu. Automated theorem proving for assertions in separation logic with all connectives. In *25th Int. Conference on Automated Deduction, CADE'2015*, LNAI 9195, pages 501–516, Berlin, Germany, 2015.
10. S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *28th ACM Symposium on Principles of Programming Languages, POPL'2001*, pages 14–26, London, UK, 2001.
11. D. Larchey-Wendling and D. Galmiche. The undecidability of boolean BI through phase semantics. In *IEEE Symposium on Logic in Computer Science, LICS'2010*, pages 140–149, Edinburgh, Scotland, 2010.
12. P.W. O'Hearn and D. Pym. The Logic of Bunched Implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.
13. P.W. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th Int. Workshop on Computer Science Logic, CSL'2001*, LNCS 2142, pages 1–19, Paris, France, 2001.
14. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science, LICS'2002*, pages 55–74, Copenhagen, Denmark, 2002.

A Proofs of $\mathrm{LS}_{\mathrm{SL}}$ acyclic list segment rules in $\mathrm{GM}_{\mathrm{SL}}$

In this section we prove that all the rules of $\mathrm{LS}_{\mathrm{SL}}$ for acyclic list segments depicted in Figure 7 are derivable in $\mathrm{GM}_{\mathrm{SL}}$.

1. Derivation of LS_1

$$\dfrac{\dfrac{\mathcal{G};\Gamma[\,e_2 \mapsto e_1\,] \vdash \Delta[\,e_2 \mapsto e_1\,]}{\dfrac{\mathcal{G};\Gamma;\epsilon : e_1 = e_2 \vdash \Delta}{\mathcal{G};\Gamma;\epsilon : e_1 = e_2;\epsilon : \mathrm{I} \vdash \Delta}\,\mathrm{I_L}}\,{=_\mathrm{L}} \qquad \dfrac{\dfrac{\dfrac{}{\mathcal{G};\Gamma;\epsilon : e_1 \neq e_2;\epsilon : e_1 \mapsto u;\epsilon : ls(u,e_2) \vdash \Delta}\,{\mapsto_{\mathrm{L}_1}}}{h_1h_2 \triangleright \epsilon;\mathcal{G};\Gamma;\epsilon : e_1 \neq e_2;h_1 : e_1 \mapsto u;h_2 : ls(u,e_2) \vdash \Delta}\,\mathrm{IU}}{\mathcal{G};\Gamma;\epsilon : e_1 \neq e_2;\epsilon : e_1 \mapsto u * ls(u,e_2) \vdash \Delta}\,{*_\mathrm{L}}}{\mathcal{G};\Gamma;\epsilon : e_1 \neq e_2;\epsilon : \exists u.e_1 \mapsto u * ls(u,e_2) \vdash \Delta}\,{\exists_\mathrm{L}}}{\mathcal{G};\Gamma;\epsilon : ls(e_1,e_2) \vdash \Delta}\,\mathrm{ls_L}$$

2. Derivation of LS_2

$$\dfrac{\dfrac{}{\mathcal{G};\Gamma \vdash \epsilon : e = e}\,{=_\mathrm{R}} \qquad \dfrac{}{\mathcal{G};\Gamma \vdash \epsilon : \mathrm{I};\Delta}\,\mathrm{I_R}}{\mathcal{G};\Gamma \vdash \epsilon : e = e \wedge (\exists u.e \mapsto u * ls(u,e));\Delta}\,{\wedge_\mathrm{R}}}{\mathcal{G};\Gamma \vdash \epsilon : ls(e,e);\Delta}\,\mathrm{ls_{R_1}}$$

3. Derivation of LS_3

$$\dfrac{\dfrac{\mathcal{G};\Gamma;h : \mathrm{I} \vdash \Delta}{\mathcal{G};\Gamma;h : e = e;h : \mathrm{I} \vdash \Delta}\,{=_\mathrm{L}} \qquad \dfrac{\dfrac{}{\mathcal{G};\Gamma;h : \exists u.e \mapsto u * ls(u,e) \vdash h : e = e;\Delta}\,{=_\mathrm{R}}}{\mathcal{G};\Gamma;h : e \neq e;h : \exists u.e \mapsto u * ls(u,e) \vdash \Delta}\,{\neg_\mathrm{L}}}{\mathcal{G};\Gamma;h : ls(e,e) \vdash \Delta}\,\mathrm{ls_L}$$

4. Derivation of LS_4

$$\dfrac{\dfrac{\mathcal{G};\Gamma[\,e \mapsto nil\,];h : \mathrm{I} \vdash \Delta[\,e \mapsto nil\,]}{\mathcal{G};\Gamma;h : nil = e;h : \mathrm{I} \vdash \Delta}\,{=_\mathrm{L}} \qquad \dfrac{\dfrac{\dfrac{}{h_1h_2 \triangleright h;\mathcal{G};\Gamma;h : e \neq nil;h_1 : nil \mapsto u;h_2 : ls(u,e) \vdash \Delta}\,\mathrm{NIL}}{\mathcal{G};\Gamma;h : e \neq nil;h : nil \mapsto u * ls(u,e) \vdash \Delta}\,{*_\mathrm{L}}}{\mathcal{G};\Gamma;h : e \neq nil;h : \exists u.nil \mapsto u * ls(u,e) \vdash \Delta}\,{\exists_\mathrm{L}}}{\mathcal{G};\Gamma;h : ls(nil,e) \vdash \Delta}\,\mathrm{ls_L}$$

5. The other cases are similar.

B Validity of (ALE)

Let us prove the validity of the following (ALE) entaiment:

$$(ALE) \stackrel{\text{def}}{=} \quad ls(x,y) \models \neg(y \mapsto z * \top)$$

1. Trivial case: $|h| = 0$
   We simply show that $(s,h) \not\models y \mapsto z * \top$.
   Let us suppose that $(s,h) \models y \mapsto z * \top$. Then, there are wo heaps h_1 and h_2 such that $h_1 \# h_2$, $h = h_1 \cdot h_2$, $(s,h_1) \models y \mapsto z$ and $(s,h_2) \models \top$. Therefore $|h| = 1 + |h_2|$, which implies $|h| > 0$, a contradiction to the assumption that $|h| = 0$ in the trivial case. Consequently, $(s,h) \not\models y \mapsto z * \top$.

2. Inductive case: $|h| = n$ with $n > 0$
We use the following induction hypothesis:

$$\forall h.\, \forall x, y, z.\, \text{if } |h| < n \text{ then } (s,h) \models ls(x,y) \text{ implies } (s,h) \not\models y \mapsto z * \top$$

Let us now suppose that $(s,h) \models ls(x,y)$. We show that $(s,h) \models y \mapsto z * \top$.
Since $|h| > 0$ implies $(s,h) \not\models \mathrm{I}$, by definition of ls, $(s,h) \models ls(x,y)$ implies:

$$\begin{array}{ll} & (s,h) \models x \neq y \wedge \exists u.\, x \mapsto u * ls(u,y) \\ \Leftrightarrow & (s,h) \models x \neq y \text{ and } (s,h) \models \exists u.\, x \mapsto u * ls(u,y) \\ \Leftrightarrow & (s,h) \models x \neq y \text{ and } (s[u \mapsto v], h) \models x \mapsto u * ls(u,y) \\ \Leftrightarrow & (s,h) \models x \neq y \text{ and } \exists h_1, h_2.\, h_1 \# h_2, h = h_1 \cdot h_2,\, (s[u \mapsto v], h_1) \models x \mapsto u, \\ & \qquad\qquad\qquad\qquad \text{and } (s[u \mapsto v], h_2) \models ls(u,y) \end{array}$$

From $(s[u \mapsto v], h_1) \models x \mapsto u$, we obtain $|h_1| = 1$. From $h = h_1 \cdot h_2$, we obtain $|h| = |h_1| + |h_2| = 1 + |h_2|$, and thus $|h_2| < h$. From $(s[u \mapsto v], h_2) \models ls(u,y)$, by induction hypothesis, we obtain $(s[u \mapsto v], h_2) \not\models y \mapsto z * \top$.
From $(s[u \mapsto v], h_2) \not\models y \mapsto z * \top$, we obtain $(s,h_2) \not\models y \mapsto z * \top$. Therefore, since $h = h_1 \cdot h_2$, the only way to have $(s,h) \models y \mapsto z * \top$ would be that $(s,h_1) \models y \mapsto z$, which cannot be the case because

- $(s,h) \models x \neq y$ implies $s(x) \neq s(y)$ and
- $(s[u \mapsto v], h_1) \models x \mapsto u$ implies that $s(x)$ is the only address in the domain of the heap h_1.

We can then conclude that $(s,h) \not\models y \mapsto z * \top$.