High-Level Properties*

Daniel Méry

September 2015

1 Introduction

Separation Logic (SL) is a logic for reasoning about programs that use pointers to manipulate and mutate (possibily shared) data structures [15, 20]. 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) [19], 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.

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 since SL is not recursively enumerable [3, 7], which means that no proof-system can be sound, finite and complete. Besides, as we shall see later in this report, the subtle interactions between the connectives (mostly $-\!\!*$, $*$ and $\mapsto$) makes intuitive reasoning in SL a highly error-prone activity. Therefore, 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.

In Section 2 we recall the basic notions about the syntax and semantics of the heap model of SL and illustrate how it can be used as an assertion language to specify (low- and high-level) properties about mutable data structures. Section 3 then gives a summary of the decidability status of various popular fragments of SL. In Section 4 we

* Research report of the ANR project DynRes (project No. ANR-11-BS02-011).

give an overview of the most significant works about theorem-proving in SL without inductive predicates. Sections 5 and 6 are respectively dedicated to the two most significant contributions to theorem proving in SL recently, namely, Brotherston's purely syntactic proof-system $\text{Cyclist}_{\text{SL}}$ [5], which allows the definition of arbitrary inductive predicates through production rules à la Martin-Löf but only addresses a small subset of SL connectives, and Hou & Gore & Tiu's labelled proof-system LS_{SL} [14], which supports the full set of SL connectives but does not allow the definition of arbitrary inductive predicates, though it has support for inductive data structures such as acyclic list-segments and binary trees through specifically devised rules, an approach that does not scale well given the great variety of data structures encountered in real-life programs. In Section 7 we present (yet) unpublished material about GM_{SL}, our new labelled proof-system developed in the context of Task 1.3 of the ANR project DynRes. GM_{SL} supports the full set of SL connectives and allows the definition of arbitrary inductive predicates, which is a highly non-trivial result. GM_{SL} currently is final-stage work in progress that needs some polishing and should be ready for submission by the end of the year.

2 Separation Logic

Separation Logic (SL) is a concrete model of the boolean variant of BI called Boolean BI (BBI) [16] 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 [20] (which was called "Pointer Logic" back then) without the machinery of pointer arithmetic.

2.1 The Heap Model

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. The heap the domain of which is empty is called the empty heap and is denoted ϵ. 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 the "points-to" predicate $\mapsto$, the equality predicate $=$, the connectives of BI and the existential quantifier. It is defined as follows:

- $P ::= e \mapsto e_1, e_2 \mid e_1 = e_2$ 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.

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* $[\![e_1]\!]_s = [\![e_2]\!]_s$
- $(s, h) \models e \mapsto e_1, e_2$ *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* $\exists v \in Val.\ ([\, s \mid u \mapsto v \,], h) \models A$

In the previous definition, the notation $[\, s \mid u \mapsto v \,]$ 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 $\models F \to G$.

2.2 Interpreting Formulas

In the heap model, the worlds are heaps (collections of cells in storage) and the multiplicative conjunction $A * B$ (often called "star") is true just when the current heap can be separated[1] (or split) into two components, the first one making A true, the second one making B true. The multiplicative implication $A \mathrel{-\!\!*} B$ (often called "magic

[1] Hence the name of the logic.

wand") talks about *heap extension*, *i.e.*, fresh pieces of heaps disjoint from the current heap. More precisely, the magic wand says that whenever a fresh heap disjoint from the current heap makes A true, the combined fresh and current heap makes B true. The other connectives are interpreted pointwise.

The formula $(x \mapsto 3, 5) * ((x \mapsto 7, 5) \mathrel{-\!\!*} P)$ is a good example of "update as heap extension". It states that the current heap contains a cell located at address x holding the pair $\langle\, 3, 5 \,\rangle$ and that, should we update the car of this pair to 7, the formula P would hold in the resulting heap. Indeed, the multiplicative conjunction $*$ splits the heap in two parts, a first one where $(x \mapsto 3, 5)$ holds and a second one where the location x is dangling. The semantics of $\mathrel{-\!\!*}$ and $\mapsto$ then ensures that P must be true when the second heap is extended with a fresh and disjoint heap that binds x to a location containing the pair $\langle\, 7, 5 \,\rangle$.

2.3 Separation Logic as an Assertion Language

The actual use of SL is as an assertion language to state invariants, pre- and post-condition in Hoare triples [19]. For example, consider an operation that disposes of memory by creating dangling pointers through the command $dispose(e)$ which deallocates a location. From a semantic point of view, it removes a location from the heap and it can be defined by the following axiom:

$$\begin{array}{c} \{\, P * \exists\, u_1\, u_2.\, e \mapsto u_1, u_2 \,\} \\ dispose(e) \\ \{\, P \,\} \end{array}$$

where u_1, u_2 are not free in e. Reasoning backwards from $\top$ we can find cases under which a program is safe to execute. With a double dispose we obtain $\bot$ for the precondition as expected, indicating that the program is not safe to execute for any start state:

$$\begin{array}{c} \{\, \bot \,\} \\ \{\, \top * \exists\, u_1\, u_2.\, x \mapsto u_1, u_2 * \exists\, u_3\, u_4.\, x \mapsto u_3, u_4 \,\} \\ dispose(x) \\ \{\, \top * \exists\, u_1\, u_2.\, x \mapsto u_1, u_2 \,\} \\ dispose(x) \\ \{\, \top \,\} \end{array}$$

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 [19]. 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. For example, the usual definition for 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 goes 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.

2.4 Separation Logic and High-Level Properties

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, which is evidence that one should be very careful when reasoning in SL as it is not at all trivial and highly error prone. For a simple counter-model, let us consider two locations ℓ_1 and ℓ_2, a stack s and two singleton heaps h_1 and h_2 heaps such that

$$s = [\, x \mapsto \ell_1, x' \mapsto \ell_2, y \mapsto \ell_1\,],\ h_1(\ell_1) = \ell_2 \text{ and } h_2(\ell_2) = \ell_1$$

The heaps h_1 and h_2 respectively represent the two acyclic list segments $\ell_1 \to \ell_2$ and $\ell_2 \to \ell_1$. Therefore, both $(s, h_1) \models ls(x, x')$ and $(s, h_2) \models ls(x', y)$ hold. Since h_1 and h_2 are disjoint, $h = h_1 \cdot h_2$ is defined and such that $(s, h) \models ls(x, x') * ls(x', y)$. However, since h represents a cycle $\ell_1 \rightleftarrows \ell_2$, h is not the empty heap, but then it cannot be the case that $(s, h) \models ls(x, y)$ since otherwise, by definition of the ls predicate, $ls(\ell_1, \ell_1)$ would imply that h should satisfy I and thus be the empty heap, a contradiction.

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. Therefore, one needs to strengthen (LC) so as to prevent panhandle lists in order to obtain a valid high-level property about concatenation of list segments in the presence of acyclicity. The corresponding entailment is the following:

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

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 $-\!\!*$ implies that y cannot be an address occurring in $ls(x, x')$. Let us note that the entailment would not remain valid without $\neg$I enforcing the non-emptyness of $ls(y, y')$ since the non-emptyness of $ls(y, y')$ is what ensures that y is an (allocated) address.

3 Decidability of Separation Logic

The quantifier-free fragment of SL is decidable, but full SL is not [7]. Full SL is not even recursively enumerable, so that no proof-system for SL can be finite, sound and complete at the same time. Restricting the points-to predicate to only one field gives the variant called 1SL, which is shown to be equivalent to second order logic and therefore undecidable [3]. Restricting 1SL further so as to allow quantification on only one variable (with finitely many program variables), 1SL is decidable [9]. However, allowing two quantified variables instead of one, 1SL becomes undecidable once again [10].

Abstract Separation Logic (ASL) is SL with no equality or points-to predicates, but with propositional variables instead. Heaps and stacks are replaced with worlds arranged in a Kripke style semantics based on relational or partial monoidal structures. ASL can be viewed as a variant of Boolean BI where world composition reflects the usual properties of heap composition in the heap model of SL: identity, commutativity, associativity, partial determinism, cancellativity, indivisibility of the unit, disjointness and cross-split (if a heap can be split in several distinct ways, there are heaps that are intersections of theses splittings). ASL is not decidable, even at the propositional level [6, 16].

A very widely used fragment of SL is the *symbolic heaps* fragment [2] (also called the Π/Σ fragment), which is defined as follows:

$$\begin{array}{ll} P ::= e = e' \mid \neg P & \Pi ::= \top \mid P \mid \Pi \wedge \Pi \\ S ::= e \mapsto [\, f : e \,] & \Sigma ::= \mathrm{I} \mid S \mid \Sigma * \Sigma \end{array}$$

Compared with the original Reynold's semantics, the points-to predicate has a list of named fields on its right-hand-side (f being the name of the field, e being its contents). Symbolic heaps S are pairs $\Pi \wedge \Sigma$ and entailments $S \vdash S'$ are restricted to symbolic heaps. The popularity of the symbolic heaps fragment comes from its decidability, which is why most of the model-checking tools developed for SL are designed to work with this fragment (SmallFoot being the first and most famous one [1]), as one can devise sound, complete and terminating procedures.

Unfortunately, decidability also comes at the expense of expressivity. Since multiplicative implication $-\!\!*$ has been left out, the symbolic heaps fragment cannot express properties about heap extension. Moreover, most of the induction hypotheses given in the literature for proving properties about programs that manipulate inductive data structures (for example properties about list concatenation) require the $-\!\!*$ connective. Even without considering such formulas, the symbolic heaps fragment cannot express many useful properties about heaps (for example cross-split or partial determinism) that are used is ASL to distinguish classes of models and variants of the logic.

4 Theorem Proving in Separation Logic

Most of the works and tools (SmallFoot, SpaceInvader, VeriStar, Asterix) developed for SL focus on model checking in the symbolic heaps fragment, which is decidable but has limited expressive power. There is little work about theorem proving in SL, even more so when one considers fragments beyond the symbolic heaps.

A first approach to theorem proving in a decidable fragment of SL that goes beyond the symbolic heaps fragment can be found in [8], which describes first-order translations of SL. BBI can also be directly translated to first-order logic [13]. Unfortunately, first-order translations cannot really be counted as proof-search methods in SL itself and are moreover not efficient with current first-order theorem provers.

The first actual attempt at proof-search in SL is the Galmiche & Mery's tableau proof-system $\mathrm{T_{SL}}$ [12]. The fragment considered in [12] is called SLP and discards quantifiers and equality. As a further restriction, SLP only allows locations in the left-hand-side of the points-to predicate (while Reynold's semantics allows arbitrary values such as *nil*). The authors then explain how to extend the tableau system to deal with quantifiers and equality, thus sacrificing completeness. The $\mathrm{T_{SL}}$ tableau system uses labels and constraints arranged into a structure called a *resource graph* that keeps track of the relations betweens heaps that must be satisfied by a given formula for it to be valid. Validity is characterised by two distinct notions of *logical* and *structural* consistency. Logical consistency simply corresponds to the branch-closing conditions of the tableau system and ensures that the resource graph actually represents a forcing relation as prescribed by the SL heap model (*e.g.*, no actual heap can force $\bot$). Structural consistency ensures that the resource graph actually represents a feasible (realisable) heap structure and uses the concept of *heap measures* (functions setting minimal realisable sizes for the heaps represented in the resource graphs) and *points-to distributions* (functions describing which points-to predicates should be forced by individual heap cells of the heaps represented in the resource graph). Because of this extra machinery, $\mathrm{T_{SL}}$ might be hard to automate and there is currently no tools implementing it. Furthermore, $\mathrm{T_{SL}}$ has no support for inductive predicates and can therefore only state low-level properties about memory states, but cannot directly be used to state high-level properties.

Another work based on capturing relations between heaps inside a graphical structure is the $\mathrm{P_{SL}}$ labelled sequent proof-system by Lee & Park [17]. $\mathrm{P_{SL}}$ is designed to deal with full SL and thus has rules for the multiplicative implication. As with the Galmiche & Mery's tableau system, the points-to predicate only allows locations on its left-hand-side. Dealing with full SL, $\mathrm{P_{SL}}$ necessarily has to be incomplete. Sadly, it is also unsound as the rule for combining heaps (the Disj$-\!\!*$ rule) relies on the wrong assumption that two heaps with no common subheap should be disjoint, while in Reynold's semantics two heaps with intersecting domains might share no common subheap. For example, the singleton heaps h_1 and h_2 such that $h_1(\ell_1) = a_1$ and $h_2(\ell_1) = a_2$ share no common subheap (since a_1 and a_2 denote distinct atoms) and are nevertheless not disjoint (since they both have location ℓ_1 in their domain).

$$\frac{}{F \vdash F}\,\mathrm{Id} \qquad \frac{}{\bot * F \vdash G}\,\bot_{\mathrm{L}} \qquad \frac{}{F \vdash \top}\,\top_{\mathrm{R}} \qquad \frac{}{F \vdash x = x}\,=_{\mathrm{R}}$$

$$\frac{}{x = y * x \neq y * F \vdash G}\,=_{\mathrm{L}} \qquad \frac{}{x \overset{1}{\mapsto} y * x \overset{1}{\mapsto} z * F \vdash G}\,\overset{1}{\mapsto}$$

$$\frac{}{x \overset{2}{\mapsto} y_1, y_2 * x \overset{2}{\mapsto} z_1, z_2 * F \vdash G}\,\overset{2}{\mapsto} \qquad \frac{F \vdash H \qquad H \vdash G}{F \vdash G}\,\mathrm{Cut}$$

$$\frac{F \vdash G}{\mathrm{I} * F \vdash G}\,\mathrm{I}_{\mathrm{L}} \qquad \frac{F \vdash G}{F \vdash G * \mathrm{I}}\,\mathrm{I}_{\mathrm{R}} \qquad \frac{F_1 \vdash G_1 \qquad F_2 \vdash G_2}{F_1 * F_2 \vdash G_1 * G_2}\,*$$

$$\frac{F_1 * F \vdash G \qquad F_2 * F \vdash G}{(F_1 \vee F_2) * F \vdash G}\,\vee_{\mathrm{L}} \qquad \frac{F \vdash G_i * G}{F \vdash (G_1 \vee G_2) * G}\,i \in \{1,2\}\vee_{\mathrm{R}}$$

Figure 1: Basic proof-rules in $\mathrm{Cyclist}_{\mathrm{SL}}$. Double-lines indicate proof-rules with interchangeable premiss and conclusion.

5 Brotherston's $\mathrm{Cyclist}_{\mathrm{SL}}$ System

A fully syntactic approach to proof-search in SL is Brotherston's $\mathrm{Cyclist}_{\mathrm{SL}}$ proof-system and its associated automated theorem-prover Cyclist [5]. Brotherston's system deals with a fragment that only includes equality and inequality, two forms of the points-to predicate (with one and with two fields on the right-hand-side), additive disjunction and multiplicative conjunction. Explicit quantification, the multiplicative conjunction as well as additive implication, negation and conjunction are left out. More formally, $\mathrm{Cyclist}_{\mathrm{SL}}$ formulas are given by the following grammar:

$$F ::= \top \mid \bot \mid x = y \mid x \neq y \mid \mathrm{I} \mid x \overset{1}{\mapsto} y \mid x \overset{2}{\mapsto} y, z \mid F \vee F \mid F * F \mid P(\mathrm{x})$$

where x, y, z range over variables in Var, P ranges over a fixed finite set of predicate symbols and x ranges over tuples of variables of appropriate length to match the arity of P. The rules of the $\mathrm{Cyclist}_{\mathrm{SL}}$ proof-system are given in Figure 1.

Although it addresses only a rather restricted fragment of SL, $\mathrm{Cyclist}_{\mathrm{SL}}$ is interesting because it supports generic definitions of arbitrary inductive predicates through the notion of *inductive rules sets* which are sets of rules $F \Rightarrow P(\mathrm{x})$ in the style of Martin-Löf productions where F and $P(\mathrm{x})$ are formulas, with P a predicate symbol. An annotated rule $F \overset{\mathrm{z}}{\Rightarrow} P(\mathrm{x})$ means that gathering all the free variables in F and in x exactly results in the tuple z. For example, the predicate ls that represents singly-linked list segments can be defined as follows:

$$\mathrm{I} \overset{x}{\Rightarrow} ls(x,x) \qquad\qquad x \overset{1}{\mapsto} z * ls(z,y) \overset{x,y,z}{\Rightarrow} ls(x,y)$$

As defined above, the predicate $ls(x,y)$ denotes a singly-linked list segment that starts with a cell located at address x and ends with a cell containing y.

$$\dfrac{\dfrac{}{\mathrm{I} * ls(x,y) \vdash ls(x,y)}\,\mathrm{I_L} \qquad \dfrac{\dfrac{}{x \overset{1}{\mapsto} z \vdash x \overset{1}{\mapsto} z} \qquad (\dagger)\ \underline{ls(z,x')} * ls(x',y) \vdash ls(z,y)}{x \overset{1}{\mapsto} z * \underline{ls(z,x')} * ls(x',y) \vdash x \overset{1}{\mapsto} z * ls(z,y)}\,*}{x \overset{1}{\mapsto} z * \underline{ls(z,x')} * ls(x',y) \vdash ls(x,y)}\,\mathrm{ls_{R_2}}}{(\dagger)\ \underline{ls(x,x')} * ls(x',y) \vdash ls(x,y)}\,\mathrm{ls_L}$$

Figure 2: $\mathrm{Cyclist_{SL}}$ cyclic proof of (LC).

With inductive rules available for a predicate symbol P, the next step in Brotherston's system is to define left- and right-unfolding sequent-style inference rules. Each inductive rule $F \overset{\mathrm{z}}{\Rightarrow} P(\mathrm{x})$ gives rise to a right-unfolding rule and to one premiss (using the left part of the inductive rule) of the single multi-premiss left-unfolding rule. For example, the ls predicate gives rise to the following right- and left-unfolding rules:

$$\dfrac{\Gamma \vdash \Delta * \mathrm{I}}{\Gamma \vdash \Delta * ls(x,x)}\,\mathrm{ls_{R_2}} \qquad \dfrac{\Gamma \vdash \Delta * x \overset{1}{\mapsto} z * ls(z,y)}{\Gamma \vdash \Delta * ls(x,y)}\,\mathrm{ls_{R_1}}$$

$$\dfrac{\Gamma[x/y] * \mathrm{I} \vdash \Delta[x/y] \qquad \Gamma * x \overset{1}{\mapsto} z * ls(z,y) \vdash \Delta}{\Gamma * ls(x,y) \vdash \Delta}\,\mathrm{ls_L}$$

where for $\mathrm{ls_L}$, y gets renamed to x in the left premiss and z (in the right premiss) does not occur free in the conclusion[2].

$\mathrm{Cyclist_{SL}}$ system handles inductive predicates with the notion of *cyclic proofs*. A derivation $\mathcal{D}$ for an initial sequent S is a *pre-proof* if each end sequent B in $\mathcal{D}$ which is not the conclusion of an inference rule can be assigned a sequent C in $\mathcal{D}$ such that $C = B\theta$ for some substitution θ. B is then called a *bud* (in $\mathcal{D}$) and C its *companion*.

In order to be a cyclic proof, a pre-proof $\mathcal{P}$ needs to satisfy the *global trace condition*: for every infinite path in $\mathcal{P}$, there is a trace following some tail of the path that contains infinitely many progress points. The full technical definitions of path, trace and progress points are given in [5]. For brevity, let us simply illustrate the previous notions with an example also given in [5].

Let $\mathcal{D}$ be the derivation depicted in Figure 2, which is a cyclic proof of the (LC) entailment presented in Section 2.4. Let $\mathcal{B}$ be the right-most branch of $\mathcal{D}$. The end sequent B of $\mathcal{B}$ is a bud and the initial sequent S of $\mathcal{D}$ is its companion C (both are marked with a dagger sign to be more easily identified). Let us note that C might not always be equal to S in the general case. Having a bud and a companion in each of its open banches, the derivation $\mathcal{D}$ is a pre-proof. The infinite sequence of sequents obtained by travelling forward on $\mathcal{B}$ from its initial sequent S to its bud B, jumping

[2] The actual conditions on freshness and variable renaming are a bit more complex and have been omitted for simplicity. See [5] for full details.

back to its companion C when reaching B, and cycling all over again towards B is called a path. In each sequent of this path P, we have underlined an ls predicate that follows the path. The infinite sequence formed by those underlined ls predicates that represent the successive transformations of the companion into the bud is called a trace following P. A progress point in a trace following a path is simply a point where the inductive predicate that follows the path is unfolded using its corresponding case-split left-unfolding rule. In our example, the first expansion step is a progress point using the left-unfolding rule ls_L. Since $\mathcal{B}$ is the only open branch and contains a progress point between C and B that is encountered infinitely many times cycling from C to B, the derivation $\mathcal{D}$ satisfies the global trace condition and is therefore a cyclic proof.

The fact that the (LC) entailment is provable in Cyclist_SL implies that the ls predicate represents arbitrary (*i.e.*, not necessarily acyclic) list segments in this system, or it would otherwise be unsound since the (LC) entailment is not valid in SL for acyclic list segments as shown in Section 2.4. However, the (ALC) entailment for acyclic list segments cannot be expressed and thus cannot be proved in the restricted language of Cyclist_SL as it lacks additive conjunction and multiplicative implication.

The ability to deal with arbitrary inductive predicates through the notion of cyclic proofs is the most interesting feature of Brotherston's Cyclist_SL system since most high-level properties used in program verification involve inductive data structures. However, the logical fragment it addresses is very restricted. On that point, let us remark that a previous work considered a more expressive logical fragment, namely, full propositional BI with inductive predicates [4]. Sadly, the paper defines the machinery of inductive rule sets and cyclic proofs using the purely syntactic sequent proof-system for (standard intuitionistic) BI but mistakenly interprets these notions using the pointwise boolean semantics of BBI. This mismatch voids the application to SL developed in the paper through examples using the ls predicate since SL is a concrete model of BBI, which currently has no known purely syntactic sequent proof-system.

6 The LS_SL Labelled Sequent System

A very recent work on theorem proving in SL beyond the symbolic heaps fragment is the LS_SL labelled sequent proof-system by Hou & Goré & Tiu [14]. Their proof-system supports full SL (thus sacrificing completeness) and has been implemented in an automated tool called Separata+. When restricted to the symbolic heaps fragments, the proof-procedure implemented in Separata+ terminates.

LS_SL takes inspiration from the Galmiche & Mery's tableau system and from the Lee & Park's labelled sequent system but incorporates graph relations directly into the sequents instead of maintaining a separate graphical structure. Therefore, sequents in LS_SL 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 $(h_1, h_2 \triangleright h_0)$ meaning that the heap h_0 can be split into two disjoint subheaps h_1 and h_2 or, conversely, that the combination of the two heaps h_1 and h_2 is defined and results in the heap h_0.

Identity and cut:

$$\frac{}{\mathcal{G};\Gamma;h:e_1\mapsto e_2\vdash h:e_1\mapsto e_2;\Delta}\,\mathrm{id} \qquad \frac{}{\mathcal{G};\Gamma;h:e_1\mapsto e_2,e_3\vdash h:e_1\mapsto e_2,e_3;\Delta}\,\mathrm{id}_2$$

$$\frac{\mathcal{G};\Gamma[e_1/e_2]\vdash\Delta[e_1/e_2] \qquad \mathcal{G};\Gamma\vdash h:e_1=e_2;\Delta}{\mathcal{G};\Gamma\vdash\Delta}\,\mathrm{cut}_{=}$$

Logical Rules:

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

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

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

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

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

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

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

Side conditions:
Each label being substituted cannot be ϵ, each expression being substituted cannot be *nil*.
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 3: Logical rules in $\mathrm{LS}_{\mathrm{SL}}$.

$$\frac{(h, \epsilon \triangleright h); \mathcal{G}; \Gamma \vdash \Delta}{\mathcal{G} \vdash \Delta}\ \mathrm{U} \qquad \frac{(h_3, h_5 \triangleright h_0); (h_2, h_4 \triangleright h_5); (h_1, h_2 \triangleright h_0); (h_3, h_4 \triangleright h_1); \mathcal{G}; \Gamma \vdash \Delta}{(h_1, h_2 \triangleright h_0); (h_3, h_4 \triangleright h_1); \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{A}$$

$$\frac{(h_2, h_1 \triangleright h_0); (h_1, h_2 \triangleright h_0); \mathcal{G}; \Gamma \vdash \Delta}{(h_1, h_2 \triangleright h_0); \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{E} \qquad \frac{(\epsilon, \epsilon \triangleright h_2); \mathcal{G}[\epsilon/h_1]; \Gamma[\epsilon/h_1] \vdash \Delta[\epsilon/h_1]}{(h_1, h_1 \triangleright h_2); \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{D}$$

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

$$\frac{(h_1, h_2 \triangleright h_0); \mathcal{G}[h_0/h_3]; \Gamma[h_0/h_3] \vdash \Delta[h_0/h_3]}{(h_1, h_2 \triangleright h_0); (h_1, h_2 \triangleright h_3); \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{P} \qquad \frac{(h_1, h_2 \triangleright h_0); \mathcal{G}[h_2/h_3]; \Gamma[h_2/h_3] \vdash \Delta[h_2/h_3]}{(h_1, h_2 \triangleright h_0); (h_1, h_3 \triangleright h_0); \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{C}$$

$$\frac{(h_5, h_6 \triangleright h_1); (h_7, h_8 \triangleright h_2); (h_5, h_7 \triangleright h_3); (h_6, h_8 \triangleright h_4); (h_1, h_2 \triangleright h_0); (h_3, h_4 \triangleright h_0); \mathcal{G}; \Gamma \vdash \Delta}{(h_1, h_2 \triangleright h_0); (h_3, h_4 \triangleright h_0); \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{CS}$$

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 4: Structural rules in $\mathrm{LS}_{\mathrm{SL}}$.

$\mathrm{LS}_{\mathrm{SL}}$ has two forms of substitutions: one for labels and one for expressions. *Label substitutions* are written $[\, h_1/h'_1, \ldots, h_n/h'_n \,]$ meaning that h'_i gets replaced with h_i. Label substitutions handle equality betweens heaps. *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 handle 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 $\mathrm{LS}_{\mathrm{SL}}$, *nil* is the only value allowed to appear in expressions.

$\mathrm{LS}_{\mathrm{SL}}$ has logical rules that are direct translations of the logical rules of T_{SL}. $\mathrm{LS}_{\mathrm{SL}}$ also comes with a plethora of structural rules to capture the various properties of heap composition (unit, associativity, exchange, disjointness, equality, partial determinism, cancellativity, cross-split) while those properties are captured as a closure operator on labels in T_{SL}. The logical and structural rules of $\mathrm{LS}_{\mathrm{SL}}$ are depicted in Figure 3 and Figure 4. Since negation $\neg A$ is defined as $(A \to \bot)$, the following rules are easily derivable in $\mathrm{LS}_{\mathrm{SL}}$:

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

Like Brotherston's $\mathrm{Cyclist}_{\mathrm{SL}}$, $\mathrm{LS}_{\mathrm{SL}}$ incorporates two forms of the $\mapsto$ predicate: the first one with one field and the second one with two fields on the right-hand-side. $\mathrm{LS}_{\mathrm{SL}}$

$$\frac{}{\mathcal{G}; \Gamma; \epsilon : e_1 \mapsto e_2 \vdash \Delta} \mapsto_{\mathrm{L}_1} \qquad \frac{(h_1, h_0 \triangleright h_2); \mathcal{G}; \Gamma; h_1 : e_1 \mapsto e_2 \vdash \Delta}{\mathcal{G}; \Gamma \vdash \Delta} \text{ HE}$$

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

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

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

$$\frac{(h_3, h_4 \triangleright h_1); (h_5, h_6 \triangleright h_2); \mathcal{G}; \Gamma; h_3 : e_1 \mapsto e_2; h_5 : e_1 \mapsto e_3 \vdash \Delta \qquad (h_1, h_2 \triangleright h_0); \mathcal{G}; \Gamma \vdash \Delta}{\mathcal{G}; \Gamma \vdash \Delta} \text{ HC}$$

Side conditions:
Each label being substituted cannot be ϵ, each expression substituted cannot be nil.
In $\mapsto_{\mathrm{L}_4}$, $\theta = mgu(\{(e_1, e_3), (e_2, e_4)\})$.
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 5: Pointer rules in $\mathrm{LS}_{\mathrm{SL}}$.

pointer rules for the $\mapsto$ predicate with one field are given in Figure 5. The rules for the $\mapsto$ predicate with two fields are similar. The $\mapsto$ predicates are not restricted to have a location on their left-hand-side but can have arbitrary values as in the original Reynold's semantics, which means that some formulas that are valid in Galmiche & Mery's tableau system and in Lee & Park's labelled sequent system are not valid in $\mathrm{LS}_{\mathrm{SL}}$. One such formula is $\mathrm{I} \rightarrow \neg((e_1 \mapsto e_2) \mathrel{-\!\!*} \neg(e_1 \mapsto e_2))$, which means that if the current heap is empty then it is not possible that extending it with a heap $(e_1 \mapsto e_2)$ could result in anything else that the heap $(e_1 \mapsto e_2)$. In Reynold's semantics $(s, \epsilon) \not\models \neg((e_1 \mapsto e_2) \mathrel{-\!\!*} \neg(e_1 \mapsto e_2))$ is equivalent to $\forall(s, h).\, (s, h) \not\models e_1 \mapsto e_2$, which holds if and only if e_1 is not a location (such as nil for example).

Unlike Brotherston's $\mathrm{Cyclist}_{\mathrm{SL}}$, $\mathrm{LS}_{\mathrm{SL}}$ does not allow the definition of arbitrary inductive predicates. $\mathrm{LS}_{\mathrm{SL}}$ does have support for inductive predicates such as acyclic singly-linked list segments and binary trees, but through a whole bunch of proof-rules[3] specifically devised to handle all the particular aspects of those predicates. The approach to inductive predicates by devising specific sets of rules is certainly bound to have 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,

[3] Eight rules for acyclic singly-linked list segments, six rules for binary trees.

$$\frac{\mathcal{G};\Gamma[e_1/e_2]\vdash\Delta[e_1/e_2]}{\mathcal{G};\Gamma;\epsilon:ls(e_1,e_2)\vdash\Delta}\ \mathrm{LS}_1 \qquad \frac{}{\mathcal{G};\Gamma\vdash\epsilon:ls(e,e);\Delta}\ \mathrm{LS}_2 \qquad \frac{\mathcal{G};\Gamma;h:\mathrm{I}\vdash\Delta}{\mathcal{G};\Gamma;h:ls(e,e)\vdash\Delta}\ \mathrm{LS}_3$$

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

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

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

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

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

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

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 $(h_1,h_3\triangleright h_4)$, $h_3:ad(e_3)$ 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 $(h_1,h_3\triangleright h_4)$, $h_3:ad(e_3)$ 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 6: Rules for acyclic list segments in $\mathrm{LS_{SL}}$.

$$\dfrac{\dfrac{}{(h_3, h_1 \rhd h_4); (h_1, h_2 \rhd h_0); h_1 : ls(x,x'); h_2 : ls(x',y); h_3 : ls(y,y') \vdash h_2 : ls(x',y); h_3 : y = y'; h_3 : \mathrm{I}; h_4 : \bot; h_0 : ls(x,y)}\,\mathrm{id_a}}{(h_3, h_1 \rhd h_4); (h_1, h_2 \rhd h_0); h_1 : ls(x,x'); h_2 : ls(x',y); h_3 : ls(y,y') \vdash h_3 : y = y'; h_3 : \mathrm{I}; h_4 : \bot; h_0 : ls(x,y)}\,\mathrm{LS_8}$$

Π_1

$$\dfrac{\dfrac{}{(h_3, h_1 \rhd h_4); (h_1, h_2 \rhd h_0); h_1 : ls(x,x'); h_2 : ls(x',y); h_3 : \mathrm{I} \vdash h_3 : \mathrm{I}; h_4 : \bot; h_0 : ls(x,y)}\,\mathrm{id_a}}{(h_3, h_1 \rhd h_4); (h_1, h_2 \rhd h_0); h_1 : ls(x,x'); h_2 : ls(x',y); h_3 : ls(y,y) \vdash h_3 : \mathrm{I}; h_4 : \bot; h_0 : ls(x,y)}\,\mathrm{LS_3}$$

Π_2

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1 \qquad \Pi_2}{(h_3, h_1 \rhd h_4); (h_1, h_2 \rhd h_0); h_1 : ls(x,x'); h_2 : ls(x',y); h_3 : ls(y,y') \vdash h_3 : \mathrm{I}; h_4 : \bot; h_0 : ls(x,y)}\,\mathrm{cut_=}}{(h_3, h_1 \rhd h_4); (h_1, h_2 \rhd h_0); h_1 : ls(x,x'); h_2 : ls(x',y); h_3 : ls(y,y'); h_3 : \neg\mathrm{I} \vdash h_4 : \bot; h_0 : ls(x,y)}\,\neg_{\mathrm{L}}}{(h_3, h_1 \rhd h_4); (h_1, h_2 \rhd h_0); h_1 : ls(x,x'); h_2 : ls(x',y); h_3 : ls(y,y') \wedge \neg\mathrm{I} \vdash h_4 : \bot; h_0 : ls(x,y)}\,\wedge_{\mathrm{L}}}{(h_1, h_2 \rhd h_0); h_1 : ls(x,x'); h_2 : ls(x',y) \vdash h_1 : (ls(y,y') \wedge \neg\mathrm{I}) \mathrel{-\!\!*} \bot; h_0 : ls(x,y)}\,\mathrel{-\!\!*}_{\mathrm{R}}}{(h_1, h_2 \rhd h_0); h_1 : ls(x,x'); h_1 : \neg((ls(y,y') \wedge \neg\mathrm{I}) \mathrel{-\!\!*} \bot); h_2 : ls(x',y) \vdash h_0 : ls(x,y)}\,\neg_{\mathrm{L}}}{(h_1, h_2 \rhd h_0); h_1 : ls(x,x') \wedge \neg((ls(y,y') \wedge \neg\mathrm{I}) \mathrel{-\!\!*} \bot); h_2 : ls(x',y) \vdash h_0 : ls(x,y)}\,\wedge_{\mathrm{L}}}{h_0 : (ls(x,x') \wedge \neg((ls(y,y') \wedge \neg\mathrm{I}) \mathrel{-\!\!*} \bot)) * ls(x',y) \vdash h_0 : ls(x,y)}\,*_{\mathrm{L}}}{\vdash h_0 : ((ls(x,x') \wedge \neg((ls(y,y') \wedge \neg\mathrm{I}) \mathrel{-\!\!*} \bot)) * ls(x',y)) \to ls(x,y)}\,\to_{\mathrm{R}}$$

Figure 7: Proof of the (ALC) entailment in $\mathrm{LS_{SL}}$.

with or without cycles...

$\mathrm{LS_{SL}}$ rules for acyclic list segments are depicted in Figure 6. Most of the rules are self explanatory and explained in full details in [14]. Since the ls predicate represents acyclic singly-linked list segments in $\mathrm{LS_{SL}}$, a proof of the (ALC) entailment discussed in Section 2.4 is given in Figure 7. As one can notice, the proof requires the $\mathrm{LS_8}$ rule, which is the most complicated of the eight list rules. It also requires the $\mathrm{cut_=}$ rule, which cannot be eliminated from the proof-system in the presence of inductive predicates. Let us finally remark that $\mathrm{LS_{SL}}$ cannot prove the (LC) entailment, which is not valid for acyclic singly-linked list-segments, as it would otherwise imply the unsoundness of the system.

7 High-Level Properties in DynRes

As seen in the previous sections, proof-search in full SL, even without inductive predicates and Presburger arithmetic, is far from easy, error prone and requires deep understanding of the subtle interactions between the points-to predicate, heap composition and heap extension.

In the context of Task 1.3 of the ANR project DynRes we took Galmiche & Mery's tableau system as a starting point to develop a labelled sequent-style proof-system called $\mathrm{GM_{SL}}$ which, like $\mathrm{LS_{SL}}$ and unlike $\mathrm{Cyclist_{SL}}$, supports the full set of SL connectives and which, unlike $\mathrm{LS_{SL}}$ and like $\mathrm{Cyclist_{SL}}$, has support for arbitrary defined inductive predicates. $\mathrm{GM_{SL}}$ currently is final stage work in progress that needs some polishing and should be ready for submission by the end of the year. A midterm perspective is the implementation of the $\mathrm{GM_{SL}}$ system as an automated theorem prover in order to make comparisons with Separata+ and Cyclist.

The decision to restart from a sequent-style reformulation of the tableau system $\mathrm{T_{SL}}$, though it has no support for inductive predicates, was motivated by three observations:

- firstly, previous work already sketched how to extend $\mathrm{T_{SL}}$ to full SL [12], though various black-corners and completeness issues (but no soundess issues as in Lee & Park's $\mathrm{P_{SL}}$) were left unexplained or simply not fully understood back then;
- secondly, though Brotherston's notions of inductive rule sets and cyclic proofs really were interesting (and were indeed adapted to $\mathrm{GM_{SL}}$), $\mathrm{Cyclist_{SL}}$ addresses only a very small fragment of SL and extending it to full SL would require a label-free sequent proof-system for BBI, which does not currently exist[4];
- thirdly, Hou & al's $\mathrm{LS_{SL}}$ approach of defining specialised sets of rules for each inductive predicate appeared to us as having scalability issues.

7.1 Buds and Companions in $\mathrm{GM_{SL}}$

In this section, we first adapt the notion of cyclic proofs discussed in Section 5 in the context of a labelled sequent system.

Let us first take a careful look at the proof-rules of $\mathrm{Cyclist_{SL}}$ given in Figure 1. We observe that in the absence of the additive conjunction $\wedge$, the left- and right-hand sides of $\mathrm{Cyclist_{SL}}$ sequents can only be constructed (or decomposed) using the multiplicative conjunction $*$. Moreover, the production rule for the inductive case of the ls predicate states that if we assume the existence of an already (inductively) constructed list segment $ls(z, y)$, then combining it with a predicate $x \mapsto z$ using the mutiplicative conjunction $*$ results in a list segment $ls(x, y)$. Now, let h_1 and h_2 be heaps such that $(s, h_1) \models x \mapsto z$ and $(s, h_2) \models ls(z, y)$ for some stack s. By the semantics of $\mapsto$, we know that h_1 has to be a singleton heap, *i.e.*, a heap of size $|h_1| = 1$. Therefore, the heap h_0 such that $(s, h_0) \models ls(x, y)$ necessarily has a size strictly greater than the size of the heap h_2 such that $(s, h_2) \models ls(z, y)$. Indeed, by the semantics of $*$[5], we know

[4] Applications to SL given in [4] do not work because of the confusion between BI and BBI.
[5] Let us note here that using $\wedge$ instead of $*$ would not work at all.

that $h_0 = h_1 \cdot h_2$ and thus $|h_0| = 1 + |h_2|$, which implies $|h_2| < |h_0|$. Therefore, when unfolding a list segment predicate using its left-unfolding rule $\mathrm{ls_L}$, we know for sure that the list segment $ls(z, y)$ occurring in the premiss corresponding to the inductive case has a size strictly lower than the list segment $ls(x, y)$ occurring in the conclusion. In other words, the induction principle behind the inductive premiss of the left-unfolding rule is in fact nothing more than well-founded induction on the size of the heaps in disguise. In a labelled proof-system where labels denote heaps, as it is the case for $\mathrm{T_{SL}}$, $\mathrm{LS_{SL}}$ or $\mathrm{GM_{SL}}$, we can explicitly state conditions on the (strictly decreasing) size of the heaps rather than implicitly relying on the semantics of $*$ and $\mapsto$ to do so. Adding the additive conjunction $\wedge$ is therefore not a big problem for such labelled systems.

Let us now take a look at the global trace condition that validates a pre-proof as a proper cyclic proof. The condition requires that every path from a companion to bud in a pre-proof should contain at least one progress point, *i.e.*, at least one application of the left-unfolding rule of the inductive predicate that follows the path. Cycling from a companion to a bud shall then necessarily contain infinitely many progress points. Indeed, in order to have a valid proof by induction of a property P, it is not enough to show P in the inductive case using a suitable induction hypothesis, it is also required to show P in at least one trivial (minimal) case. This is precisely what the global trace condition requires since the left-unfolding rule of an inductive predicate is the only rule that split the analysis between the trivial and the inductive cases. For the ls predicate, there is exactly one trivial case and one inductive case. Both cases are respectively analysed in the left and right premiss of the $\mathrm{ls_L}$ case-split rule. The left premiss solves the trivial case and, as we discussed in the previous paragraph, the right premiss is a well-founded induction that solves a subproblem of the same nature as the problem represented in the conclusion.

All of the notions illustrated in Section 5 for $\mathrm{Cyclist_{SL}}$ remain the same in $\mathrm{GM_{SL}}$ except for the notions of bud and companion in a pre-proof. In $\mathrm{GM_{SL}}$, the relationship between a bud B and its companion C in a pre-proof is generalised so that B should contain a *subproblem of the same nature* as C. More precisely, there exist a label substitution σ and an expression substitution θ such that $C\theta\sigma \subseteq B$ and the label h associated with the inductive predicate $P(\mathrm{x})$ on which the induction applies in C has a size strictly greater than the size of the label $h\sigma$ associated to $P(\mathrm{x}\theta)$ in B.

7.2 $\mathrm{GM_{SL}}$ Core System

The core of the $\mathrm{GM_{SL}}$ proof-system is obtained by keeping only the logical, structural and pointer rules of $\mathrm{LS_{SL}}$[6] and adding the rules depicted in Figure 8. The rules for negation are given for convenience and easily derivable. The structural rule IU 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

[6] Without the rules for data stuctures such as lists or tree, there is very little difference between $\mathrm{LS_{SL}}$ and $\mathrm{T_{SL}}$ and the translation between the two systems is straightforward.

Identity and cut:

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

Logical rules:

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

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

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

Structural rules:

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

Pointer rules:

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

$$\frac{\begin{array}{l}(h_1,h_5\triangleright h_4);(h_3,h_5\triangleright h_2);\\(h_1,h_2\triangleright h_0);(h_3,h_4\triangleright h_0);\mathcal{G};\Gamma;h_1:e_1\mapsto e_2;h_3:e_3\mapsto e_4\vdash h:e_1=e_3;\Delta\end{array}}{(h_1,h_2\triangleright h_0);(h_3,h_4\triangleright h_0);\mathcal{G};\Gamma;h_1:e_1\mapsto e_2;h_3:e_3\mapsto e_4\vdash h:e_1=e_3;\Delta}\,\mapsto_{\mathrm{L}_7}$$

Side conditions:
Each label being substituted cannot be ϵ, each expression being substituted cannot be nil.
In $\mapsto_{\mathrm{L}_6}$, $\theta = mgu(\{e_2,e_3\})$.

Figure 8: $\mathrm{GM_{SL}}$ additional proof rules. Double-lines indicate proof-rules with interchangeable premiss and conclusion.

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

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

Figure 9: $\mathrm{GM}_{\mathrm{SL}}$ rules for arbitrary list segments.

captures the fact that whevener 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 $h_4 = h_1 \cdot h_5$.

7.3 From Production to Sequent Rules

Beyond the core, let us explain how $\mathrm{GM}_{\mathrm{SL}}$ allows the definition of arbitrary inductive predicates using the notions of inductive rules sets and production rules as presented in Section 5 for $\mathrm{Cyclist}_{\mathrm{SL}}$. Since $\mathrm{GM}_{\mathrm{SL}}$ is able to deal with either arbitrary, or acyclic list segments, from now on and to avoid confusion, we write $\ell\!\!s$ to denote arbitrary list segments[7] and keep writing ls to denote acyclic list segments.

Given a production rule $F \overset{x}{\Rightarrow} P(\mathrm{x})$, let F be its *body* and $P(\mathrm{x})$ be its *head*. Given a finite set of production rules $F_i \overset{x}{\Rightarrow} P(\mathrm{x_i})$ for a predicate symbol P, the first step is to rewrite all rules so that they share the same head $P(\mathrm{y})$, where the tuple y contains only distinct variables, using equalities when necessary. After this step we get production rules of the form $\{\, E_i \,\} \;;\; F_i \overset{z}{\Rightarrow} P(\mathrm{y})$, where $\{\, E_i \,\}$ is a possibly empty set and E_i (when it exists) is an additive conjunction of equalities[8]. Let us call its *guard* the E_i part of a production rule.
For the $\ell\!\!s$ predicate, the initial production rules

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

get rewritten as follows

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

The second step is to bind to an existential quantifier all the free variables of the production rule that occur in its body but not in its head, thus removing these variables from its set of free variables. After this step, we obtain *canonical production rules*, which are production rules of the form $\{\, E_i \,\} \;;\; QF_i \overset{z-u}{\Rightarrow} P(\mathrm{y})$, where u is the tuple

[7] The cursive ℓ reminding us that such list segments might have cycles.
[8] Also moving into E_i all equalities (and inequalities) occurring in the body of the production rule.

containing all the free variables that occur in F_i but not in y, the formula QF_i being defined as follows:

$$QF_i \stackrel{\text{def}}{=} \begin{cases} F_i & \text{if u is empty} \\ \exists \mathrm{u}.\, F_i & \text{otherwise} \end{cases}$$

For the ℓs predicate, this binding step yields

$$\{\, x = y \,\} \;;\; \mathrm{I} \stackrel{x,y}{\Rightarrow} \ell s(x,y) \qquad\qquad \emptyset \;;\; \exists u.\, x \mapsto u * \ell s(u,y) \stackrel{x,y}{\Rightarrow} \ell s(x,y)$$

As in Brotherston's approach, the next step is to define left- and right-unfolding rules, but in a labelled context. Each production rule $\{\, E_i \,\} \;;\; QF_i \stackrel{z-u}{\Rightarrow} P(\mathrm{y})$ gives rise to a right-unfolding rule and to one premiss (using the left-hand-side of the production rule) of the single multi-premiss left-unfolding rule as follows:

$$\frac{\mathcal{G}; \Gamma; h : E_i \vdash h : QF_i; \Delta}{\mathcal{G}; \Gamma \vdash h : P(\mathrm{y}); \Delta}\, \mathrm{P}_{\mathrm{R_i}}$$

$$\frac{\mathcal{G}; \Gamma; h : E_1; h : QF_1 \vdash \Delta \qquad \ldots \qquad \mathcal{G}; \Gamma; h : E_n; h : QF_n \vdash \Delta}{\mathcal{G}; \Gamma; h : P(\mathrm{y}) \vdash \Delta}\, \mathrm{P}_{\mathrm{L}}$$

For the ℓs predicate, this step yields the following proof-rules:

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

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

Generalizing from variables to expressions, we finally obtain the proof-rules depicted in Figure 9 for the arbitratry singly-linked list segments.

Let us remark that if the initial production rules of an inductive predicate contain inequalities in their bodies, then the previous procedure sometimes requires a special "merging" final step that we shall discuss later in Section 7.5 to retain soundness.

7.4 Case Study: List Concatenation

Figure 10 gives a cyclic proof of the (LC) entailment in $\mathrm{GM_{SL}}$ which follows the same pattern as the one given in Figure 2 for $\mathrm{Cyclist_{SL}}$. As explained in Section 2.4, the (LC) entailment is valid in Reynold's semantics for arbitrary list segments but not for acyclic list segments. The bud B and companion C of this cyclic proof are indicated by the (†) marks and respectively take the following forms:

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

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

Π_1

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

Π_2

$$\dfrac{\begin{array}{c}\Pi_1\end{array}\qquad \dfrac{\begin{array}{c}\Pi_2\end{array}\qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{l}(h_3, h_5 \triangleright h_0); (h_4, h_2 \triangleright h_5);\\ (h_3, h_4 \triangleright h_1); (h_1, h_2 \triangleright h_0);\\ (\dagger)\ h_3 : x \mapsto u; h_4 : \underline{\ell\!s(u,x')}; h_2 : \ell\!s(x',y) \vdash h_5 : \ell\!s(u,y)\end{array}}\,\mathrm{E}}{\begin{array}{l}(h_3, h_5 \triangleright h_0); (h_2, h_4 \triangleright h_5);\\ (h_3, h_4 \triangleright h_1); (h_1, h_2 \triangleright h_0);\\ h_3 : x \mapsto u; h_4 : \underline{\ell\!s(u,x')}; h_2 : \ell\!s(x',y) \vdash h_5 : \ell\!s(u,y)\end{array}}\,*_{\mathrm{R}}}{\begin{array}{l}(h_3, h_5 \triangleright h_0); (h_2, h_4 \triangleright h_5);\\ (h_3, h_4 \triangleright h_1); (h_1, h_2 \triangleright h_0);\\ h_3 : x \mapsto u; h_4 : \underline{\ell\!s(u,x')}; h_2 : \ell\!s(x',y) \vdash h_0 : x \mapsto u * \ell\!s(u,y)\end{array}}\,\mathrm{A}}{\begin{array}{l}(h_3, h_4 \triangleright h_1); (h_1, h_2 \triangleright h_0);\\ h_3 : x \mapsto u; h_4 : \underline{\ell\!s(u,x')}; h_2 : \ell\!s(x',y) \vdash h_0 : x \mapsto u * \ell\!s(u,y)\end{array}}\,\exists_{\mathrm{R}}}{\begin{array}{l}(h_3, h_4 \triangleright h_1); (h_1, h_2 \triangleright h_0);\\ h_3 : x \mapsto u; h_4 : \underline{\ell\!s(u,x')}; h_2 : \ell\!s(x',y) \vdash h_0 : \exists u.x \mapsto u * \ell\!s(u,y)\end{array}}\,\ell\!s_{\mathrm{R}_2}}{\begin{array}{l}(h_3, h_4 \triangleright h_1); (h_1, h_2 \triangleright h_0);\\ h_3 : x \mapsto u; h_4 : \underline{\ell\!s(u,x')}; h_2 : \ell\!s(x',y) \vdash h_0 : \ell\!s(x,y)\end{array}}\,*_{\mathrm{L}}}{(h_1, h_2 \triangleright h_0); h_1 : x \mapsto u * \underline{\ell\!s(u,x')}; h_2 : \ell\!s(x',y) \vdash h_0 : \ell\!s(x,y)}\,\exists_{\mathrm{L}}}{(h_1, h_2 \triangleright h_0); h_1 : \exists u.x \mapsto u * \underline{\ell\!s(u,x')}; h_2 : \ell\!s(x',y) \vdash h_0 : \ell\!s(x,y)}\,\ell\!s_{\mathrm{L}}}{(\dagger)\ (h_1, h_2 \triangleright h_0); h_1 : \underline{\ell\!s(x,x')}; h_2 : \ell\!s(x',y) \vdash h_0 : \ell\!s(x,y)}\,*_{\mathrm{L}}}{h_0 : \ell\!s(x,x') * \ell\!s(x',y) \vdash h_0 : \ell\!s(x,y)}\,\rightarrow_{\mathrm{R}}}{\vdash h_0 : (\ell\!s(x,x') * \ell\!s(x',y)) \rightarrow \ell\!s(x,y)}$$

Figure 10: Cyclic proof of $(\ell\!s(x,x') * \ell\!s(x',y)) \rightarrow \ell\!s(x,y)$ in $\mathrm{GM_{SL}}$.

It appears that $C\theta\sigma \subseteq B$ with $\sigma = [\, h_4/h_1, h_5/h_0 \,]$ and $\theta = [\, x \mapsto u \,]$. On one hand, since $\mathcal{G}_B$ contains $(h_3, h_4 \triangleright h_1)$, the fact that Γ_B contains $h_3 : x \mapsto u$ implies that $|h_1| = |h_4| + 1$ and thus $|h_4| < |h_1|$. It then follows from $(h_4, h_2 \triangleright h_5)$ and $(h_1, h_2 \triangleright h_0)$ that $|h_5| < |h_0|$. On the other hand, h_4, h_2 and h_5 are in the same kind of relationship in B as h_1, h_2 and h_0 were in C, more precisely, the heap that represents the list segment on which the induction applies combined with the heap that represents the second list segment results in the heap that represents the concatenation of the two list segments. Therefore, the bud actually contains a subproblem of the same nature as the problem represented by its companion. Furthermore, the underlined $\ell\!\!\triangleleft$ predicates form a trace following the infinite path (cycle) from C to B and this trace has infinitely many progress points since the rule $\ell\!\!\triangleleft_{\mathrm{L}}$ is applied right after the companion, which finally leads to the conclusion that the derivation presented in Figure 10 actually is a cyclic proof in $\mathrm{GM}_{\mathrm{SL}}$.

7.5 Taming Acyclic List Segments

The initial production rules for the ls predicate that represents acyclic singly-linked list segments are the following:

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

where $x \neq y$ is syntactic sugar for $\neg(x = y)$. Applying the first two steps of the procedure described in Section 7.3 we get the following canonical rules C_1 and C_2:

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

which, after we apply the $\neg_{\mathrm{L}}$ rule to get rid of the $\neg$ connective, leads to the following left- and right-unfolding rules:

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

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

However adding those rules to $\mathrm{GM}_{\mathrm{SL}}$ would lead to unsoundess as we could then prove $\vdash h_0 : ls(nil, nil)$ for all heaps h_0 and not just for empty heaps as shown below:

$$\dfrac{\dfrac{}{\vdash h_0 : nil = nil; h_0 : \exists u. nil \mapsto u * ls(u, nil)}\,=_{\mathrm{R}}}{\vdash h_0 : ls(nil, nil)}\,\mathrm{ls}_{\mathrm{R}_2}$$

The unsoundness comes from the fact that the inequality $x \neq y$ can sometimes be absurd, as it is the case for nil. The presence of $x = y$ and $x \neq y$ in the respective guards of the canonical rules C_1 and C_2 entails that, as opposed to what happens with $\ell\!\!\triangleleft$, the only way to produce $ls(z,z)$ is by using C_1 and the only way to produce

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

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

Figure 11: $\mathrm{GM_{SL}}$ rules for acyclic list segments.

$ls(x,y)$ when $s(x)\neq s(y)$ for some stack s is by using C_2. Therefore, to falsify $ls(x,y)$ one needs to make sure that, depending on the status of the equality $x=y$, neither C_1, nor C_2 can produce $ls(x,y)$. This is achieved by merging the two rules $\mathrm{ls_{R_1}}$ and $\mathrm{ls_{R_2}}$ into a single right-unfolding rule $\mathrm{ls_R}$ with two premises as shown below:

$$\frac{\mathcal{G};\Gamma;h:x=y\vdash h:\mathrm{I};\Delta \qquad \mathcal{G};\Gamma\vdash h:\exists u.x\mapsto u*ls(u,y);h:x=y;\Delta}{\mathcal{G};\Gamma\vdash h:ls(x,y);\Delta}\,\mathrm{ls_R}$$

Generalizing from variables to expressions we get the proof-rules of Figure 11.

For simplicity, we used the example of ls which is a binary predicate with two canonical rules in order to explain the final step that merges the two right-unfolding rules of a binary predicate as soon as the corresponding canonical rules respectively contain an equality and its negation. This merging step easily generalises to binary predicates with more than two canonical rules. It also generalises to n-any predicates with a finite number of canonical rules (though less easily and with a handful of big and cumbersome conjunctions and disjunctions of equalities and inequalities).

7.6 Case Study: Acyclic List Extension

Let us consider 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):

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

(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 as shown by the following proof by induction on the size $|h|$ of the heap h.

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:

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

   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 $(s, h) \not\models y \mapsto z * \top$.

A cyclic proof of (ALE) in $\mathrm{GM}_{\mathrm{SL}}$ is given in Figure 12. The bud B and companion C of this cyclic proof are indicated by the (†) marks and respectively take the forms:

$$\begin{array}{ll} B \stackrel{\text{def}}{=} & (h_1, h_5 \triangleright h_4); \mathcal{G}_B; h_4 : ls(u, y); h_1 : y \mapsto z; \Gamma_B \vdash \\ C \stackrel{\text{def}}{=} & (h_1, h_2 \triangleright h_0); h_0 : ls(x, y); h_1 : y \mapsto z \vdash \end{array}$$

It is clear that $C\theta\sigma \subseteq B$ with $\sigma = [\, h_5/h_2, h_4/h_0 \,]$ and $\theta = [\, x \mapsto u \,]$. On one hand, since $\mathcal{G}_B$ contains $(h_3, h_5 \triangleright h_2)$, the fact that Γ_B contains $h_3 : x \mapsto u$ implies that $|h_5| < |h_2|$. It then follows from $(h_1, h_2 \triangleright h_0)$ and $(h_1, h_5 \triangleright h_4)$ that $|h_4| < |h_0|$. Therefore, the bud actually contains a subproblem of the companion. On the other hand, the subproblem is also of the same nature as the problem represented by the companion because h_1, h_5 and h_4 in B are in the same kind of relationship as h_1, h_2 and h_0 were in C, more precisely, the heap that represents the list segment on which the induction applies is the result of the combination of a heap that forces the extension $y \mapsto z$ of that list segment with a heap that forces $\top$. Furthermore, the

$$\dfrac{\dfrac{\dfrac{}{\epsilon : x = y; \epsilon : y \mapsto z \vdash}\mapsto_{\mathrm{L}_1}}{(h_1, h_2 \rhd \epsilon);\ \epsilon : x = y; h_1 : y \mapsto z \vdash}\mathrm{IU}}{(h_1, h_2 \rhd h_0); h_0 : \mathrm{I};\ h_0 : x = y; h_1 : y \mapsto z \vdash}\mathrm{I_L} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{l}(h_1, h_5 \rhd h_4); (h_3, h_5 \rhd h_2);\\ (h_3, h_4 \rhd h_0); (h_1, h_2 \rhd h_0);\\ (\dagger)\ h_3 : x \mapsto u; h_4 : \underline{ls(u,y)}; h_1 : y \mapsto z \vdash h_0 : x = y\end{array}}\mapsto_{\mathrm{L}_7}}{\begin{array}{l}(h_3, h_4 \rhd h_0); (h_1, h_2 \rhd h_0);\\ h_3 : x \mapsto u; h_4 : \underline{ls(u,y)}; h_1 : y \mapsto z \vdash h_0 : x = y\end{array}}*_{\mathrm{L}}}{(h_1, h_2 \rhd h_0); h_0 : x \mapsto u * \underline{ls(u,y)}; h_1 : y \mapsto z \vdash h_0 : x = y}\exists_{\mathrm{L}}}{(h_1, h_2 \rhd h_0); h_0 : \exists u.x \mapsto u * \underline{ls(u,y)}; h_1 : y \mapsto z \vdash h_0 : x = y}\mathrm{ls_L}$$

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

$$\vdash h_0 : ls(x,y) \to \neg(y \mapsto z * \top)$$

Figure 12: Cyclic proof of $(ls(x,y) \to \neg(y \mapsto z * \top))$ in $\mathrm{GM_{SL}}$.

underlined ls predicates form a trace following the infinite path (cycle) from C to B. This trace has infinitely many progress points as the rule $\mathrm{ls_L}$ is applied right after the companion, which finally leads to the conclusion that the derivation presented in Figure 12 actually is a cyclic proof in $\mathrm{GM_{SL}}$.

Let us firstly remark that (ALE) cannot be proved in $\mathrm{GM_{SL}}$ for arbitrary list segments[9] since the left-unfolding rule for arbitrary list segments would not introduce $h_0 : x = y$ on the right-hand-side of its second premiss, which would in turn prevent the final application of the $\mapsto_{\mathrm{L}_7}$ rule in the second branch of the derivation in Figure 12. Let us secondly remark that we can prove that all $\mathrm{LS_{SL}}$ rules for acyclic list segments given in Figure 6 are derivable in $\mathrm{GM_{SL}}$[10], which entails that $\mathrm{GM_{SL}}$ has strictly more proving power than $\mathrm{LS_{SL}}$ since we have seen in Section 7.4 that $\mathrm{LS_{SL}}$ cannot prove the (LC) entailment as it has no rules for arbitrary singly-linked list segments.

[9] Unsoundness of $\mathrm{GM_{SL}}$ would otherwise immediately follow since (ALE) is not valid for arbitrary list segments.

[10] In the presence of the $\mathrm{cut_a}$ rule though.

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] R. Brochenin, S. Demri, and E. Lozes. On the almighty wand. In *Information and Computation* 211:106–137, 2012.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] C. Calcagno and M. Hague. From separation logic to first-order logic. In *8th Int. Conference on Foundations of Software Science and Computational Structures, FoSSaCS'05*, pages 395–409, Edinburgh, Scotland, 2005.

[9] 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.

[10] S. Demri and M. Deters. Two-Variable Separation Logic and Its Inner Circle. *ACM Transaction on Computational Logic*, 16(2):15, 2015.

[11] D. Galmiche and D. Méry. Semantic labelled tableaux for propositional BI without bottom. *Journal of Logic and Computation*, 13(5):707–753, 2003.

[12] D. Galmiche and D. Méry. Tableaux and resource graphs for separation logic. *Journal of Logic and Computation*, 20(1):189–231, 2007.

[13] Z. Hou, A. Tiu, and R. Gore. A labelled sequent calculus for BBI: Proof theory and proof search. In *22th Int. Conference on Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX'2013*, pages 172–187, Nancy, France, 2013.

[14] 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*, to appear, Berlin, Germany, 2015.

[15] 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.

[16] 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.

[17] W. Lee and S. Park. A proof system for separation logic with magic wand. In *41st ACM Symposium on Principles of Programming Languages, POPL'2014*, pages 477–490, New York, USA, 2014.

[18] P.W. O'Hearn and D. Pym. The Logic of Bunched Implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[19] 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.

[20] 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.