Labelled Cyclic Proofs for Separation Logic

Didier Galmiche and Daniel Méry

Université de Lorraine, CNRS, 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. It is successful for program verification as an assertion language to state properties about memory heaps using Hoare triples. 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 are purely syntactic or labelled systems dedicated to some fragments of SL and they mainly allow either the full set of connectives, or the definition of arbitrary inductive predicates, but not both. In this work, we present a labelled proof system, called $\mathrm{G_{SL}}$, that allows both the definition of cyclic proofs with arbitrary inductive predicates and the full set of SL connectives. We prove its soundness and show that we can derive in $\mathrm{G_{SL}}$ the built-in rules for data structures of another non-cyclic labelled proof system and also that $\mathrm{G_{SL}}$ is strictly more powerful than that system.

1 Introduction

Separation Logic (SL) is a logic for reasoning about programs that use pointers to manipulate and mutate (possibly shared) data structures [19,27]. 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) and its boolean version Boolean BI (BBI) [25,26], 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 features of SL (and a significant part of its success) is its built-in ability for *local reasoning*, which allows program specifications to be kept tight 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 somewhat limited interest if such specifications could not be verified or proven. It is therefore very important to provide (preferably efficient) proof methods and automated verification tools for SL. A significant and sustained effort has been

made on that subject recently, all the more since the task is not trivial. Indeed, although the quantifier-free fragment of SL is decidable, full SL is not [6,8,21]. It is not even recursively enumerable, meaning 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 becomes decidable [11]. However, moving from one to two quantified variables makes 1SL undecidable once again [12]. Abstract Separation Logic (ASL), which is SL with no equality or points-to predicates, but with propositional variables, is not decidable, even at the propositional level [6,21].

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 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 in ASL to distinguish classes of models and variants of the logic.

Recent proof systems that go beyond symbolic heaps allow either the full set of connectives, or the definition of arbitrary inductive predicates, but not both. Some of them are purely syntactic (label-free) [4,5] and are derived from bunched logics like BI or Boolean BI (BBI) [25,26]. Some others introduce labels and label relations to capture the specific aspects of the heap model of SL [15,18,22]. Let us note that, even if in some cases like BI, bunched sequents can be related to or translated into labelled sequents [16] (and vice versa), such relationships are usually far from trivial and difficult to define for other (more elaborated) resource logics (like BBI for instance). Therefore the design of proof systems with labelled sequents from proof systems with bunched sequents is not a straightforward task.

In order to address related works more thoroughly and with an appropriate level of understanding, we delay the discussion until Section 3, after the main ideas and technical concepts of SL have been introduced in Section 2.

Our main goal 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. The main contribution of this paper is therefore a new labelled proof system for SL, called $\mathrm{G_{SL}}$, which combines the notions of inductive definition sets and cyclic proofs described in [4,5] with the labelled proof system $\mathrm{LS_{SL}}$ presented in [18] extended with notions such as size constraints inspired from the labelled tableau system $\mathrm{T_{SL}}$ [15].

The paper is organized as follows: 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. We then discuss

the most relevant related works and existing proof systems for SL in Section 3. In Section 4, we follow [4,5] to introduce inductive definition sets for defining arbitrary inductive predicates based on productions à la Martin-Löf. G_{SL}, our new labelled cyclic proof system for SL, is described in Section 5, where we explain how to turn inductive definitions into unfolding proof rules. We also show the soundness of G_{SL} which derives from the local soundness of the proof rules and of the cyclic mechanism. As an illustration of G_{SL}, we proceed in Section 6 with a case study about acyclic list extension using an entailment which is valid in SL but not provable in LS_{SL}. As a final contribution before the conclusion, we show in Section 7 that the built-in rules for data structures in LS_{SL} can be derived in G_{SL}, thus showing that G_{SL} is strictly more powerful than LS_{SL} with data structures.

2 Separation Logic

In this section we present Separation Logic (SL), its use as an assertion language and discuss the validity of some high-level properties.

Separation Logic (SL) is a concrete model of the boolean variant of BI called Boolean BI (BBI) [21] in which worlds are pairs of memory heaps and stacks called *states*. There are many variants of SL. In this section we follow the presentation of SL given in [19] (where it was called "Pointer Logic") without the machinery of pointer arithmetic.

2.1 The Heap Model

Let us consider the set of *values* Val as the set of integers. Val contains two disjoint subsets Loc and $Atoms$. Loc contains an infinite number of *locations* (addresses of memory cells), while $Atoms$ comprises 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*) $s : Var \rightharpoonup_{fin} Val$ is a finite partial 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 with empty domain is called the *empty heap* and is denoted ϵ. We write $Heaps$ and $Stacks$ to denote the sets of all heaps and all stacks respectively. 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:

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

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

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 $[\![v]\!]_s = v$. For ground expressions we frequently drop the s subscript.

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 propositional units and connectives of BI and the existential quantifier. It is defined as follows:

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

where e, e_1 and e_2 are expressions. As usual when not primitive, negation $\neg F$ and disequality $e_1 \neq e_2$ are defined as $F \to \bot$ and $\neg(e_1 = e_2)$ respectively.

From a semantic point of view formulas are interpreted 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* $\exists v \in Val.\ ([\, s \mid u \mapsto v \,], h) \models A$

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 Separation Logic as an Assertion Language

The actual use of SL is as an assertion language to state invariants, pre- and post-conditions in Hoare triples [25]. For example, one can define the command $dispose(e)$ that deallocates a location 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 [25]. 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:

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

Such a formula states that a heap either corresponds to an empty list (a list with identical starting and ending points) if it is empty, or 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.

2.3 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:

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

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:

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

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.

3 Proof Systems for Separation Logic

A significant body of work has been developed on model checking and theorem proving in the symbolic heaps fragment of SL. A first approach to theorem proving in a decidable fragment of SL that goes beyond the symbolic heaps fragment can be found in [9], which describes first-order translations of SL. BBI can also be directly translated to first-order logic [17]. Unfortunately, first-order translations cannot really be considered as proof search methods in SL itself and are moreover not efficient with current first-order theorem provers.

Let us now give an overview of both label-free (purely syntactic) and labelled proof systems for SL which are the most relevant and closely related to the work presented in this paper.

3.1 Purely Syntactic Proof Systems

Very significant works on (label-free) automated inductive theorem proving in SL can be found in [4,5]. Both works rely on the elegant approach of using production rules in the spirit of Martin-Löf [23] to define arbitrary inductive predicates. Such productions then give rise to left and right unfolding proof rules that handle inductive predicates via (purely syntactic) cyclic proof systems based on the principle of infinite descent.

The scope and significance of our contribution in this paper cannot be fully appreciated without a precise understanding of how it relates to [4] and [5].

The (label-free) cyclic proof system described in [5], which is implemented in a tool called $\text{Cyclist}_{\text{SL}}$, only addresses a small fragment of (classical) SL in which the grammar for formulas and inductive definitions discards additive conjunction, multiplicative and additive implications (and thus negation) and all forms of explicit quantification. Our contribution is therefore clearly an improvement w.r.t. [5] since we propose a (labelled) cyclic proof system which supports the full set of SL connectives with more general inductive definitions including additive and multiplicative conjunctions, additive and multiplicative implications and universal quantification.

Our contribution w.r.t. [4] is more difficult to grasp since the logic actually addressed in [4] is not SL, but BBI[2]. The cyclic proof system described in [4] addresses the full set of BBI connectives with inductive definitions as general as the ones we use in this paper.

Since BBI is often presented as a generalization of SL and since [4] explicitly mentions SL in an example involving the definition of a list predicate relying on an ordinary points-to predicate[3], it might be tempting to argue that our results were already contained in [4]. We obviously strongly disagree with such an argument and therefore give more details about how BBI relates to SL.

While it is acceptable as a first approximation to view BBI as an extension of SL, the actual situation is in fact more subtle for at least three reasons.

[2] Let us remark that the title of [4] is misleading as it mentions BI, not BBI, but the semantics actually used in the paper for the additive connectives is indeed classical.

[3] Although this points-to predicate is never actually formally defined in the paper.

Firstly, although BBI and SL both inherit the same set of connectives from BI, SL includes a built-in points-to predicate and interprets its formulas in a concrete model based on a monoid of heaps (introduced in Section 2) which satisfies a long list of very specific properties (*e.g.*, disjointness, cancellativity, partial determinism, indivisibility of the unit, cross-split, etc.) that more abstract models of BBI do not necessarily satisfy. SL is therefore not just a random instance of BBI, but a very specific and quite elaborated one.

Secondly, SL admits both an intuitionistic and a (more widely used) classical variant. Contrary 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 the monotonicity property $((x \mapsto y) * F) \to (x \mapsto y)$ is valid in intuitionistic SL for any formula F, but not in classical SL.

Thirdly, even the points-to predicate is subject to various interpretations when it has more than one field on its right-hand side: papers following [19] (the non-Reynolds' semantics) interpret $e \mapsto e_1, \ldots, e_n$ as a function mapping the location $[\![e]\!]_s$ to a single memory cell which contains the tuple of values $\langle\, [\![e_1]\!]_s, \ldots, [\![e_n]\!]_s \,\rangle$, while papers following [27] (the Reynolds' semantics with pointer arithmetic) interpret $e \mapsto e_1, \ldots, e_n$ as a shorthand for $(e \mapsto e_1) * \ldots * (e + n - 1 \mapsto e_n)$, *i.e.*, as n adjacent memory cells starting at address $[\![e]\!]_s$ containing the sequence of values $[\![e_1]\!]_s, \ldots, [\![e_n]\!]_s$. The two interpretations are not equivalent: the formula $(e \mapsto e_1, e_2) \to (e \mapsto e_1)$ is valid in the non-Reynolds' semantics, but not in Reynolds' semantics.

Since the intention of [4] was never to give a proof system specifically for SL[4], it was only mentioned as a potential application without taking into account any of its specific aspects discussed above. For example, one cannot prove the disjointness property $((x \mapsto y) * (x \mapsto z)) \to F$, which is valid in SL. Of course, one could argue that it would be straightforward to extend the proof system in [4] with suitable proof rules for intuitionistic or classical SL. Actually, such an extension would not be a quick and easy task at all for several reasons.

Firstly, it is not clear whether all properties of SL (*e.g.*, cross-split) are as easy as disjointness or monotonicity (to handle intuitionistic SL) to turn into purely syntactic sequent style proof rules (and there are many of them).

Secondly, and more importantly, let us recall that the core of the cyclic proof system in [4] is LBI, the purely syntactic single-conclusioned bunched sequent calculus developed for standard BI, which is inherently intuitionistic. Using an intuitionistic proof system with a classical semantics does not impair soundness. However, the ability to prove classical theorems that are not also intuitionistic theorems is lost.

It is clearly pointed out in [4] that the use of an intuitionistic sequent calculus with a classical semantics is not guided by philosophical reasons, but by technical ones. One such reason is that, while handling classical theorems[5] in a Hilbert style proof system could straightforwardly be solved with the simple addition of an axiom like $F \vee \neg F$, a proper treatment of classical negation in a

[4] Confirmed by email correspondence with the main author.

[5] Recall that classical SL is the most widely used variant of SL.

Gentzen-style sequent calculus would require the extension to multi-conclusioned sequents. For bunched logics like BBI or BI, devising a purely syntactic Gentzen-style multi-conclusioned sequent calculus is closely related to the definition of an appropriate form of disjunctive multiplication (dual to multiplicative conjunction) to handle bunches on the right-hand side of sequents[6], something which is anything but trivial. Even in the case of the symbolic heap fragment, which does not even have bunches (since it does not allow unconstrained mixing of additive and multiplicative conjunctions), devising sound proof rules for multiplicative conjunction in a purely syntactic cyclic multi-conclusioned proof system is already notoriously hard (see [28] for a detailed discussion). So far and to the best of our current knowledge, even though there exists a display calculus involving multiplicative disjunction[7] [7], there exists no true Gentzen-style purely syntactic multi-conclusioned bunched sequent calculus for BBI or SL.

For all the reasons discussed above, although the proof system in [4] is a significant contribution to the proof theory of BBI, it does not actually cover the case of SL and cannot be straightforwardly extended to do so, which is also why the case of SL had to be treated specifically in [5], but for a drastically smaller fragment of the logic. Therefore, the $\mathrm{G_{SL}}$ proof system presented in this paper is the first labelled sequent calculus that both admits the full set of SL connectives and the definition of reasonably general inductive predicates.

As a concluding remark on [4] and [5], our opinion is that the choice of label-free proof systems to implement the notions of inductive definitions and cyclic proofs unnecessarily restricts their expressive power. We strongly believe that they could more easily reach their full potential in the context of labelled deduction, which is more adapted to handle the full set of SL connectives. Let us now briefly review such labelled proof systems for SL.

3.2 Labelled Proof Systems

The first attempt at (labelled) proof search with the full set of SL connectives is the tableau proof system $\mathrm{T_{SL}}$ by Galmiche & Mery [15]. $\mathrm{T_{SL}}$ deals with a fragment, called SLP, which discards quantifiers and equality. As a further restriction, SLP only allows locations in the left-hand side of the points-to predicate (while Reynolds' semantics allows arbitrary values such as *nil*). In the last pages of [15], the authors briefly sketch 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: *logical* and *structural* consistency. Logical consistency simply corresponds to the branch-closing conditions of the tableau system and ensures

[6] It is not necessarily the case for labelled sequent systems or display calculi.

[7] Let us note that display calculi are great tools for studying theoretical properties like cut-elimination, but that they are also known to be highly inefficient and not well adapted for automated theorem proving.

that the resource graph actually represents a forcing relation as prescribed by SL semantics (*e.g.*, no actual heap can force $\bot$). Structural consistency ensures that the resource graph actually represents a 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). Although $\mathrm{T_{SL}}$ addresses the full set of connectives of SL, it has no support for inductive predicates and can therefore only state low-level properties about memory states.

Another labelled system based on capturing relations between heaps inside a graphical structure is $\mathrm{P_{SL}}$ by Lee & Park [22]. $\mathrm{P_{SL}}$ is designed to deal with full SL and thus has rules for the multiplicative implication. As in $\mathrm{T_{SL}}$, the points-to predicate only allows locations on its left-hand side. Dealing with full SL, $\mathrm{P_{SL}}$ necessarily has to be incomplete. Unfortunately, $\mathrm{P_{SL}}$ 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 Reynolds' semantics two heaps with intersecting domains might share no common subheap. For instance, 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).

The most significant work on labelled theorem proving in SL going beyond symbolic heaps is the $\mathrm{LS_{SL}}$ proof system by Hou, Goré & Tiu [18]. It supports full SL and is thus incomplete. Without the rules for data structures, $\mathrm{LS_{SL}}$ can be seen as a sequent-style reformulation of $\mathrm{T_{SL}}$ where structural aspects (resource graph 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-hand side of points-to predicates. Let us also note that $\mathrm{LS_{SL}}$ has been implemented in a tool called Separata+ with a proof-strategy which guarantees termination when restricted to the symbolic heaps fragment.

Like $\mathrm{Cyclist_{SL}}$, $\mathrm{LS_{SL}}$ has two forms of the points-to predicate (with either one, or two fields on the right-hand side). The left-hand side of a points-to predicate is not restricted to be a location but can be an arbitrary value as in Reynolds' original semantics, which means that some formulas that are valid in $\mathrm{T_{SL}}$ are not valid in $\mathrm{LS_{SL}}$. One such formula is $\mathrm{I} \to \neg((e_1 \mapsto e_2) \mathrel{-\!\!*} \neg(e_1 \mapsto e_2))$, 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 than the heap $(e_1 \mapsto e_2)$. In Reynolds' 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 $[\![e_1]\!]_s$ is not a location (such as *nil* for example).

Unlike $\mathrm{Cyclist_{SL}}$ and the proof systems in [4,5], $\mathrm{LS_{SL}}$ does not allow the definition of arbitrary inductive predicates, but it does have support for data structures such as acyclic singly-linked list segments and binary trees through built-in predicates and dedicated proof rules[8] specifically devised to handle all the particular aspects of those predicates.

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

3.3 Combining Inductive Definition with Labelled Deduction

Let us recall here that our goal in this paper is to combine the approach for arbitrary inductive definitions described in [4,5] with an extension of the $\mathrm{LS}_{\mathrm{SL}}$ labelled proof system presented in [18] to propose a new labelled proof system for SL, called G_{SL}, that allows both the full set of SL connectives and the definition of reasonably general inductive predicates.

Let us also once again remark that this is currently not the case for the proof systems previously discussed.

On one hand, the label-based approach yields proof systems that can easily deal with full SL but with no support for arbitrary inductive definitions. Data structures are then handled through built-in predicates with dedicated sets of proof rules, which is an approach that is likely to scale poorly given the great variety of data structures encountered in actual programs and given the fact that even the simplest inductive data structures such as lists admit a large number 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. To prove (LC), one would have to add a new set of proof rules for the arbitrary list segment predicate.

On the other hand, the purely syntactic approach faces non-trivial technical issues addressing the full set of SL connectives and thus puts quite strong restrictions on the formulas allowed both in the proof system and in the definition of inductive predicates. For example, while (LC) is provable in $\mathrm{Cyclist}_{\mathrm{SL}}$, (ALC) is not expressible "as is" syntactically in $\mathrm{Cyclist}_{\mathrm{SL}}$ as it lacks multiplicative implication and additive conjunction. Although one can sometimes find tricks and workarounds for the lack of expressive power (for instance, (ALC) can be defined using an auxiliary and more general $ls(x, y, z)$ inductive predicate), this is not always possible and still an unnecessary complication.

4 Inductive Definitions

We now follow [5] and [4] to extend SL with inductive definitions in the spirit of Martin-Löf productions.

We first 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 $\mathbf{x}$ as a shorthand for tuples $(x_1, \ldots, x_n)$ and π_i^n for the ith projection function on n-tuples such that $\pi_i^n(x_1, \ldots, x_n) = x_i$.

The small fragment of SL considered in [5] for the $\mathrm{Cyclist}_{\mathrm{SL}}$ proof system restricts formulas to the following grammar:

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

where x, y, z range over variables in Var, P ranges over predicate symbols and $\mathbf{x}$ ranges over tuples of variables of appropriate length to match the arity of P. Formulas are also considered up to associativity and commutativity of $*$ and $\vee$.

Inductive predicates are defined using *inductive definition sets* which are finite sets of *productions*. A *production* is a rule of the form $F \Rightarrow P\mathbf{x}$ where F and $P\mathbf{x}$ are formulas with P a predicate symbol. Since productions are supposed to later give rise to unfolding proof rules in cyclic proof systems it is useful to know what variables occur free in a production. This leads to the notion of *annotated productions* of the form $F \overset{\mathbf{z}}{\Rightarrow} P\mathbf{x}$, which are productions such that $FV(F) \cup \{\mathbf{x}\} = \{\mathbf{z}\}$, where $FV(F)$ denotes the set of free variables occurring in a formula F. Variables occurring in $\mathbf{z}$ but not in $\mathbf{x}$ are supposed to be implicitly existentially quantified.

Let us illustrate the notions with list segments. From now on, we shall write ℓs for arbitrary list segments and let ls denote only acyclic list segments.

Example 1. The ℓs predicate can be defined as follows:

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

Example 2. The ls predicate can be defined as follows:

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

It is stated in [5] that the fragment of separation logic does not include, for example, function symbols, additive conjunction or multiplicative implication because these features are not typically employed in SL verification tools. We only partially agree with this statement. Indeed, the fragment is already powerful enough to define most of the commonly used data structures such as lists (singly or doubly linked) or trees. We also agree that for defining inductive data structures, multiplicative implication is not very useful. However, tools for automated generation of program invariants, especially in the context of backward reasoning from post-conditions, often generate formulas including multiplicative implications. It is therefore useful to have support in the proof system for multiplicative implication at least in the grammar for arbitrary formulas, even if it is not included in the grammar for defining inductive predicates. Moreover, function symbols are useful for defining inductive predicates that do not necessarily represent data structures, for instance, the following N predicate (given in [4]) for representing natural numbers.

Example 3. The N predicate is defined as follows:

$$\top \Rightarrow N(nil) \qquad\qquad N(x) \overset{x}{\Rightarrow} N(s(x))$$

The inductive predicates considered in [4] are significantly more general than the ones in [5] as they allow additive and multiplicative implications and also functions symbols in the definition of terms. To take advantage of this higher level of generality, we now adopt and follow the terminology given in [4] rather than the one given in [5]. For convenience, we only slightly deviate from [4] by not adding an arbitrary set of ordinary (non-inductive) predicates[9]. Our set of

[9] Adding such a set of ordinary predicates would not be particularly difficult.

ordinary predicates is therefore restricted to the ones that are already built into the definition of SL given in Section 2.1, which are interpreted as follows:

- $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = Heaps$,
- $[\![\mathbf{I}]\!] = \{h \in Heaps \mid dom(h) = \emptyset\}$,
- $[\![=]\!] = \{(h, v, v) \in Heaps \times Val^2\}$,
- $[\![\overset{1}{\mapsto}]\!] = \{(h, v_1, v_2) \in Heaps \times Val^2 \mid dom(h) = \{v_1\}, h(v_1) = v_2\}$,
- $[\![\overset{2}{\mapsto}]\!] = \{(h, v_1, v_2, v_3) \in Heaps \times Val^3 \mid dom(h) = \{v_1\}, h(v_1) = \langle\, v_2, v_3 \,\rangle\}$.

It is straightforward to generalize SL expressions to function symbols by extending the signature of SL with interpreting functions $f^M : Val^n \to Val$ for each function symbol f of arity n (see [4] for more details). The interpretation $[\![t(\mathbf{x})]\!]_s$ of a term $t(\mathbf{x})$ containing function symbols is defined as usual by inductively replacing each function symbol in t by its interpretation. Let us however remark that in order to simplify comparisons with other proof systems, the proof system we shall introduce in Section 5 will stick to the original definition of expressions and will not make use of general terms with function symbols.

Definition 2 (Inductive definition set). *An* inductive definition set *for* SL *is a set of* productions *of the form:*

$$C(\mathbf{x}) \Rightarrow P_i\mathbf{t}(\mathbf{x}) \quad i \in \{1, \ldots, n\}$$

where $C(\mathbf{x})$ *is an* inductive clause *given by the following grammar:*

$$\begin{aligned} C(\mathbf{x}) ::= {} & Q\mathbf{t}(\mathbf{x}) \mid P_j\mathbf{t}(\mathbf{x})\,(j \in \{\,1, \ldots, n\,\} \mid \\ & C(\mathbf{x}) \wedge C(\mathbf{x}) \mid C(\mathbf{x}) * C(\mathbf{x}) \mid \hat{F}(\mathbf{x}) \to C(\mathbf{x}) \mid \hat{F}(\mathbf{x}) \mathrel{-\!\!*} C(\mathbf{x}) \mid \forall x C(\mathbf{x}) \end{aligned}$$

where Q *ranges over ordinary predicate symbols and* $\hat{F}(\mathbf{x})$ *ranges over all formulas in which no inductive predicates occur and whose free variables are contained in* $\{\mathbf{x}\}$.

Given a production $B \Rightarrow H$, we call H (the target of the arrow symbol) its *head* and B (the source of the arrow symbol) its *body*.

From now on we assume a fixed finite set Φ of productions.

The *inductive definition set* of an inductive predicate P_i is the set Φ_{P_i} of all the productions whose heads feature P_i. We therefore partition the set Φ into disjoint subsets $\Phi_{P_1}, \ldots, \Phi_{P_n}$. We index with $j \in \{1, \ldots, |\Phi_{P_i}|\}$ the productions $\Phi_{P_{i,j}}$ of each inductive definition set Φ_{P_i}. Such productions should be read as disjunctive clauses of the definition of P_i whose free variables are implicitly existentially quantified. As stated in [4], the following equivalent notation for definitions might be more familiar for some readers:

$$P_i\mathbf{y} =_{def} (\exists \mathbf{x_1}.\mathbf{y} = \mathbf{t_1}(\mathbf{x_1}) \wedge C_1(\mathbf{x_1})) \vee \ldots \vee (\exists \mathbf{x_k}.\mathbf{y} = \mathbf{t_k}(\mathbf{x_k}) \wedge C_k(\mathbf{x_k}))$$

where $\{\,\mathbf{y}\,\} \cap \{\,\mathbf{x_1}, \ldots, \mathbf{x_k}\,\} = \emptyset$ and $C_1(\mathbf{x_1}), \ldots, C_k(\mathbf{x_k})$ are inductive clauses.

Definition 3. *For each predicate symbol P_i with arity a_i defined by the productions $C_j(\mathbf{x_j}) \Rightarrow P_i\mathbf{t_j}(\mathbf{x_j})$ $(j \in \{1,\ldots,k\})$ we obtain a corresponding n-ary function*

$$\varphi_{\Phi_{P_i}} : \wp(Heaps \times Val^{a_1}) \times \ldots \times \wp(Heaps \times Val^{a_n}) \to \wp(Heaps \times Val^{a_i})$$

as follows:

$$\varphi_{\Phi_{P_i}}(\mathbf{X}) = \bigcup_{1 \leq j \leq k} \{(h, [\![\mathbf{t}(\mathbf{v})]\!]) \mid (s[\mathbf{x_j} \mapsto \mathbf{v}], h) \models_{[\![\mathbf{P}]\!] \mapsto \mathbf{X}} C_j(\mathbf{x_j})\}$$

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

Let us note that all variables occurring in C_j on the right-hand side of the set expression but not in the sequence $\mathbf{x_j}$ are implicitly existentially quantified.

Definition 4. *The* definition set operator *for Φ is defined as the operator φ_Φ, with domain and codomain $\wp(Heaps \times Val^{a_1}) \times \ldots \times \wp(Heaps \times Val^{a_n})$ such that $\varphi_\Phi(\mathbf{X}) = (\varphi_{\Phi_{P_1}}(\mathbf{X}), \ldots, \varphi_{\Phi_{P_n}}(\mathbf{X}))$.*

It is proven in [4] that the operator generated from a set of inductive definitions by Definition 4 is monotone and therefore has a least fixed-point that can be iteratively approached by *approximants*.

Firstly, we define a chain of ordinal-indexed sets $(\varphi_\Phi^\alpha)_{\alpha \geq 0}$ by transfinite induction: $\varphi_\Phi^\alpha = \bigcup_{\beta<\alpha} \varphi_\Phi(\varphi_\Phi^\beta)$ (note that this implies $\varphi_\Phi^0 = (\emptyset, \ldots, \emptyset)$). Then for each $i \in \{1,\ldots,n\}$, the set $P_i^\alpha = \pi_i^n(\varphi_\Phi^\alpha)$ 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

$$(s,h) \models P_i\mathbf{t}(\mathbf{x}) \text{ iff } (h, [\![\mathbf{t}(\mathbf{x})]\!]_s) \in [\![P_i]\!].$$

Example 4. The definition set operator for the ℓs predicate is given by:

$$\begin{aligned}\varphi_{\Phi_{\ell s}}(X) = &\{\, (\epsilon, (v,v)) \mid v \in Val \,\} \cup \\ &\{\, (h_1 \cdot h_2, (v,v')) \mid (h_1, (v,v'')) \in [\![\overset{1}{\mapsto}]\!] \text{ and } (h_2, (v'',v')) \in X \,\}\end{aligned}$$

where v'' in the second set comprehension is, implicitly, existentially quantified.

Example 5. The definition set operator for the ls predicate is given by:

$$\begin{aligned}\varphi_{\Phi_{ls}}(X) = &\{\, (\epsilon, (v,v)) \mid v \in Val \,\} \cup \\ &\{\, (h_1 \cdot h_2, (v,v')) \mid v \neq v', (h_1, (v,v'')) \in [\![\overset{1}{\mapsto}]\!] \text{ and } (h_2, (v'',v')) \in X \,\}\end{aligned}$$

where v'' in the second set comprehension is, implicitly, existentially quantified.

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

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

$$\frac{\epsilon h_2 \rhd h_2; \mathcal{G}[h_2/h_1]; \Gamma[h_2/h_1] \vdash \Delta[h_2/h_1]}{\epsilon h_1 \rhd h_2; \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{Eq_1} \qquad \frac{\epsilon h_1 \rhd h_1; \mathcal{G}[h_1/h_2]; \Gamma[h_1/h_2] \vdash \Delta[h_1/h_2]}{\epsilon h_1 \rhd h_2; \mathcal{G}; \Gamma \vdash \Delta}\ \mathrm{Eq_2}$$

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

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

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

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.

Fig. 1. Structural rules in $\mathrm{G_{SL}}$.

5 The $\mathrm{G_{SL}}$ Proof System

In this section, we introduce the labelled sequent-style proof system $\mathrm{G_{SL}}$. We take the core of the $\mathrm{LS_{SL}}$ [18] proof system, *i.e.*, $\mathrm{LS_{SL}}$ without its dedicated proof rules for data structures (acyclic list and binary trees), as our starting point for $\mathrm{G_{SL}}$ because, as we mentioned above, the core of $\mathrm{LS_{SL}}$ can be seen as a sequent-style reformulation of the tableau system $\mathrm{T_{SL}}$ [15] in which in the closure operator on labels and the points-to distributions are translated into explicit structural and pointer rules, with refinements to handle heap extension and values on the left-hand side of points-to predicates. As we shall study relationships between $\mathrm{G_{SL}}$ augmented with inductive predicates and $\mathrm{LS_{SL}}$ with data structures in Section 7, we intentionally keep close to the concepts and notations used in [18].

Let L be the countable set of symbols called *label letters*. The set *Labels* of *labels for* $\mathrm{G_{SL}}$ is defined as $L \cup \{ \epsilon \}$, where ϵ is a label constant not in L. We use the same letter (h possibly subscripted) to range over both labels and heaps since, as we shall see later more formally, labels in $\mathrm{G_{SL}}$ are meant to be semantically interpreted as heaps in SL. The label constant ϵ should be interpreted as the empty heap.

Let us remark that in $\mathrm{G_{SL}}$ (like in $\mathrm{LS_{SL}}$) all labels are atomic. Indeed, label composition (like in $\mathrm{T_{SL}}$) is replaced in $\mathrm{G_{SL}}$ (and in $\mathrm{LS_{SL}}$) with *label relations* which are expressions of the form $h_1h_2 \rhd h_0$, where h_1, h_2, h_0 are labels. Label relations syntactically reflect the ternary relation used to generalize heap compo-

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

Fig. 2. Logical rules in $\mathrm{G_{SL}}$.

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

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

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

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

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

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

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

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.

Fig. 3. Pointer rules in $\mathrm{G_{SL}}$.

sition in the relational semantics of more abstract extensions of SL (*e.g.*, BBI). Intuitively, $h_1h_2 \rhd h_0$ means that the heap represented by the label h_0 can be split into two disjoint subheaps represented by the labels h_1 and h_2 (or conversely that combining the two heaps represented by h_1 and h_2 yields the heap represented by h_0).

A *labelled formula* is a pair (h, F), written $h : F$, where F is a formula of SL and h is a label. The sequents of $\mathrm{G_{SL}}$ take the form $\mathcal{G};\Gamma \vdash \Delta$, where Γ and Δ are multi-sets of labelled formulas, and $\mathcal{G}$ is a multi-set of label relations.

Let us now explain more precisely how labels, label relations and labelled formulas are semantically interpreted in the context of labelled sequents.

Definition 5 (Realization). *Given a labelled sequent $S = \mathcal{G};\Gamma \vdash \Delta$, a* label mapping *for S is a function ρ mapping each heap label in S to an actual heap of the heap model of* SL *such that:*

$$\rho(\epsilon) = \epsilon \text{ and for all } h_i h_j \triangleright h_k \in \mathcal{G},\ \rho(h_i) \cdot \rho(h_j) = \rho(h_k).$$

A realization *for S is a pair (s, ρ) where s is a stack and ρ is a label mapping for S such that: for all $h_i : F \in \Gamma$, $(s, \rho(h_i)) \models F$ and for all $h_i : F \in \Delta$, $(s, \rho(h_i)) \not\models F$.*
A labelled sequent S is realizable *if there is a realization for S.*

In Section 4 we considered inductive predicates with terms that can contain function symbols (for the sake of generality). However, in order to simplify comparisons with $\mathrm{LS_{SL}}$, we restrict the presentation of $\mathrm{G_{SL}}$ in this paper to terms (expressions) without function symbols and where *nil* is the only constant.

The $\mathrm{G_{SL}}$ system 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 between 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.

The core of the $\mathrm{G_{SL}}$ labelled proof system consists of the structural, logical and pointer rules depicted in Figures 1, 2 and 3, which can be viewed as an extension of the core of the $\mathrm{LS_{SL}}$ proof system [18]. 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 $\mathrm{G_{SL}}$ and do not appear in $\mathrm{LS_{SL}}$.

Let us first explain the rules HE and HC (already present in $\mathrm{LS_{SL}}$) which are less intuitive than the other pointer rules. HE (for "heap extension") captures the fact that in SL a heap can always be extended with a new cell. Indeed, the rule states that the heap denoted by h_0 can be extended into a fresh heap (denoted by h_2) by gluing to (the denotation of) h_0 a heap (denoted by h_1) whose unique cell is located at an address (denoted by e_1) which (by the freshness condition on e_1) does not occur in the domain of (the denotation of) h_0. HC (for "heap composition") captures the fact that two heaps (denoted by h_1 and h_2) either are disjoint, and thus composable into a bigger heap (denoted by h_0), or they share at least one cell located at the same address (here denoted by e_1) whose content might or not differ in both heaps. The sharing of a cell by (the denotations of) h_1 and h_2 is captured by stating that they respectively admit one-cell subheaps (denoted by h_3 for h_1 and by h_5 for h_2) with the same domain. Let us remark that for the second premiss of HC to be realizable, $h_1 h_2 \triangleright h_0$ must be realizable, which implies that (the denotations of) h_1 and h_2 should be disjoint (otherwise the denotation of h_0 should not exist).

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 explicitly captured in the remaining structural rules[10]. The structural rule IU explicitly

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

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 $h_4 = h_1 \cdot h_5$.

Let us finally remark that the rules $*_{\mathrm{R}}$ and $-\!\!*_{\mathrm{L}}$ are made reusable as many times as required with distinct label relations by keeping a copy of their principal formula in their premises. This is standard in labelled sequent systems for left rules (resp. right rules) dealing with connectives whose semantic interpretation involves a universal (resp. existential) quantification. On the contrary, all other rules discard their principal formula from their premises.

5.1 Unfolding Rules for Inductive Definitions

We now adapt to our labelled proof system the (purely syntactic) approach used in [4] and [5] to derive proof rules, called *unfolding rules*, from the inductive definitions described in Section 4.

For each $i \in \{1, \ldots, n\}$ and each production $\Phi_{P_{i,j}} \in \Phi_{P_i}$ $(j \in \{1, \ldots, |\Phi_{P_i}|\})$, we obtain a *right unfolding rule* $\mathrm{P}_{\mathrm{iR_j}}$ for the predicate P_i as follows:

$$C(\mathbf{x}) \Rightarrow P_i\mathbf{t}(\mathbf{x}) \qquad \rightsquigarrow \qquad \frac{\mathcal{G}; \Gamma \vdash h : C(\mathbf{u}); \Delta}{\mathcal{G}; \Gamma \vdash h : P_i\mathbf{t}(\mathbf{u}); \Delta}\,\mathrm{P}_{\mathrm{iR_j}}$$

The *left unfolding rule* (also called *case-split rule*) for the predicate P_i is a multi-premiss rule of the form:

$$\frac{\text{case distinctions}}{\mathcal{G}; \Gamma; h : P_i\mathbf{u} \vdash \Delta}\,\mathrm{P}_{\mathrm{iL}}$$

where there are as many case distinctions as there are productions in Φ_{P_i}, each case distinction being derived from a production $\Phi_{P_{i,j}} \in \Phi_{P_i}$ as follows:

$$C(\mathbf{x}) \Rightarrow P_i\mathbf{t}(\mathbf{x}) \quad \rightsquigarrow \quad \mathcal{G}; \Gamma; h : \mathbf{u} = \mathbf{t}(\mathbf{x}) \wedge C(\mathbf{x}) \vdash \Delta \;\; (\forall x \in \mathbf{x}. x \notin FV(\Gamma \cup \Delta))$$

Rewriting some equalities, generalizing from variables to expressions and expanding additive conjunctions on the left-hand side of sequents, the $\mathrm{G_{SL}}$ left and right unfolding rules for the ℓs and ls predicates defined in Examples 1 and 2 are the rules depicted in Figure 4. Let us remark that to avoid writing cumbersome side conditions about the instantiation of free variables, we performed an explicit existential closure on the otherwise implicitly existentially quantified free variables (*i.e.*, the variables occurring in the body of a production, but not in its head).

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

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

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

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

Fig. 4. G_{SL} rules for list segments.

5.2 Labelled Cyclic Proofs

The labelled system G_{SL} handles induction with the notion of *cyclic proofs*. Therefore, we reuse the notions of *buds*, *companions*, *pre-proofs*, *paths* and *traces* used in [4,5] and adapt them in the context of a labelled proof system.

Let $S_1 = \mathcal{G}_1; \Gamma_1 \vdash \Delta_1$ and $S_2 = \mathcal{G}_2; \Gamma_2 \vdash \Delta_2$ be two labelled sequents. We define the inclusion of sequents as follows:

$$S_1 \subseteq S_2 \text{ holds} \quad \text{iff} \quad \mathcal{G}_1 \subseteq \mathcal{G}_2, \Gamma_1 \subseteq \Gamma_2 \text{ and } \Delta_1 \subseteq \Delta_2 \text{ hold.}$$

Definition 6 (Pre-proof). *Let $\mathcal{D}$ be a derivation in G_{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 $C\theta\sigma \subseteq B$, 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.*

Definition 7 (Path). *A* path *in a pre-proof $(\mathcal{D}, \mathcal{R})$ is a sequence of labelled sequent occurrences $(\mathcal{G}_i; \Gamma_i \vdash \Delta_i)_{i \geq 0}$ such that, for all $i \geq 0$, either*

- *$\mathcal{G}_{i+1}; \Gamma_{i+1} \vdash \Delta_{i+1}$ is a premise of the rule instance in $\mathcal{D}$ with conclusion $\mathcal{G}_i; \Gamma_i \vdash \Delta_i$, or*
- *$\mathcal{G}_{i+1}; \Gamma_{i+1} \vdash \Delta_{i+1} = \mathcal{R}(\mathcal{G}_i; \Gamma_i \vdash \Delta_i)$.*

Definition 8 (Trace). *Let $(\mathcal{G}_i; \Gamma_i \vdash \Delta_i)_{i \geq 0}$ be a path in a pre-proof $(\mathcal{D}, \mathcal{R})$. A* trace following *$(\mathcal{G}_i; \Gamma_i \vdash \Delta_i)_{i \geq 0}$ is a sequence $(A_i)_{i \geq 0}$ such that, for all $i \geq 0$, A_i is a subformula occurrence of the form $P\mathbf{t}(\mathbf{x})$ of some labelled formula $h : F$ in Γ_i, and either:*

1. *A_{i+1} is the subformula occurrence in Γ_{i+1} corresponding to A_i in Γ_i, or*

2. $(\mathcal{G}_i; \Gamma_i \vdash \Delta_i)_{i\geq 0}$ *is the conclusion of a left unfolding rule* $\mathrm{P_L}$*,* A_i *is the formula unfolded and* A_{i+1} *is an atomic predicate instance subformula of the formula obtained by the unfolding, in which case* i *is said to be a* progress point *of the trace.*

An *infinitely progressing trace* is a trace having infinitely many progress points. The previous notions lead to the definition of a cyclic proof in this labelled context, that is similar to the one given in [4,5].

Definition 9 (Cyclic proof). *A pre-proof* $(\mathcal{D}, \mathcal{R})$ *of a sequent* S *is a* (labelled) cyclic proof *if it satisfies the following* global trace condition*: for every infinite path* $(\mathcal{G}_i; \Gamma_i \vdash \Delta_i)_{i\geq 0}$ *in* $(\mathcal{D}, \mathcal{R})$*, there is an infinitely progressing trace following some tail* $(\mathcal{G}_i; \Gamma_i \vdash \Delta_i)_{i\geq n}$ *of the path.*

Figure 5 gives an example of a pre-proof in $\mathrm{G_{SL}}$ for the (LC) entailment (which we recall is valid in Reynolds' semantics for arbitrary list segments but not for acyclic list segments). The bud B and companion C of this pre-proof are indicated by the (†) marks and respectively take the following forms:

$$\begin{array}{ll} B =_{\text{def}} & h_4 h_2 \triangleright h_5; \mathcal{G}_B; h_4 : \ell\!\!s(u, x'); h_2 : \ell\!\!s(x', y); \Gamma_B \vdash h_5 : \ell\!\!s(u, y) \\ C =_{\text{def}} & h_1 h_2 \triangleright h_0; h_1 : \ell\!\!s(x, x'); h_2 : \ell\!\!s(x', y) \vdash h_0 : \ell\!\!s(x, y) \end{array}$$

Moreover, we have $C\theta\sigma \subseteq B$ with $\sigma = [\, h_4/h_1, h_5/h_0 \,]$ and $\theta = [\, x \mapsto u \,]$. A trace from C to B is indicated by the underlined formulas. Since C is the conclusion of an application of the $\ell\!\!s_{\mathrm{L}}$ rule, the trace also contains a progress point from C to B, which implies that the pre-proof is actually a cyclic proof.

5.3 Adding Size Constraints

In a labelled proof system where labels represent heaps, one can explicitly state size constraints about heap domains, and easily take advantage of the fact that the domain of a heap is finite. Such size-constraints open the way for alternate global soundness criteria that do not rely on the global trace condition of Definition 9. As an illustration, we give such a criterion in Definition 11 for the particular case of non-overlapping cyclic proofs, which is therefore weaker than the more general global trace condition, but reasonably easy to check.

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

Definition 10 (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 $\mathrm{G_{SL}}$ sequent $S = \mathcal{G}; \Gamma \vdash \Delta$ induces a set $Size(S)$ defined as the smallest set of size constraints such that:

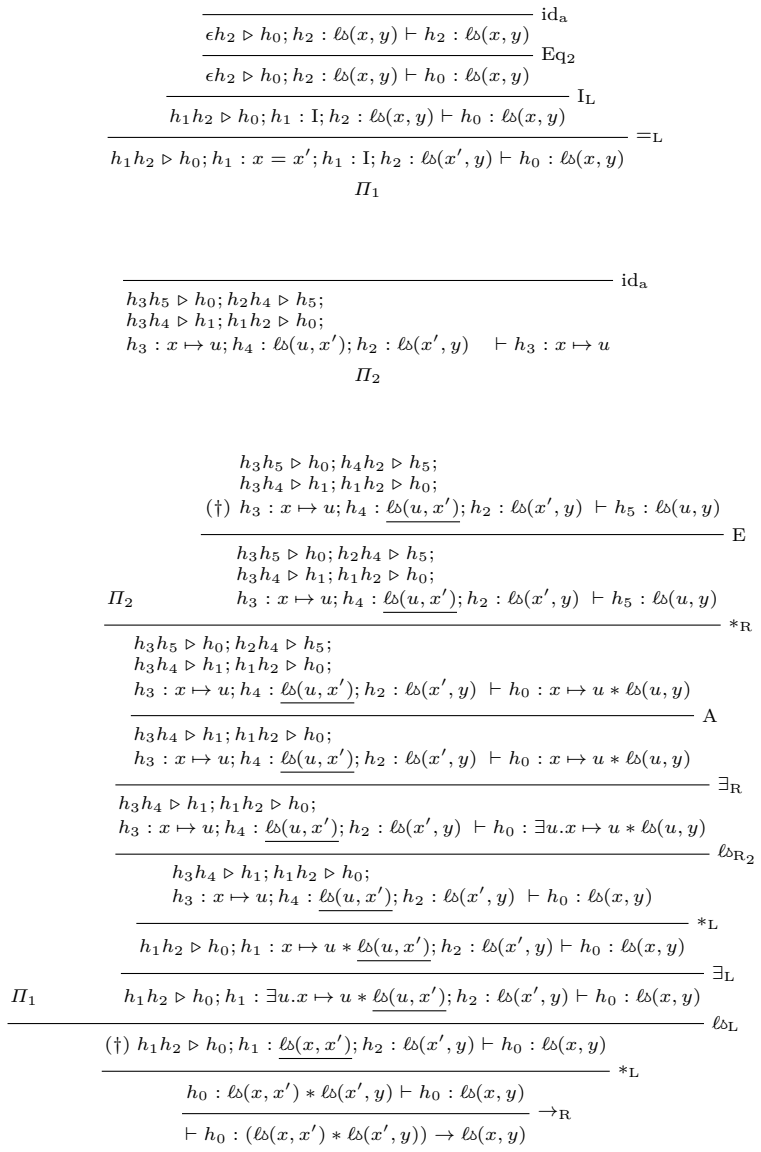

Fig. 5. Cyclic proof of $(\ell s(x,x') * \ell s(x',y)) \rightarrow \ell s(x,y)$ in $\mathrm{G_{SL}}$.

– 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 $|h| = 1 \in Size(S)$.

We now give an easy-to-check global soundness criterion for pre-proofs with no overlapping cycles.

Definition 11. *A pre-proof $(\mathcal{D}, \mathcal{R})$ of a sequent S with no overlapping cycles is a* (labelled) cyclic proof *if it satisfies the following* decreasing size condition*: for each bud B in $\mathcal{D}$ and its assigned triple $(C, \theta, \sigma) = \mathcal{R}(B)$:*

– *the companion C contains at a least one labelled formula $h : P\mathbf{t}(\mathbf{x})$, where P is an inductive predicate symbol, and*
– *the bud B is such that $Size(B) \models \{|h\sigma| < |h|\}$.*

For the (LC) entailment depicted in Figure 5, where we have $C\theta\sigma \subseteq B$ with $\sigma = [\, h_4/h_1, h_5/h_0 \,]$ and $\theta = [\, x \mapsto u \,]$, the pre-proof is a cyclic proof in the sense of Definition 11 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 $\ell_{\mathfrak{b}}$ predicate in B and C.

Theorem 1. *If there is a cyclic proof of $\vdash h_0 : F$ in* $\mathrm{G_{SL}}$*, then F is valid in* SL*.*

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

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 $\mathrm{T_{SL}}$ [15] and $\mathrm{LS_{SL}}$ [18]. The new proof rules of $\mathrm{G_{SL}}$ are easily proven sound along the lines of their intuitive justifications at the beginning of Section 5. The local soundness of the unfolding rules obtained as explained in Section 5.1 is an easy consequence of the productions being read as a disjunction $\bigvee_i C_i$ of inductive clauses.

Proving that the cyclic mechanism is sound in the sense of Definition 9 goes along the lines of the proofs given in [4,5], *i.e.*, showing that the global trace condition induces the existence of an infinitely decreasing chain of ordinals indexing the chain of approximants underlying the least fixed point interpretation of an inductive predicate, which contradicts the well-foundedness of the ordinals.

Let us prove that the decreasing size condition given in Definition 11 is sound for pre-proofs with no overlapping cycles. Suppose otherwise, then we have a cyclic proof $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ for a sequent $\vdash h_0 : F$ but F is not valid in SL. Then the root sequent S is realizable. Since local soundness implies that proof rules preserve realizability, we would be able to construct from $\mathcal{P}$ an infinite path of realizable sequents. Since initial sequents (axioms) are not realizable and since $\mathcal{P}$ is a cyclic proof with no overlapping cycles, every infinite path necessarily

contains a tail consisting of infinite repetitions of cycles involving occurrences of the same bud B and associated companion C in $\mathcal{P}$. However, since $\mathcal{P}$ satisfies the decreasing size condition of Definition 11, for at least one occurrence of an inductive predicate P, each cyclic jump from B to C strictly decreases the size of the heap realizing that occurrence of P, which contradicts the fact that the domain of a heap should be finite.

6 A Case Study: Acyclic List Extension

Let us focus in this section on 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):

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

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

$$(s, h) \models ls(x, y) \rightarrow \neg(y \mapsto z * \top)$$

which is equivalent to:

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

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

Lemma 1. (ALE) *is valid in* SL *for acyclic list segments.*

Proof. The proof is by induction on the size 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 $(s, h) \not\models \text{I}$, by definition of ls, $(s, h) \models ls(x, y)$ implies:

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

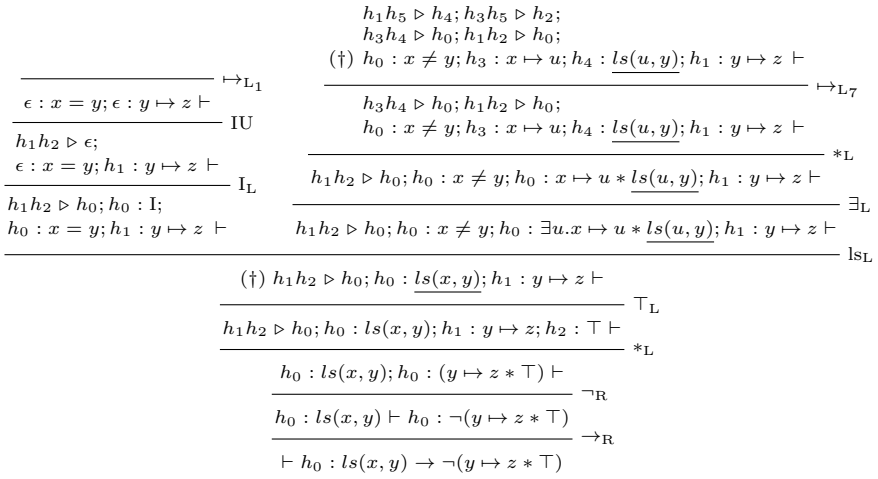

Fig. 6. Cyclic proof of $(ls(x,y) \to \neg(y \mapsto z * \top))$ in $\mathrm{G_{SL}}$.

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

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

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 in the sense of Definition 11 because it contains no overlapping cycles and in the bud B, $h_3 h_5 \rhd h_2$ and $h_3 : x \mapsto u$ imply that $|h_5| < |h_2|$ and it then follows from $h_1 h_2 \rhd h_0$ and $h_1 h_5 \rhd h_4$ that $Size(B) \models \{|h_4| < |h_0|\}$. It also satisfies the global trace condition of Definition 9.

Let us first remark that (ALE) is not provable in $\mathrm{LS_{SL}}$ as it lacks the rules IU and $\mapsto_{\mathrm{L}_7}$. Let us also remark that (ALE) cannot be proven in $\mathrm{G_{SL}}$ for arbitrary list segments as unsoundness of $\mathrm{G_{SL}}$ would otherwise immediately follow from the non-validity of (ALE) w.r.t. arbitrary list segments. Indeed, looking at the proof of Figure 6, we notice that an application of the left unfolding rule for

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

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

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

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

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

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

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

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 $G(e \mapsto e') =_{\mathrm{def}} G(e \mapsto e', e'') =_{\mathrm{def}} \bot$, $G(ls(e, e')) =_{\mathrm{def}} (e = e')$.
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 $h_1 h_3 \rhd h_4$, $h_3 : ad(e_3)$ 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 $h_1 h_3 \rhd h_4$, $h_3 : ad(e_3)$ 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'.

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

arbitrary list segments would not introduce $h_0 : x \neq y$ on the left-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 6.

7 Deriving $\mathrm{LS}_{\mathrm{SL}}$ Data Structures in G_{SL}

In this section, we show that the dedicated proof rules for acyclic list segments and binary trees that are built in $\mathrm{LS}_{\mathrm{SL}}$ are derivable in G_{SL}. Since the core of G_{SL} includes the core of $\mathrm{LS}_{\mathrm{SL}}$, it immediately follows that G_{SL} is strictly more powerful than $\mathrm{LS}_{\mathrm{SL}}$ in terms of what it can prove.

7.1 Handling Acyclic Lists

The rules for acyclic list segments in $\mathrm{LS}_{\mathrm{SL}}$ are given in Figure 7.

Theorem 2. *The rules for acyclic list segments in* $\mathrm{LS_{SL}}$ *are derivable in* $\mathrm{G_{SL}}$.

Proof. We proceed by case analysis for each rule of Figure 7. We only give a few illustrative cases, the others being similar.

1. Case of $\mathrm{LS_1}$:

$$\dfrac{\dfrac{\dfrac{\mathcal{G};\Gamma[\,e_2 \mapsto e_1\,] \vdash \Delta[\,e_2 \mapsto e_1\,]}{\mathcal{G};\Gamma;\epsilon : e_1 = e_2 \vdash \Delta}\,{=_{\mathrm{L}}}}{\mathcal{G};\Gamma;\epsilon : e_1 = e_2;\epsilon : \mathrm{I} \vdash \Delta}\,{\mathrm{I_L}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{G};\Gamma;\epsilon : e_1 \neq e_2;\epsilon : e_1 \mapsto u;\epsilon : ls(u,e_2) \vdash \Delta}\,{\mapsto_{\mathrm{L_1}}}}{h_1h_2 \rhd \epsilon;\ \mathcal{G};\Gamma;\epsilon : e_1 \neq e_2;h_1 : e_1 \mapsto u;h_2 : ls(u,e_2) \vdash \Delta}\,{\mathrm{IU}}}{\mathcal{G};\Gamma;\epsilon : e_1 \neq e_2;\epsilon : e_1 \mapsto u * ls(u,e_2) \vdash \Delta}\,{*_{\mathrm{L}}}}{\mathcal{G};\Gamma;\epsilon : e_1 \neq e_2;\epsilon : \exists u.e_1 \mapsto u * ls(u,e_2) \vdash \Delta}\,{\exists_{\mathrm{L}}}}{\mathcal{G};\Gamma;\epsilon : ls(e_1,e_2) \vdash \Delta}\,{\mathrm{ls_L}}$$

2. Case of $\mathrm{LS_2}$:

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

3. Case of $\mathrm{LS_3}$:

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

4. Case of $\mathrm{LS_4}$:

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

5. Case of $\mathrm{LS_6}$: We derive $\mathrm{LS_6}$ using the $\mathrm{cut_a}$ rule as follows:

$$\dfrac{\begin{array}{ll} \multicolumn{2}{c}{\Pi_0} \\ \multicolumn{2}{c}{\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots} \\ h_1h_2 \rhd h_0;\mathcal{G};\Gamma; & h_1h_2 \rhd h_0;\mathcal{G};\Gamma; \\ h_1 : ds(e_1,e_2); & h_1 : ds(e_1,e_2); \\ h_0 : ls(e_1,e_3) \quad \vdash\ h_2 : ls(e_2,e_3);\Delta & h_0 : ls(e_1,e_3);h_2 : ls(e_2,e_3) \ \vdash\ \Delta \end{array}}{h_1h_2 \rhd h_0;\mathcal{G};\Gamma;h_1 : ds(e_1,e_2);h_0 : ls(e_1,e_3) \vdash \Delta}\,{\mathrm{cut_a}}$$

We then proceed with the proof of the left-premiss of the cut rule. For conciseness, we omit $\mathcal{G}$ and Γ as they play no significant role in the proof.

If $ds(e_1, e_2)$ is $e_1 \mapsto e_2$:
We construct the proof given below in which we may assume that e_1 is a variable (otherwise LS_6 should be replaced with the NIL rule to finish the proof immediately). Therefore, in the $=_{\mathrm{L}}$ step, $mgu(\{\, e_1, e_3 \,\}) = [\, e_1 \mapsto e_3 \,]$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon : e_3 \mapsto e_2 \;\vdash\; \epsilon : ls(e_2, e_3)}\,{\scriptstyle \mapsto_{\mathrm{L}_1}}}{h_1 h_2 \triangleright \epsilon;\; h_1 : e_3 \mapsto e_2 \;\vdash\; h_2 : ls(e_2, e_3)}\,\mathrm{IU}}{h_1 h_2 \triangleright \epsilon;\; h_1 : e_1 \mapsto e_2;\; \epsilon : e_1 = e_3 \;\vdash\; h_2 : ls(e_2, e_3)}\,{\scriptstyle =_{\mathrm{L}}}}{h_1 h_2 \triangleright h_0;\; h_1 : e_1 \mapsto e_2;\; h_0 : e_1 = e_3;\; h_0 : \mathrm{I} \;\vdash\; h_2 : ls(e_2, e_3)}\,\mathrm{I}_{\mathrm{L}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{h_1 h_2 \triangleright h_0;\; h_1 : e_1 \mapsto e_2;\; h_0 : e_1 \neq e_3;\; h_2 : ls(e_2, e_3) \;\vdash\; h_2 : ls(e_2, e_3)}\,\mathrm{id}_{\mathrm{a}}}{h_1 h_2 \triangleright h_0;\; h_3 h_4 \triangleright h_0;\; h_1 : e_1 \mapsto e_2;\; h_0 : e_1 \neq e_3;\; h_3 : e_1 \mapsto u;\; h_4 : ls(u, e_3) \;\vdash\; h_2 : ls(e_2, e_3)}\,{\scriptstyle \mapsto_{\mathrm{L}_6}}}{h_1 h_2 \triangleright h_0;\; h_1 : e_1 \mapsto e_2;\; h_0 : e_1 \neq e_3;\; h_0 : e_1 \mapsto u * ls(u, e_3) \;\vdash\; h_2 : ls(e_2, e_3)}\,{\scriptstyle *_{\mathrm{L}}}}{h_1 h_2 \triangleright h_0;\; h_1 : e_1 \mapsto e_2;\; h_0 : e_1 \neq e_3;\; h_0 : \exists u. e_1 \mapsto u * ls(u, e_3) \;\vdash\; h_2 : ls(e_2, e_3)}\,{\scriptstyle \exists_{\mathrm{L}}}$$

$$\dfrac{}{h_1 h_2 \triangleright h_0;\; h_1 : e_1 \mapsto e_2;\; h_0 : ls(e_1, e_3) \;\vdash\; h_2 : ls(e_2, e_3)}\,{\scriptstyle \mathrm{ls}_{\mathrm{L}}}$$

If $ds(e_1, e_2)$ is $ls(e_1, e_2)$:

$$(\dagger)\quad \begin{array}{l} h_6h_2 \triangleright h_4; h_2h_6 \triangleright h_4; \\ h_1h_2 \triangleright h_0; h_3h_4 \triangleright h_0; h_3h_6 \triangleright h_1; \\ h_1 : e_1 \neq e_2; h_6 : ls(u, e_2); \\ h_0 : e_1 \neq e_3; h_3 : e_1 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3) \end{array}$$

$\overline{\qquad\qquad\qquad\qquad}$ E

$$\begin{array}{l} h_2h_6 \triangleright h_4; h_1h_2 \triangleright h_0; h_3h_4 \triangleright h_0; h_3h_6 \triangleright h_1; \\ h_1 : e_1 \neq e_2; h_6 : ls(u, e_2); \\ h_0 : e_1 \neq e_3; h_3 : e_1 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3) \end{array}$$

$\overline{\qquad\qquad\qquad\qquad}$ C

$$\begin{array}{l} h_3h_7 \triangleright h_0; h_2h_6 \triangleright h_7; \\ h_1h_2 \triangleright h_0; h_3h_4 \triangleright h_0; h_3h_6 \triangleright h_1; \\ h_1 : e_1 \neq e_2; h_6 : ls(u, e_2); \\ h_0 : e_1 \neq e_3; h_3 : e_1 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3) \end{array}$$

$\overline{\qquad\qquad\qquad\qquad}$ $\mapsto_{\mathrm{L6}}$

$$\begin{array}{l} h_5h_7 \triangleright h_0; h_2h_6 \triangleright h_7; \\ h_1h_2 \triangleright h_0; h_3h_4 \triangleright h_0; h_5h_6 \triangleright h_1; \\ h_1 : e_1 \neq e_2; h_5 : e_1 \mapsto v; h_6 : ls(v, e_2); \\ h_0 : e_1 \neq e_3; h_3 : e_1 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3) \end{array}$$

$\overline{\qquad\qquad\qquad\qquad}$ A

$$\begin{array}{l} h_1h_2 \triangleright h_0; h_3h_4 \triangleright h_0; h_5h_6 \triangleright h_1; \\ h_1 : e_1 \neq e_2; h_5 : e_1 \mapsto v; h_6 : ls(v, e_2); \\ h_0 : e_1 \neq e_3; h_3 : e_1 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3) \end{array}$$

$\overline{\qquad\qquad\qquad\qquad}$ $*_{\mathrm{L}}$

$$\begin{array}{l} h_1h_2 \triangleright h_0; h_3h_4 \triangleright h_0; \\ h_1 : e_1 \neq e_2; h_1 : e_1 \mapsto v * ls(v, e_2); \\ h_0 : e_1 \neq e_3; h_3 : e_1 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3) \end{array}$$

$\overline{\qquad\qquad\qquad\qquad}$ $\exists_{\mathrm{L}}$

$$\begin{array}{l} h_1h_2 \triangleright h_0; h_3h_4 \triangleright h_0; \\ h_1 : e_1 \neq e_2; h_1 : \exists v.e_1 \mapsto v * ls(v, e_2); \\ h_0 : e_1 \neq e_3; h_3 : e_1 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3) \end{array}$$

$\overline{\qquad\qquad\qquad\qquad}$ $\wedge_{\mathrm{L}}$

$$\varPi_2 \qquad \begin{array}{l} h_1h_2 \triangleright h_0; h_3h_4 \triangleright h_0; \\ h_1 : e_1 \neq e_2 \wedge (\exists v.e_1 \mapsto v * ls(v, e_2)); \\ h_0 : e_1 \neq e_3; h_3 : e_1 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3) \end{array}$$

$\overline{\qquad\qquad\qquad\qquad}$ ls_{L}

$$\begin{array}{l} h_1h_2 \triangleright h_0; h_3h_4 \triangleright h_0; \\ h_1 : ls(e_1, e_2); \\ h_0 : e_1 \neq e_3; h_3 : e_1 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3) \end{array}$$

$\overline{\qquad\qquad\qquad\qquad}$ $*_{\mathrm{L}}$

$$\begin{array}{l} h_1h_2 \triangleright h_0; \\ h_1 : ls(e_1, e_2); \\ h_0 : e_1 \neq e_3; h_0 : e_1 \mapsto u * ls(u, e_3) \vdash h_2 : ls(e_2, e_3) \end{array}$$

$\overline{\qquad\qquad\qquad\qquad}$ $\exists_{\mathrm{L}}$

$$\varPi_1 \qquad \begin{array}{l} h_1h_2 \triangleright h_0; \\ h_1 : ls(e_1, e_2); \\ h_0 : e_1 \neq e_3; h_0 : \exists u.e_1 \mapsto u * ls(u, e_3) \vdash h_2 : ls(e_2, e_3) \end{array}$$

$\overline{\qquad\qquad\qquad\qquad}$ ls_{L}

$$(\dagger)\quad h_1h_2 \triangleright h_0; h_1 : ls(e_1, e_2); h_0 : ls(e_1, e_3) \vdash h_2 : ls(e_2, e_3)$$

Before giving the $\varPi_1$ and $\varPi_2$ subproofs, let us focus on the right part of the derivation which makes it a pre-proof having a single cycle with a bud B and a companion C (indicated by the (†) marks) such that $C\theta\sigma \subseteq B$ where $\sigma = [\,h_6/h_1, h_4/h_0\,]$ and $\theta = [\,e_1 \mapsto u\,]$. We get $|h_3| = 1$, $|h_0| = 1 + |h_4|$ and $|h_1| = 1 + |h_6|$ from the occurrence of $h_3 : e_1 \mapsto u$, $h_3h_4 \triangleright h_0$ and $h_3h_6 \triangleright h_1$ in the bud B. We also have both $|h_4| < |h_0|$ and $|h_6| < |h_1|$, which is enough to conclude that the pre-proof is a cyclic proof in the sense of Definition 11 (w.r.t. either $h_1 : ls(e_1, e_2)$ or $h_0 : ls(e_1, e_3)$ in C). It is easy to see that it is also a cyclic proof in the sense of Definition 9 following either the $ls(e_1, e_2)$ or the $ls(e_1, e_3)$ predicate.

We now complete the subproof $\varPi_1$:

$$
\dfrac{\dfrac{\dfrac{}{\epsilon : \mathrm{I} \vdash \epsilon : e_3 = e_3}\,{=_\mathrm{R}} \quad \dfrac{}{\epsilon : \mathrm{I} \vdash \epsilon : \mathrm{I}}\,{\mathrm{I}_\mathrm{R}}}{\epsilon : \mathrm{I} \vdash \epsilon : e_3 = e_3 \wedge \mathrm{I}}\,{\wedge_\mathrm{R}}}{\epsilon : \mathrm{I} \vdash \epsilon : ls(e_3, e_3)}\,{\mathrm{ls}_{\mathrm{R}_2}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{l}\epsilon : e_3 \neq e_2;\\ \epsilon : e_3 \mapsto u; \epsilon : ls(u, e_2) \vdash \epsilon : ls(e_2, e_3)\end{array}}\,{\mapsto_{\mathrm{L}_1}}}{\begin{array}{l}h_3 h_4 \rhd \epsilon; \epsilon : e_3 \neq e_2;\\ h_3 : e_3 \mapsto u; h_4 : ls(u, e_2) \vdash \epsilon : ls(e_2, e_3)\end{array}}\,{\mathrm{IU}}}{\begin{array}{l}\epsilon : e_3 \neq e_2;\\ \epsilon : e_3 \mapsto u * ls(u, e_2) \vdash \epsilon : ls(e_2, e_3)\end{array}}\,{*_\mathrm{L}}}{\begin{array}{l}\epsilon : e_3 \neq e_2;\\ \epsilon : \exists u.e_3 \mapsto u * ls(u, e_2) \vdash \epsilon : ls(e_2, e_3)\end{array}}\,{\exists_\mathrm{L}}}{\epsilon : e_3 = e_2; \epsilon : \mathrm{I} \vdash \epsilon : ls(e_2, e_3)}\,{=_\mathrm{L}}}{\epsilon : ls(e_3, e_2) \vdash \epsilon : ls(e_2, e_3)}\,{\mathrm{ls}_\mathrm{L}}
$$

$$
\dfrac{\dfrac{\dfrac{\epsilon : ls(e_3, e_2) \vdash \epsilon : ls(e_2, e_3)}{h_1 h_2 \rhd \epsilon; h_1 : ls(e_3, e_2) \vdash h_2 : ls(e_2, e_3)}\,{\mathrm{IU}}}{h_1 h_2 \rhd \epsilon; h_1 : ls(e_1, e_2); \epsilon : e_1 = e_3 \vdash h_2 : ls(e_2, e_3)}\,{=_\mathrm{L}}}{h_1 h_2 \rhd h_0; h_1 : ls(e_1, e_2); h_0 : e_1 = e_3; h_0 : \mathrm{I} \vdash h_2 : ls(e_2, e_3)}\,{\mathrm{I}_\mathrm{L}}
$$

Π_1

Finally, we complete the subproof Π_2:

$$
\dfrac{\dfrac{}{\begin{array}{l}\epsilon h_2 \rhd h_0;\\ h_3 h_4 \rhd h_2;\\ h_2 : e_2 \neq e_3;\\ h_3 : e_2 \mapsto u;\\ h_4 : ls(u, e_3) \vdash h_2 : e_2 \neq e_3\end{array}}\,{\mathrm{id}_\mathrm{a}} \qquad \dfrac{\dfrac{\dfrac{}{\begin{array}{l}\epsilon h_2 \rhd h_0;\\ h_3 h_4 \rhd h_2;\\ h_2 : e_2 \neq e_3;\\ h_3 : e_2 \mapsto u;\\ h_4 : ls(u, e_3)\end{array} \vdash h_2 : e_2 \mapsto u * ls(u, e_3)}\,{\exists_\mathrm{L}}}{\begin{array}{l}\epsilon h_2 \rhd h_0; h_3 h_4 \rhd h_2;\\ h_2 : e_2 \neq e_3;\\ h_3 : e_2 \mapsto u;\\ h_4 : ls(u, e_3)\end{array} \vdash h_2 : \exists v.e_2 \mapsto v * ls(v, e_3)}\,{\exists_\mathrm{L}}}{\begin{array}{l}\epsilon h_2 \rhd h_0; h_3 h_4 \rhd h_2;\\ h_2 : e_2 \neq e_3;\\ h_3 : e_2 \mapsto u;\\ h_4 : ls(u, e_3) \vdash h_2 : e_2 \neq e_3 \wedge (\exists v.e_2 \mapsto v * ls(v, e_3))\end{array}}\,{\wedge_\mathrm{R}}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{l}\epsilon h_2 \rhd h_0; h_3 h_4 \rhd h_2;\\ h_2 : e_2 \neq e_3; h_3 : e_2 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3)\end{array}}\,{\mathrm{Eq}_2}}{\begin{array}{l}\epsilon h_2 \rhd h_0; h_3 h_4 \rhd h_0;\\ h_0 : e_2 \neq e_3; h_3 : e_2 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3)\end{array}}\,{\mathrm{I}_\mathrm{L}}}{\begin{array}{l}h_1 h_2 \rhd h_0; h_3 h_4 \rhd h_0;\\ h_1 : \mathrm{I};\\ h_0 : e_2 \neq e_3; h_3 : e_2 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3)\end{array}}\,{=_\mathrm{L}}}{\begin{array}{l}h_1 h_2 \rhd h_0; h_3 h_4 \rhd h_0;\\ h_1 : e_1 = e_2, h_1 : \mathrm{I};\\ h_0 : e_1 \neq e_3; h_3 : e_1 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3)\end{array}}\,{\wedge_\mathrm{L}}}{\begin{array}{l}h_1 h_2 \rhd h_0; h_3 h_4 \rhd h_0;\\ h_1 : e_1 = e_2 \wedge \mathrm{I};\\ h_0 : e_1 \neq e_3; h_3 : e_1 \mapsto u; h_4 : ls(u, e_3) \vdash h_2 : ls(e_2, e_3)\end{array}}
$$

Π_2

$$\frac{\mathcal{G}; \Gamma[e \mapsto nil] \vdash \Delta[e \mapsto nil]}{\mathcal{G}; \Gamma; \epsilon : tr(e) \vdash \Delta}\ \mathrm{TR}_1 \qquad \frac{\mathcal{G}; \Gamma[\, e_2 \mapsto e_1\,]; h : tr(e_1) \vdash \Delta[\, e_2 \mapsto e_1\,]}{\mathcal{G}; \Gamma; h : tr(e_1); h : tr(e_2) \vdash \Delta}\ \mathrm{TR}_4$$

$$\frac{}{\mathcal{G}; \Gamma \vdash \epsilon : tr(nil); \Delta}\ \mathrm{TR}_2 \qquad \frac{\mathcal{G}; \Gamma; \epsilon : \mathrm{I} \vdash \Delta}{\mathcal{G}; \Gamma; \epsilon : tr(nil) \vdash \Delta}\ \mathrm{TR}_3$$

$$\frac{h_1 h_2 \triangleright h_0; \mathcal{G}; \Gamma; h_1 : e \mapsto e_1, e_2; h_0 : tr(e); h_2 : tr(e_1) * tr(e_2) \vdash \Delta}{h_1 h_2 \triangleright h_0; \mathcal{G}; \Gamma; h_1 : e \mapsto e_1, e_2; h_0 : tr(e) \vdash \Delta}\ \mathrm{TR}_5$$

$$\frac{h_1 h_2 \triangleright h_0; \mathcal{G}; \Gamma; h_1 : e \mapsto e_1, e_2 \vdash h_2 : tr(e_1) * tr(e_2); h_0 : tr(e); \Delta}{h_1 h_2 \triangleright h_0; \mathcal{G}; \Gamma; h_1 : e \mapsto e_1, e_2 \vdash h_0 : tr(e); \Delta}\ \mathrm{TR}_6$$

Fig. 8. Rules for binary trees in $\mathrm{LS}_{\mathrm{SL}}$.

$$\frac{\mathcal{G}; \Gamma \vdash h : e = nil \wedge \mathrm{I}; \Delta}{\mathcal{G}; \Gamma \vdash h : tr(e); \Delta}\ \mathrm{tr}_{\mathrm{R}_1} \qquad \frac{\mathcal{G}; \Gamma \vdash \exists uv.e \mapsto u, v * tr(u) * tr(v); \Delta}{\mathcal{G}; \Gamma \vdash h : tr(e); \Delta}\ \mathrm{tr}_{\mathrm{R}_2}$$

$$\frac{\mathcal{G}; \Gamma; h : e = nil; h : \mathrm{I} \vdash \Delta \qquad \mathcal{G}; \Gamma; h : \exists uv.e \mapsto u, v * tr(u) * tr(v) \vdash \Delta}{\mathcal{G}; \Gamma; h : tr(e) \vdash \Delta}\ \mathrm{tr}_{\mathrm{L}}$$

Fig. 9. Rules for binary trees in G_{SL}.

7.2 Handling Binary Trees

In G_{SL} we define the binary tree predicate tr by the following productions:

$$x = nil \wedge \mathrm{I} \overset{x}{\Rightarrow} tr(x) \qquad\qquad x \mapsto y, z * tr(y) * tr(z) \overset{x,y,z}{\Rightarrow} tr(x)$$

which gives the proof rules depicted in Figure 9. The rules for binary trees in $\mathrm{LS}_{\mathrm{SL}}$ are given in Figure 8.

Defining binary trees requires the use of the two-field points-to predicate. As explained in [18], the pointer-rules of G_{SL} can directly be generalized to cover the case of points-to predicates with an arbitrary number of fields by reading the right-hand side e' of each occurrence of $e \mapsto e'$ in Figure 3 as a sequence of values $e_1, \ldots, e_n$ instead of a single value e'.

In Reynolds' semantics, a multi-field points-to predicate $e \mapsto e_1, \ldots, e_n$ is defined as a shorthand for $(e \mapsto e_1) * (e + 1 \mapsto e_2) \ldots * (e + (n-1) \mapsto e_n)$. However, the semantics we presented in Section 2 is the non-Reynolds' semantics in which $e \mapsto e_1, \ldots, e_n$ is interpreted (w.r.t. a store s) as a singleton heap that maps the value $[\![e]\!]_s$ of e to a unique cell which contains the tuple $\langle\, [\![e_1]\!]_s, \ldots, [\![e_n]\!]_s \,\rangle$. As discussed in [18], we thus need the following additional rule (where e_1 and e_2 should be read as sequences of values):

$$\frac{}{\mathcal{G}; \Gamma; h : e \mapsto e_1, e_2 \vdash h : e \mapsto e_1; \Delta}\ \mathrm{NR}$$

Theorem 3. *The rules for binary trees in* $\mathrm{LS_{SL}}$ *are derivable in* $\mathrm{G_{SL}}$.

Proof. We only give the most interesting cases and leave the others to the reader.

1. Case of TR_5:
   Let $\Delta = h_2 : tr(e_1) * tr(e_2)$. We first construct the following proof:

$$\dfrac{\dfrac{}{h_1h_2 \rhd h_0;\ h_1 : nil \mapsto e_1, e_2;\ h_0 : \mathrm{I} \vdash \Delta}\ \mathrm{NIL}}{h_1h_2 \rhd h_0;\ h_1 : e \mapsto e_1, e_2;\ h_0 : e = nil;\ h_0 : \mathrm{I} \vdash \Delta}\ =_{\mathrm{L}} \qquad \dfrac{\dfrac{\dfrac{}{h_1h_2 \rhd h_0;\ h_1 : e \mapsto e_1, e_2;\ h_2 : tr(e_1) * tr(e_2) \vdash \Delta}\ \mathrm{id_a}}{h_1h_2 \rhd h_0;\ h_3h_4 \rhd h_0;\ h_1 : e \mapsto e_1, e_2;\ h_3 : e \mapsto u, v; h_4 : tr(u) * tr(v) \vdash \Delta}\ \mapsto_{\mathrm{L}_6}}{h_1h_2 \rhd h_0;\ h_1 : e \mapsto e_1, e_2;\ h_0 : e \mapsto u, v * tr(u) * tr(v) \vdash \Delta}\ *_{\mathrm{L}}}{h_1h_2 \rhd h_0;\ h_1 : e \mapsto e_1, e_2;\ h_0 : \exists uv.e \mapsto u, v * tr(u) * tr(v) \vdash \Delta}\ \exists_{\mathrm{L}}$$

$$\dfrac{\qquad\qquad}{h_1h_2 \rhd h_0; h_1 : e \mapsto e_1, e_2; h_0 : tr(e) \vdash \Delta}\ \mathrm{tr_L}$$

   The rule TR_5 then easily follows by an application of $\mathrm{cut_a}$.
2. Case of TR_6:
   We first show that right-contraction ($\mathrm{C_R}$) is derivable in $\mathrm{G_{SL}}$:

$$\dfrac{\dfrac{\mathcal{G}; \Gamma \vdash h : A; h : A; \Delta}{\mathcal{G}; \Gamma \vdash h : A \vee A; \Delta}\ \vee_{\mathrm{R}} \qquad \dfrac{\dfrac{}{\mathcal{G}; \Gamma; h : A \vdash h : A; \Delta}\ \mathrm{id_a} \qquad \dfrac{}{\mathcal{G}; \Gamma; h : A \vdash h : A; \Delta}\ \mathrm{id_a}}{\mathcal{G}; \Gamma; h : A \vee A \vdash h : A; \Delta}\ \vee_{\mathrm{L}}}{\mathcal{G}; \Gamma \vdash h : A; \Delta}\ \mathrm{cut_a}$$

   Then we proceed with the actual proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{h_1h_2 \rhd h_0; \mathcal{G}; \Gamma;\ h_1 : e \mapsto e_1, e_2 \vdash h_1 : e \mapsto e_1, e_2, \ldots}\ \mathrm{id_a} \qquad h_1h_2 \rhd h_0; \mathcal{G}; \Gamma;\ h_1 : e \mapsto e_1, e_2 \vdash h_2 : tr(e_1) * tr(e_2); \ldots}{h_1h_2 \rhd h_0; \mathcal{G}; \Gamma; h_1 : e \mapsto e_1, e_2 \vdash h_0 : e \mapsto e_1, e_2 * tr(e_1) * tr(e_2); h_0 : tr(e); \Delta}\ *_{\mathrm{R}}}{h_1h_2 \rhd h_0; \mathcal{G}; \Gamma; h_1 : e \mapsto e_1, e_2 \vdash h_0 : \exists uv.e \mapsto u, v * tr(u) * tr(v); h_0 : tr(e); \Delta}\ \exists_{\mathrm{R}}}{h_1h_2 \rhd h_0; \mathcal{G}; \Gamma; h_1 : e \mapsto e_1, e_2 \vdash h_0 : tr(e); h_0 : tr(e); \Delta}\ \mathrm{tr_{R_2}}}{h_1h_2 \rhd h_0; \mathcal{G}; \Gamma; h_1 : e \mapsto e_1, e_2 \vdash h_0 : tr(e); \Delta}\ \mathrm{C_R}$$

Corollary 1. *If* A *is provable in* $\mathrm{LS_{SL}}$ *extended with the built-in rules for data structures (acyclic lists and binary trees) then* A *is also provable in* $\mathrm{G_{SL}}$ *(with the appropriate predicates).*

8 Conclusion and Perspectives

In this paper we have proposed the first labelled cyclic proof system for SL that supports both the full set of SL connectives and arbitrarily defined inductive predicates. Our proof system, called G_{SL}, successfully combines (an extension of) the labelled proof system LS_{SL} [18] without its built-in rules for data structures with the approach described in [4,5] for defining inductive predicates and translating them into unfolding rules in a cyclic proof system based on the principle of infinite descent.

We have also proved that the dedicated proof rules for acyclic list segments and binary trees built in LS_{SL} are derivable in G_{SL} and consequently that G_{SL} is strictly more powerful than LS_{SL} in terms of what it can prove.

Future work could be developed in various directions. First we could study the cut-elimination property in G_{SL} in order to complete the proof-theoretical analysis of this proof system. A recent work shows that cut-elimination fails in cyclic proofs systems for SL [20] and suggests to study restricted forms of cuts in order to characterize some proof search strategies. For instance, the approach in [10] could be seen as a cyclic proof system with cuts restricted to only those against buds. We could investigate similar restricted cuts for G_{SL} and also study whether the cut-elimination property could be recovered by restricting the use of inductive definitions.

It could also be interesting to implement the G_{SL} system and make comparisons with existing theorem provers for SL. Moreover it could be fruitful to develop new examples of entailments involving multiplicative implication inside predicates and then test how G_{SL} behaves on such examples.

Ackowledgements. We would like to thank the anonymous referees for their helpful comments and recommendations that allowed us to improve the quality of the document. This work was partially supported by the TICAMORE project (ANR grant 16-CE91-0002).

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 neighbours. In *IEEE Symposium on Logic in Computer Science, LICS'2010*, pages 130–139, Edinburgh, Scotland, 2010.
7. J. Brotherston and J. Villard. Sub-classical Boolean bunched logics and the meaning of par. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015*, 325–342, Berlin, Germany, 2015.
8. 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.
9. 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*, LNCS 3441, pages 395–409, Edinburgh, Scotland, 2005.
10. D.-H. Chu, J. Jaffer and M.-T.Trinh. Automatic Induction Proofs of Data-Structres in Imperative Programs. In *36th ACM SIGPLAN Conference on Programming Design and Implementation, PLDI'15*, pages 457–466, ACM Press, 2015.
11. 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*, LNCS 8476, pages 125–138, Moscow, Russia, 2014.
12. S. Demri and M. Deters. Two-Variable Separation Logic and Its Inner Circle. *ACM Transaction on Computational Logic*, 16(2):15, 2015.
13. D. Galmiche and D. Méry. Semantic labelled tableaux for propositional BI without bottom. *Journal of Logic and Computation*, 13(5):707–753, 2003.
14. D. Galmiche, D. Méry, and D. Pym. The semantics of BI and Resource Tableaux. *Math. Struct. Comp. Sci.* 15(6):1033–1088, 2005.
15. D. Galmiche and D. Méry. Tableaux and resource graphs for separation logic. *Journal of Logic and Computation*, 20(1):189–231, 2010.
16. D. Galmiche, M. Marti, and D. Méry. Relating Labelled and Label-free Bunched Calculi in BI Logic. In *Int. Conference on Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX'2019*, LNAI 11714, pages 130–146, London, UK, 2019.

17. Z. Hou, A. Tiu, and R. Gore. A labelled sequent calculus for BBI: Proof theory and proof search. In *Int. Conference on Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX'2013*, LNAI 8123, pages 172–187, Nancy, France, 2013.
18. 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.
19. 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.
20. D. Kimura, K. Nakazawa, T. Terauchi and H. Hunno. Failure of Cut-elimination in Cyclic Proofs of Separation Logic. *Computer Software*, 37(1):139-152, 2020.
21. 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.
22. 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.
23. P. Martin-Löf. Haupstatz for the intuitionistic theory of iterated inductive definitions. In *J.E. Fenstad, editor, Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216, North-Holland, 1971.
24. P.W. O'Hearn and D. Pym. The Logic of Bunched Implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.
25. 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.
26. D. Pym. The Semantics and Proof Theory of the Logic of Bunched Implications. *Applied Logic Series* Vol. 26, Kluwer Academic Publishers, 2002.
27. 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.
28. M. Tatsuta, K. Nakazawa, and D. Kimura. Completeness of Cyclic Proofs for Symbolic Heaps with Inductive Definitions. In *17th Asian Symposium, APLAS'2019*, LNCS 11893, pages 367–387, Bali, Indonesia, 2019.