Automated theorem proving for assertions in separation logic with all connectives

Zhé Hóu[1], Rajeev Goré[1], and Alwen Tiu[2]

[1] Research School of Computer Science, The Australian National University
[2] School of Computer Engineering, Nanyang Technological University

Abstract. This paper considers Reynolds's separation logic with all logical connectives but without arbitrary predicates. This logic is not recursively enumerable but is very useful in practice. We give a sound labelled sequent calculus for this logic. Using numerous examples, we illustrate the subtle deficiencies of several existing proof calculi for separation logic, and show that our rules repair these deficiencies. We extend the calculus with rules for linked lists and binary trees, giving a sound, complete and terminating proof system for a popular fragment called symbolic heaps. Our prover has comparable performance to Smallfoot, a prover dedicated to symbolic heaps, on valid formulae extracted from program verification examples; but our prover is not competitive on invalid formulae. We also show the ability of our prover beyond symbolic heaps, our prover handles the largest fragment of logical connectives in separation logic.

1 Introduction

Separation logic (SL) was invented to verify the correctness of programs that mutate possibly shared data structures [30]. SL is an extension of Hoare logic with logical connectives $\top^*, *, -\!\!*$ from the logic of bunched implications (BI) [29] to capture the empty heap, heap composition, and heap extension respectively, and a predicate $\mapsto$ to describe singleton heaps. Reynolds [34] coupled the semantics for the above extensions with classical connectives, making Boolean BI (BBI) the basis of separation logic, although there are earlier versions that consider intuitionistic additive connectives [33]. Using BI logics to enable local reasoning has proven very successful, and many variants of separation logic have been developed. For instance, separation logic for higher-order store [32], bunched typing [3], concurrency [6], owned variables [4, 31], rely/guarantee reasoning [37], abstract data types [22], amongst many.

These separation logics require proof methods to reason about their assertion languages, and since most separation logic variants are based on the original SL, automated tools usually respect Reynolds's semantics [34]. However, most existing tools for Reynolds-like semantics SL, such as Smallfoot [1], jStar [13], VeriStar [35], SLP [27], and Asterix [28], are all restricted to small fragments, most notably, the symbolic heaps fragment of Berdine et al. [2]. On the other hand, there are also existing tools that handle larger fragments than symbolic heaps, but for non-Reynolds semantics, e.g., Lee and Park's theorem prover [24], and Thakur et al.'s unsatisfiability checker [36], cf. Section 4 and 6.

There is a growing demand from the program verification community to move beyond symbolic heaps and to deal with $-\!\!*$, which is ignored in most SL fragments. Having $-\!\!*$ is a desirable feature, since many algorithms/programs are verified using this connective, especially when expressing tail-recursive operations [26], iterators [23], septraction in rely/guarantee [37] etc.. Moreover, $-\!\!*$ is useful in the weakest precondition calculus for SL, which introduces $-\!\!*$ "in each statement in the program being analysed" [25]. See the introduction of [24] and [36] for other examples requiring $-\!\!*$. In addition to $-\!\!*$, allowing arbitrary combinations of logical connectives is also useful when describing overlaid data structures [16], properties such as cross-split can be useful in proof search in this setting [14]. Nevertheless, existing tools for SL with Reynolds's semantics do not support the reasoning for all logical connectives. Thus, an important area of research is to obtain a practical proof system for SL with all connectives.

SL is not recursively enumerable in general [10, 5], neither is the fragment we consider here, so there is no finite, sound and complete proof system for this logic, and computability results are not our focus. Interested readers are referred to [12, 11] for other fragments of SL and their decidability and complexity results. Building upon the labelled sequent calculi for propositional abstract separation logics (PASL, cf. [19]), we give a sound, w.r.t. Reynolds's semantics, proof method that is useful in program verification. Since we focus on SL with heap model semantics here, although [19] is complete for PASL, it is not comparable to this work in terms of provability. We extend PASL with inference rules for quantifiers, equality and the $\mapsto$ predicate. The latter involves heaps and stores in the semantics in a very subtle way, making this study error-prone. Some subtle mistakes in the literature are discussed in Section 4.

Capturing data structures is important since they are frequently used in program verification. We extend our proof system with treatments for singly linked lists based on similar rules in Smallfoot [2]. Binary trees can be handled similarly; see [18]. We also move beyond symbolic heaps to consider arbitrary combinations of logical connectives. We show that our proof method is complete w.r.t. symbolic heaps. We give a sound, complete, and terminating proof search procedure for symbolic heaps. Our implementation is competitive with Smallfoot on valid formulae, but not on invalid formulae, taken from benchmarks extracted from program verification problems. In addition, we demonstrate that our prover can handle a wider range of formulae than existing tools, thus it handles the largest fragment of SL in terms of logical connectives, and paves the way to more sophisticated program verification using Reynolds's SL.

2 Separation Logic

Separation logic generally refers to a combination of a programming language, an assertion logic and a specification logic for Hoare triples [21]. Here, we focus on the assertion logic that is compliant with Reynolds's semantics [34].

Following Reynolds [34], we consider all *values* as *integers*, an infinite number of which are *addresses. Atoms*, containing *nil*, form a subset of values that is disjoint from addresses. *Heaps* are finite partial functions from addresses to values, and *stores* are total functions from finite sets of *program variables* to values. These are formalised as below:

$$\begin{array}{lll} \mathit{Val} = \mathit{Int} & \mathit{Atoms} \cup \mathit{Addr} \subseteq \mathit{Int} & \mathit{nil} \in \mathit{Atoms} \\ \mathit{Atoms} \cap \mathit{Addr} = \emptyset & H = \mathit{Addr} \rightharpoonup_{fin} \mathit{Val} & S = \mathit{Var} \to \mathit{Val} \end{array}$$

We assume a set of program variables, ranged over by x, y, z, and a constant *nil*. An *expression* is either a constant or a program variable. Expressions are denoted by e. We ignore arithmetic expressions such as those allowed by Reynolds [34].

The syntax for formulae is given by:

$$F ::= \; e = e' \mid e \mapsto e' \mid e \mapsto e', e'' \mid \bot \mid F \to F \mid \top^* \mid F * F \mid F \mathrel{-\!\!*} F \mid \exists x.F$$

The only atomic formulae are $\bot$, $\top^*$, $(e = e')$, $(e \mapsto e')$, and $(e \mapsto e', e'')$. The latter two are called the "points-to" predicates. The domain of the quantifier is the set of values. We assume the usual notion of free and bound variables in formulae. We prefer to write $\top^*$ for the empty heap constant *emp* to be consistent with the prior work for BBI and PASL [20, 19]. The points-to predicate $e \mapsto e'$ denotes a singleton heap sending the value of e to the value of e'. The connectives $*$ and $\mathrel{-\!\!*}$ denote heap composition and heap extension respectively. These two connectives are interpreted with the binary operator $\circ$ defined as $h_1 \circ h_2 = h_1 \cup h_2$ when h_1, h_2 have disjoint domains, and undefined otherwise. A *state* is a pair (s, h) of a store and a heap.

A *separation logic model* is a pair (S, H) of stores and heaps, both are non-empty as defined previously. The forcing relation between a state and the formulae is formally defined in Table 1. We write $[\![e]\!]_s$ to denote the valuation of an expression e by looking up the value of variables in e in the store s. We fix that $[\![nil]\!]_s = nil$. We write $s[x \mapsto v]$ to denote a stack that is identical to s, except possibly on the valuation of x, i.e., $s[x \mapsto v](x) = v$ and $s[x \mapsto v](y) = s(y)$ for $y \neq x$. A formula F is true at the state (s, h) if $(s, h) \Vdash F$, and it is *valid* if $(s, h) \Vdash F$ for every $s \in S, h \in H$.

The literature contains the following useful abbreviations:

$$e \mapsto \_ \equiv \exists x.e \mapsto x \qquad\qquad e \mapsto e_1, \cdots, e_n \equiv (e \mapsto e_1) * \cdots * (e + n - 1 \mapsto e_n)$$

The multi-field points-to predicate $e \mapsto e_1, \cdots, e_n$ has different interpretations in the literature. In Reynolds's notation, the formula $e \mapsto e_1, e_2$ is equivalent to $(e \mapsto e_1) * (e + 1 \mapsto e_2)$, thus it is a heap of size two. However, in other versions of SL, the set of heaps may be defined as finite partial functions from addresses to pairs of values [10, 7], as shown below left:

$$H = \mathit{Addr} \rightharpoonup_{fin} Val \times Val \qquad H = \mathit{Addr} \rightharpoonup_{fin} (\mathit{Fields} \to Val)$$

In this setting the formula $e \mapsto e_1, e_2$ is a singleton heap. A more general case can be found in the definition of symbolic heaps [2] with heaps defined as shown

$s, h \Vdash \bot$ iff never	$s, h \Vdash \top^*$ iff $h = \emptyset$
$s, h \Vdash e = e'$ iff $[\![e]\!]_s = [\![e']\!]_s$	$s, h \Vdash A \to B$ iff $s, h \Vdash A$ implies $s, h \Vdash B$
$s, h \Vdash e \mapsto e'$ iff $dom(h) = \{[\![e]\!]_s\}$ and $h([\![e]\!]_s) = [\![e']\!]_s$	
$s, h \Vdash \exists x.A$ iff $\exists v \in Val$ such that $s[x \mapsto v], h \Vdash A$	
$s, h \Vdash A * B$ iff $\exists h_1, h_2.(h_1 \circ h_2 = h$ and $s, h_1 \Vdash A$ and $s, h_2 \Vdash B)$	
$s, h \Vdash A \mathrel{-\!\!*} B$ iff $\forall h_1, h_2.(h_1 \circ h = h_2$ and $s, h_1 \Vdash A)$ implies $s, h_2 \Vdash B)$	

Table 1. The semantics of the assertion logic of separation logic.

above right with a slight modification to make addresses a subset of values. *Fields* are simply the names for the data being pointed to.

The syntax of SL in this paper is more expressive than the popular *symbolic heaps* fragment of SL [2], which is restricted to the following syntax:

$$P ::= e = e' \mid \neg P \qquad\qquad \Pi ::= \top \mid P \mid \Pi \wedge \Pi$$
$$S ::= e \mapsto [f : e] \qquad\qquad \Sigma ::= \top^* \mid S \mid \Sigma * \Sigma$$

The $\mapsto$ predicate in symbolic heaps allows a list $[f : e]$ of fields, where f is the name of a field, and e is the content. Symbolic heaps are pairs $\Pi \wedge \Sigma$. The entailment of symbolic heaps is written as $\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'$. Symbolic heaps also allow formulae of the form $e \mapsto \_$ which does not specify the content of the heap.

3 LS_{SL}: A Labelled Sequent Calculus for SL

Let LVar be an infinite set of *label variables*, the set $\mathcal{L}$ of *labels* is LVar$\cup\{\epsilon\}$, where $\epsilon \notin$ LVar is a label constant. Labels are ranged over by h. We may sometimes use "heap" to mean a label h or a $\mapsto$ atomic formula. A *labelled formula* has the form $h : F$, where h is a label and F is a formula. We use ternary relational atoms $(h_1, h_2 \triangleright h_3)$ to indicate that the composition of the heaps represented by h_1, h_2 gives the heap represented by h_3. A *sequent* takes the form $\mathcal{G}; \Gamma \vdash \Delta$ where $\mathcal{G}$ is a set of ternary relational atoms, Γ, Δ are sets of labelled formulae, and ; denotes set union. Thus $\Gamma; h : A$ is the union of Γ and $\{h : A\}$. The left hand side of a sequent is the *antecedent* and the right hand side is the *succedent*.

The labelled sequent calculus LS_{SL} consists of inference rules taken from $LS_{PASL} + D + CS$ [19] with the addition of some special *id* rules, a cut rule for =, and the general rules for $\exists$ and =, as shown in Figure 1 and Figure 2, and the rules for the $\mapsto$ predicate, as shown in Figure 3, which are new to this paper. In these figures we write A, B for formulae. Although our proof system is incomplete for SL with heap model semantics and it may not be complete even for the quantifier-free fragment, the underlying system $LS_{PASL} + D + CS$ is complete for PASL with disjointness and cross-split [19]. The inference rules for the $\mapsto$ predicate with two fields are analogous to the rules in Figure 3.

A *label substitution* is a mapping from label variables to labels, which is an identity map except for a finite subset of $LVar$. We write $[h'_1/h_1, \ldots, h'_n/h_n]$ for a label substitution which maps h_i to h'_i. Label substitutions are extended to

$$\frac{}{\mathcal{G};\Gamma;h:e_1\mapsto e_2\vdash h:e_1\mapsto e_2;\Delta}\ {\scriptstyle id}\qquad\frac{}{\mathcal{G};\Gamma;h:e_1\mapsto e_2,e_3\vdash h:e_1\mapsto e_2,e_3;\Delta}\ {\scriptstyle id_2}$$

$$\frac{\mathcal{G};\Gamma[e_1/e_2]\vdash\Delta[e_1/e_2]\qquad\mathcal{G};\Gamma\vdash h:e_1=e_2;\Delta}{\mathcal{G};\Gamma\vdash\Delta}\ {\scriptstyle cut_=}$$

$$\frac{}{\mathcal{G};\Gamma;h:\bot\vdash\Delta}\ {\scriptstyle \bot L}\qquad\frac{\mathcal{G}[\epsilon/h];\Gamma[\epsilon/h]\vdash\Delta[\epsilon/h]}{\mathcal{G};\Gamma;h:\top^*\vdash\Delta}\ {\scriptstyle \top^* L}\qquad\frac{}{\mathcal{G};\Gamma\vdash\epsilon:\top^*;\Delta}\ {\scriptstyle \top^* R}$$

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

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

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

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

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

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

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

Fig. 1. Logical rules in LS_{SL}.

mappings between labelled formulae and labelled sequents in the obvious way. An *expression substitution* is defined similarly, where the domain is the set of program variables and the codomain is the set of expressions. We use θ (possibly with subscripts) to range over expression substitutions, and write $e\theta$ for the result of applying θ to e. Given a set of pairs of expressions $E = \{(e_1, e'_1), \ldots, (e_n, e'_n)\}$, a *unifier* for E is an expression substitution θ such that $\forall i,\ e_i\theta = e'_i\theta$. We assume the usual notion of the *most general unifier* (mgu) from logic programming. We denote with $mgu(E)$ the most general unifier of E when it exists. The formulae (resp. relational atoms) shown explicitly in the conclusion of a rule are called *principal formulae* (resp. *principal relational atoms*). A formula F is *provable* or *derivable* if there is a derivation of the sequent $\vdash h : F$ for an arbitrary $h \in \mathrm{LVar}$.

A *label mapping* ρ is a function $\mathcal{L} \to H$ such that $\rho(\epsilon) = \emptyset$. We define an *extended separation logic model* (S, H, s, ρ) as a separation logic model plus a stack and a label mapping.

Theorem 1 (Soundness). *For any formula F, and for any $h \in \mathrm{LVar}$, if the sequent $\vdash h : F$ is derivable in LS_{SL}, then F is valid in Reynolds's semantics.*

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

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

$$\frac{(\epsilon, h_2 \triangleright h_2); \mathcal{G}[h_2/h_1]; \Gamma[h_2/h_1] \vdash \Delta[h_2/h_1]}{(\epsilon, h_1 \triangleright h_2); \mathcal{G}; \Gamma \vdash \Delta}\, Eq_1 \qquad \frac{(\epsilon, h_1 \triangleright h_1); \mathcal{G}[h_1/h_2]; \Gamma[h_1/h_2] \vdash \Delta[h_1/h_2]}{(\epsilon, h_1 \triangleright h_2); \mathcal{G}; \Gamma \vdash \Delta}\, Eq_2$$

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

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

Side conditions:
Each label being substituted cannot be ϵ, each expression being substituted cannot be *nil*.
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. 2. Structural rules in LS_{SL}.

The rules $\mapsto L_1, L_2$ specify that $e_1 \mapsto e_2$ is a singleton heap, so it cannot be empty, nor a composite heap. These were all anticipated in [19]. However, the $\mapsto L_3$ rule proposed in [19], which says that any two singleton heaps with the same address are identical, is unsound for Reynolds's semantics. The corresponding $\mapsto L_3$ rule in Figure 3 is correct, which states that it is fine to have two singleton heaps with the same address, but they cannot be combined to form another heap. The rules $\mapsto L_5$ and $\mapsto L_4$ state that singleton heaps are uniquely determined by the $\mapsto$ relation. The rule *NIL* states that *nil* is not a valid address.

Since the set of addresses is infinite, we can extend any heap with fresh addresses, giving rise to the rule HE. The rule HC captures heap composition: given any two heaps h_1, h_2, either they can be combined, giving the right premise; or they cannot be combined, hence their domains intersect, i.e., there is some e_1 whose value is in this intersection, yielding the left premise. To our knowledge, proof systems for SL in the literature do not have rules similar to HE and HC, which enable us to prove many formulae that no other systems can prove.

4 Comparison with Existing Proof Calculi

This section compares and contrasts our calculus with existing proof calculi for "separation logics" and points out some subtleties in the literature.

Formula 1 says that any heap can be combined with a composite heap:

$$\neg(((\neg\top^*) * (\neg\top^*)) -\!\!* \bot) \tag{1}$$

The key to proving this formula is to show that any heap can be extended with a heap that contains at least two singleton mappings. This can be done using the rule HE. We show here the key part of the derivation for the above formula.

$$\frac{}{\mathcal{G}; \varGamma; \epsilon : e_1 \mapsto e_2 \vdash \varDelta} \mapsto L_1 \qquad \frac{(h_1, h_0 \rhd h_2); \mathcal{G}; \varGamma; h_1 : e_1 \mapsto e_2 \vdash \varDelta}{\mathcal{G}; \varGamma \vdash \varDelta} HE$$

$$\frac{\begin{array}{c}(\epsilon, h_0 \rhd h_0); \mathcal{G}[\epsilon/h_1, h_0/h_2]; \varGamma[\epsilon/h_1, h_0/h_2]; h_0 : e_1 \mapsto e_2 \vdash \varDelta[\epsilon/h_1, h_0/h_2] \\ (h_0, \epsilon \rhd h_0); \mathcal{G}[\epsilon/h_2, h_0/h_1]; \varGamma[\epsilon/h_2, h_0/h_1]; h_0 : e_1 \mapsto e_2 \vdash \varDelta[\epsilon/h_2, h_0/h_1]\end{array}}{(h_1, h_2 \rhd h_0); \mathcal{G}; \varGamma; h_0 : e_1 \mapsto e_2 \vdash \varDelta} \mapsto L_2$$

$$\frac{}{(h_1, h_2 \rhd h_0); \mathcal{G}; \varGamma; h_1 : e \mapsto e_1; h_2 : e \mapsto e_2 \vdash \varDelta} \mapsto L_3 \quad \frac{\mathcal{G}; \varGamma\theta; h : e_1\theta \mapsto e_2\theta \vdash \varDelta\theta}{\mathcal{G}; \varGamma; h : e_1 \mapsto e_2; h : e_3 \mapsto e_4 \vdash \varDelta} \mapsto L_4$$

$$\frac{\mathcal{G}[h_1/h_2]; \varGamma[h_1/h_2]; h_1 : e_1 \mapsto e_2 \vdash \varDelta[h_1/h_2]}{\mathcal{G}; \varGamma; h_1 : e_1 \mapsto e_2; h_2 : e_1 \mapsto e_2 \vdash \varDelta} \mapsto L_5 \qquad \frac{}{\mathcal{G}; \varGamma; h : nil \mapsto e \vdash \varDelta} NIL$$

$$\frac{(h_3, h_4 \rhd h_1); (h_5, h_6 \rhd h_2); \mathcal{G}; \varGamma; h_3 : e_1 \mapsto e_2; h_5 : e_1 \mapsto e_3 \vdash \varDelta \qquad (h_1, h_2 \rhd h_0); \mathcal{G}; \varGamma \vdash \varDelta}{\mathcal{G}; \varGamma \vdash \varDelta} HC$$

Side conditions:
Each label being substituted cannot be ϵ, each expression being substituted cannot be nil.
In $\mapsto L_4$, $\theta = mgu(\{(e_1, e_3), (e_2, e_4)\})$.
In HE, h_0 occurs in conclusion, h_1, h_2, e_1 are fresh.
In HC, h_1, h_2 occur in the conclusion, $h_0, h_3, h_4, h_5, h_6, e_1, e_2, e_3$ are fresh in the premise.

Fig. 3. Pointer rules in LS_{SL}.

$$\cfrac{\cfrac{\cfrac{(h_2, h_3 \rhd h_4); (h_0, h_1 \rhd h_2); h_0 : ((\neg\top^*) * (\neg\top^*)) \mathrel{-\!\!*} \bot; h_1 : e_1 \mapsto e_2; h_3 : e_3 \mapsto e_4 \vdash}{(h_0, h_1 \rhd h_2); h_0 : ((\neg\top^*) * (\neg\top^*)) \mathrel{-\!\!*} \bot; h_1 : e_1 \mapsto e_2 \vdash} HE}{; h_0 : ((\neg\top^*) * (\neg\top^*)) \mathrel{-\!\!*} \bot \vdash} HE}{; \vdash h_0 : \neg(((\neg\top^*) * (\neg\top^*)) \mathrel{-\!\!*} \bot)} \neg R$$

To our knowledge, current proof systems for separation logic lack this kind of mechanism. It is possible to prove this formula by changing or adding some rules in resource graph tableaux [15], but their proof relies on the restriction that every l in $(l \mapsto e)$ is an address. Thus their method cannot be used in a more general situation like ours.

Formula 2 is another interesting example, it is not valid in Reynolds's separation logic and not provable in LS_{SL}, but is provable in Lee and Park's system [24].

$$(((e_1 \mapsto e_2) * \top) \mathrel{-\!\!*} \bot) \vee (((e_1 \mapsto e_3) * \top) \mathrel{-\!\!*} \neg((e_1 \mapsto e_2) \mathrel{-\!\!*} \bot)) \vee (e_2 = e_3) \quad (2)$$

The meaning of Formula 2 is not straightforward, but we can construct a counter-model for it in Reynolds's semantics by trying to derive it in LS_{SL}. The following sequent will occur in the backward proof search for Formula 2:

$$\begin{array}{c}(h_5, h_6 \rhd h_1); (h_7, h_8 \rhd h_3); (h_1, h_0 \rhd h_2); (h_3, h_0 \rhd h_4); \\ h_5 : (e_1 \mapsto e_2); h_7 : (e_1 \mapsto e_3); h_4 : (e_1 \mapsto e_2) \mathrel{-\!\!*} \bot \vdash h_0 : (e_2 = e_3)\end{array}$$

It is easy to see that the above is a counter-model, and h_5 cannot be combined with h_4. By using Park et al.'s rule $Disj\mathrel{-\!\!*}$, we obtain

$$(v_1, v_2 \rhd h_1); (v_2, v_3 \rhd h_3); (v_1, h_4 \rhd w); (v_3, h_2 \rhd w)$$

where v_1, v_2, v_3, w are fresh. The common subheap v_2 of h_1 and h_3 cannot contain h_5, so h_5 must be in v_1. However, if v_1 can be combined with h_4, so can h_5, contradiction. Thus their rule $Disj{-\!*}$ is unsound in Reynolds's semantics.

As mentioned before, our proof system is not complete. For a valid example that cannot be proved by LS_{SL}, consider Formula 3:

$$\top^* \vee (\exists e1, e2.(e1 \mapsto e2)) \vee ((\neg \top^*) * (\neg \top^*)) \tag{3}$$

This formula is valid because any heap can only be either (1) an empty heap, or (2) a singleton heap, or (3) a composite heap. To prove Formula 3, we can add a $\mapsto R$ rule with four premises:

$$\frac{\begin{array}{ll}(1) & (h_1, h_2 \triangleright h); (h_1 \neq \epsilon); (h_2 \neq \epsilon); \mathcal{G}; \mathit{\Gamma} \vdash h : e_1 \mapsto e_2; \mathit{\Delta} \\ (2) & \mathcal{G}; \mathit{\Gamma}; h : e_1 \mapsto e_3 \vdash h : e_2 = e_3; h : e_1 \mapsto e_2; \mathit{\Delta} \\ (3) & \mathcal{G}; \mathit{\Gamma}; h : e_3 \mapsto e_4 \vdash h : e_1 = e_3; h : e_1 \mapsto e_2; \mathit{\Delta} \\ (4) & \mathcal{G}[\epsilon/h]; \mathit{\Gamma}[\epsilon/h] \vdash \epsilon : e_1 \mapsto e_2; \mathit{\Delta}[\epsilon/h]\end{array}}{\mathcal{G}; \mathit{\Gamma} \vdash h : e_1 \mapsto e_2; \mathit{\Delta}} \mapsto R$$

The rule $\mapsto R$ essentially negates the semantics for $\mapsto$, giving four possibilities when $e_1 \mapsto e_2$ is false at a heap h: (1) h is a composite heap, so it is possible to split it into two non-empty heaps; (2) h is a singleton heap, its address is the value of e_1, but it does not map this address to the value of e_2; (3) h is a singleton heap, but its address is not the value of e_1; (4) h is the empty heap. We will also need a new type of expression, namely inequality of labels as shown in the first premise. The rule $\mapsto R$ is not included in our proof system for efficiency. For more interesting formula and their derivations, see [18].

5 Inference Rules for Data Structures

Many data structures can be defined inductively by using separation logic's assertion language [7]. We focus here on the widely used singly linked lists. The treatment for binary trees is similar [18]. We adopt several rules from Berdine et al.'s method for symbolic heaps entailment [2] and extend these rules with new ones for formulae outside the symbolic heaps fragment.

We use the definition of linked lists for provers for symbolic heaps [7, 2], i.e.,

$$ls(e_1, e_2) \iff (e_1 = e_2 \wedge \top^*) \vee (e_1 \neq e_2 \wedge \exists x.(e_1 \mapsto x * ls(x, e_2)))$$

to facilitate comparison between our prover and the other provers. The inference rules for singly linked lists are given in Figure 4. The rules LS_6 and LS_7 are for non-symbolic heaps, they handle cases where two lists overlap. There $ds(e, e')$ stands for a data structure that starts from the address e, and ends with e'. We use $ad(e)$ for a data structure that *may* contain the address of value of e, and use $G(ad(e))$ in the succedent to ensure that $ad(e)$ is non-empty.

For LS_8, suppose the heap h_1 is a data structure from e_1 to e_2, and h_3 is a data structure that mentions e_3. By $G(ad(e_3))$ in the succedent, we know that h_3 is non-empty and indeed contains the address of e_3. Since $(h_1, h_3 \triangleright h_4)$ holds,

$$\frac{\mathcal{G}; \Gamma[e_1/e_2] \vdash \Delta[e_1/e_2]}{\mathcal{G}; \Gamma; \epsilon : ls(e_1, e_2) \vdash \Delta}\ {\scriptstyle LS_1} \qquad \frac{}{\mathcal{G}; \Gamma \vdash \epsilon : ls(e, e); \Delta}\ {\scriptstyle LS_2} \qquad \frac{\mathcal{G}; \Gamma; h : \top^* \vdash \Delta}{\mathcal{G}; \Gamma; h : ls(e, e) \vdash \Delta}\ {\scriptstyle LS_3}$$

$$\frac{\mathcal{G}; \Gamma[nil/e]; h : \top^* \vdash \Delta[nil/e]}{\mathcal{G}; \Gamma; h : ls(nil, e) \vdash \Delta}\ {\scriptstyle LS_4} \qquad \frac{}{\mathcal{G}; \Gamma; h : A \vdash h : A; \Delta}\ {\scriptstyle id_a}$$

$$\frac{\mathcal{G}; \Gamma\theta_1; h : \top^* \vdash \Delta\theta_1 \qquad \mathcal{G}; \Gamma\theta_2; h : ls(e_1\theta_2, e_2\theta_2) \vdash \Delta\theta_2}{\mathcal{G}; \Gamma; h : ls(e_1, e_2); h : ls(e_3, e_4) \vdash \Delta}\ {\scriptstyle LS_5}$$

$$\frac{(h_1, h_2 \triangleright h_0); \mathcal{G}; \Gamma; h_1 : ds(e_1, e_2); h_0 : ls(e_1, e_3); h_2 : ls(e_2, e_3) \vdash \Delta}{(h_1, h_2 \triangleright h_0); \mathcal{G}; \Gamma; h_1 : ds(e_1, e_2); h_0 : ls(e_1, e_3) \vdash \Delta}\ {\scriptstyle LS_6}$$

$$\frac{(h_1, h_2 \triangleright h_0); \mathcal{G}; \Gamma; h_1 : ds(e_2, e_3); h_0 : ls(e_1, e_3); h_2 : ls(e_1, e_2) \vdash \Delta}{(h_1, h_2 \triangleright h_0); \mathcal{G}; \Gamma; h_1 : ds(e_2, e_3); h_0 : ls(e_1, e_3) \vdash \Delta}\ {\scriptstyle LS_7}$$

$$\frac{\begin{array}{c}(h_1, h_2 \triangleright h_0); (h_1, h_3 \triangleright h_4); \mathcal{G};\\ \Gamma; h_1 : ds(e_1, e_2); h_3 : ad(e_3)\end{array} \vdash h_2 : ls(e_2, e_3); h_0 : ls(e_1, e_3); h : G(ad(e_3)); \Delta}{(h_1, h_2 \triangleright h_0); (h_1, h_3 \triangleright h_4); \mathcal{G}; \Gamma; h_1 : ds(e_1, e_2); h_3 : ad(e_3) \vdash h_0 : ls(e_1, e_3); h : G(ad(e_3)); \Delta}\ {\scriptstyle LS_8}$$

$$\frac{}{(h_1, h_2 \triangleright h_0); \mathcal{G}; \Gamma; h_1 : ad(e_1); h_2 : ad(e_1)' \vdash h_3 : G(ad(e_1)); h_4 : G(ad(e_1)'); \Delta}\ {\scriptstyle IC}$$

Abbreviations and side conditions:
$ds(e, e')$ is either $(e \mapsto e')$ or $ls(e, e')$.
$ad(e)$ stands for one of $(e \mapsto e'), (e \mapsto e', e''), ls(e, e')$, or $tr(e)$, for some e', e''. Similarly for $ad(e)'$.
$G(ad(e))$ is defined as $G(e \mapsto e') \equiv G(e \mapsto e', e'') \equiv \bot, G(ls(e, e')) \equiv (e = e'), G(tr(e)) \equiv (e = nil)$.
In LS_5, $\theta_1 = mgu(\{(e_1, e_2), (e_3, e_4)\})$ and $\theta_2 = mgu(\{(e_1, e_3), (e_2, e_4)\})$.
In LS_8, if e_3 is nil, then $(h_1, h_3 \triangleright h_4), h_3 : ad(e_3)$ and $h : G(ad(e_3))$ in the conclusion are optional.
In LS_8, if $ds(e_1, e_2)$ is $(e_1 \mapsto e_2)$, then $(h_1, h_3 \triangleright h_4)$, $h_3 : ad(e_3)$ and $h : G(ad(e_3))$ in the conclusion are optional, on the condition that $h' : (e_1 = e_3)$ occurs in the RHS of the conclusion, for some h'.

Fig. 4. Inference rules for data structures.

the address e_3 is not in the domain of h_1. The labelled formula $h_0 : ls(e_1, e_3)$ in the succedent indicates that h_0 should also make $ds(e_1, e_2) * ls(e_2, e_3)$ false, thus by an $*R$ application on this formula using $(h_1, h_2 \triangleright h_0)$, the branch with $h_1 : ds(e_1, e_2)$ in the succedent can be closed, and we only have the other branch with $h_2 : ls(e_2, e_3)$ in the succedent. There are two special cases as indicated by the side conditions. First, if e_3 is nil, then e_3 can never be an address. Thus we do not need $(h_1, h_3 \triangleright h_4)$, $h_3 : ad(e_3)$ and $h : G(ad(e_3))$ in the conclusion. Second, if $ds(e_1, e_2)$ is a singleton heap $(e_1 \mapsto e_2)$, then we only require that e_3 does not have the same value as e_1, thus $(h_1, h_3 \triangleright h_4)$, $h_3 : ad(e_3)$ and $h : G(ad(e_3))$ can be neglected as long as $(e_1 = e_3)$ occurs in the succedent.

The rules IC and id_a respectively generalise $\mapsto L_3$ and id. Thus IC captures that two data structures that contain the same address cannot be composed by $*$, and id_a simply forbids a heap to make a formula both true and false.

We refer to the labelled system LS_{SL} plus the rules introduced in Figure 4 and the rules for binary trees (not shown here, but can be found in [18]) as $LS_{SL} + DS$. The soundness of $LS_{SL} + DS$ for Reynolds's semantics can be proved in the same way as previously showed for Theorem 1.

$$\frac{}{\mathcal{G};\Gamma;h:e\mapsto\alpha,\alpha'\vdash h:e\mapsto\alpha;\Delta}\qquad\qquad\frac{}{\mathcal{G};\Gamma;\epsilon:e\mapsto\alpha\vdash\Delta}$$

$$\frac{}{\mathcal{G};\Gamma;h:nil\mapsto\alpha\vdash\Delta}\qquad\frac{\mathcal{G};\Gamma\theta;h:e_1\mapsto\alpha\vdash\Delta\theta}{\mathcal{G};\Gamma;h:e_1\mapsto\alpha;h:e_2\mapsto\alpha'\vdash\Delta}\ \theta=mgu(\{(e_1,e_2),(\alpha,\alpha')\})$$

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

$$\frac{\mathcal{G}[h_1/h_2];\Gamma[h_1/h_2];h_1:e_1\mapsto\alpha,\alpha';h_1:e_1\mapsto\alpha\vdash\Delta[h_1/h_2]}{\mathcal{G};\Gamma;h_1:e_1\mapsto\alpha,\alpha';h_2:e_1\mapsto\alpha\vdash\Delta}$$

Fig. 5. Generalised rules for $\mapsto$ with arbitrary fields in non-Reynolds's semantics.

Recall that symbolic heaps employ slightly different semantics for the multi-field $\mapsto$ predicate, and treat it as a singleton heap. This reading would not make sense in our setting because our logic is based on Reynolds's semantics. Here we develop a branch of our system by compromising both kinds of semantics and viewing $(e_1 \mapsto e_2, e_3)$ as a singleton heap that maps the value of e_1 to the value of e_2, and the next address contains the value of e_3. We give the generalised $\mapsto$ rules for non-Reynolds's semantics in Figure 5 where α, α' denote any number of fields. For the non-Reynolds's semantics, the rules in Figure 4 need to be adjusted so that $ds(e1, e2)$ now considers $(e_1 \mapsto e_2, \alpha)$ and $ad(e)$ considers $(e \mapsto \alpha)$. We refer to the variant of $LS_{SL} + DS$ with these changes and the addition of rules in Figure 5 as $LS'_{SL} + DS$, which is complete for the symbolic heaps fragment; see [18] for the proof.

Theorem 2. *Any symbolic heap formula provable in $LS'_{SL} + DS$ is valid, and any valid symbolic heap formula is provable in $LS'_{SL} + DS$.*

6 Proof Search and Experiment

This section describes proof search and automated reasoning based on the system $LS'_{SL} + DS$, these tactics can also be used on the variant $LS_{SL} + DS$.

We have implemented our labelled calculus $LS'_{SL} + DS$ as a prover called Separata+, in which several restrictions for the logical and structural rules are incorporated without sacrificing provability. See Figure 1 and 2 for the related inference rules in LS'_{SL}. Some of these restrictions are also used in the prover for PASL [19]. The rule U only creates identity relations for existing labels. The rule A is only applicable when the following holds: if the principal relational atoms are $(h_1, h_2 \triangleright h_0)$ and $(h_3, h_4 \triangleright h_1)$, then the conclusion does not contain $(h_3, h \triangleright h_0)$ and $(h_2, h_4 \triangleright h)$, or any commutative variants of them, for any h.

In applying the cross-split rule CS, we choose the principal relational atoms such that the parent label has the least number of children. Other strategies to apply cross-split are possible; see e.g., [24]. Calcagno et al. [10] showed how to deal with $-\!\!*$ formulae in the quantifier-free fragment, but we do not know whether their results hold for our SL. Nevertheless, inspired by their result, the rules HE, HC in our prover are driven by $-\!\!*$ formulae in the antecedent. We first define a notion of the *size* of a formula as below:

$$\begin{array}{llll} |e \mapsto e'| = |e \mapsto e', e''| = 1 & |e = e'| = 0 & |\bot| = 0 & |A * B| = |A| + |B| \\ |A \to B| = max(|A|, |B|) & |\exists x.A| = |A| & |\top^*| = 1 & |A -\!\!* B| = |B| \end{array}$$

Given a labelled formula $h : A -\!\!* B$ in the antecedent of a sequent, we allow to use the HE rule to extend h for at most $max(|A|, |B|)/2 + 1$ times instead of $max(|A|, |B|)$ as indicated in [10], because we do not worry about completeness w.r.t. SL here. The HC rule is restricted to only combine three types of heaps: any singleton heaps that occur as subformulae of $A -\!\!* B$; any heaps created by HE for $A -\!\!* B$; and any compositions of the previous two. The restrictions on HE and HC are parameters which can be fine-tuned for specific applications.

The atomic formula $e \mapsto \_$, translated to $\exists x.(e \mapsto x)$, is the only type of formula in symbolic heaps that involves quantifiers. Since nested quantifiers are forbidden in symbolic heaps, the $\exists R$ rule can be restricted so that it only instantiates the quantified variable with an existing expression or *nil*. We call this restricted version $\exists R'$. Although not explicitly allowed in the symbolic heaps fragment nor in our assertion logic, some symbolic heaps provers can recognise numbers. To match them, we check when a rule wants to globally replace a number (expression) by another number, and close the branch immediately because two distinct numbers should not be made equal. The rule $cut_=$ is restricted to apply only on existing expressions and the constant *nil*.

Our proof search procedure for $LS'_{SL} + DS$ builds in the above tactics, and applies the first applicable rule in the following order:

1. Any zero-premise rule.
2. Any unary rule that involves global substitutions.
3. Any other unary non-structural rule except $\exists R$.
4. Any binary rule that involves global substitutions except $cut_=$.
5. $\to L$. 6. $*R$, $-\!\!* L$ and $\exists R'$. 7. U, E, A, CS. 8. $cut_=$.

Theorem 3 (Termination for symbolic heaps). *The proof search procedure for $LS'_{SL} + DS$ is complete and terminating for the symbolic heaps fragment.*

The experiments were run on a machine with a Core i7 2600 3.4GHz processor and 8GB memory, in Ubuntu 14.04. The code is written in OCaml. Our prover and test suites can be found at [17]. The proof of theorems is in [18].

Our first experiment compares our prover with state-of-the-art provers for symbolic heaps using the Clones benchmark from Navarro and Rybalchenko [27], which is generated from "real life" list manipulating programs and specifications involved in verification. We filter out problems that contain a data structure that we do not consider in this paper, the remaining set consists of 164 valid formulae and 39 invalid formulae. Each Clones test set has the same type of formulae, but the length (number of copies of subformulae) of formulae increases from Clones 1 to Clones 10. We compare our prover with Asterix, Smallfoot, and Cyclist$_{\text{SL}}$ [8], the last of which is designed for a $\forall\exists$ DNF-like fragment of separation logic.

Cyclist$_{\mathrm{SL}}$ cannot recognise numbers, and there are 17 formulae in each Clones test set that cannot be parsed by it (counted as not proved).

Table 2 shows the results of the first experiment. Time out is 50 seconds. The proved column for each prover shows the number of formulae the prover proves or disproves within the time out, the avg. time column shows the average time used when successfully proving a formula. Unsuccessful attempts counted in average time. Asterix outperformed all the compared provers. Cyclist$_{\mathrm{SL}}$ is not complete, so it might terminate without giving a proof. It also cannot determine if a formula is invalid. However, the advantage of Cyclist$_{\mathrm{SL}}$ is not in its performance, but in its generality. For example, Cyclist$_{\mathrm{SL}}$ can be easily extended to handle other inductive definitions, this is not the case for the other provers in comparison. Separata+ and Smallfoot have similar performance on valid formulae, but Separata+ is not efficient on invalid formulae.

The second experiment features some formulae outside the symbolic heaps fragment, thus we cannot find other provers to compare with, except for a recent work by Thakur, Breck, and Reps [36]. However, their semantics assume acyclic heaps. For example, $(e_1 \mapsto e_2) * (e_2 \mapsto e_1)$ is a satisfiable formula in Reynolds's semantics, but is unsatisfiable in Thakur et al's semantics. The fragment of separation logic they consider has "septraction" $A -\!\!\circledast B$, defined as $\neg(A -\!\!* \neg B)$, and only allows classical negation on atomic formulae. Table 3 shows some formulae derived from [36, Table 3], using the definition of septraction as given above. The other formulae from [36, Table 3] are not included, as they are unsatisifiable in Reynolds's semantics. Formula T3.3 to T3.13 in Table 3 are identified as "beyond the scope of existing tools" [36]. More specifically, Formula T3.1, T3.3 and T3.4 describe overlapping data structures; the other formulae in Table 3 demonstrate the use of list and septraction. For example, Formula T3.6 is an instance of an elimination rule for $-\!\!\circledast$ and linked list segment in [9].

Maeda, Sato, and Yonezawa [26] provide more examples that use $-\!\!*$ in program verification. Many of their inferences, e.g. [26, Section 3.1], are easily proved by Separata+ if their syntax is carefully translated into ours, such as Formula 4, which captures a property described in their original type system.

$$\begin{aligned}(ls(e_0, nil) -\!\!* (ls(e_0, nil) * (ls(e_0, nil) -\!\!* ((ls(e_1, nil) -\!\!* ls(e_2, nil)) * (e_1 \mapsto e_3) *\\ ls(e_0, nil))) * (ls(e_0, nil) -\!\!* ls(e_3, nil)))) \to (ls(e_0, nil) -\!\!* (((ls(e_1, nil)\\ -\!\!* ls(e_2, nil)) * (e_1 \mapsto e_3) * ls(e_0, nil)) * (ls(e_0, nil) -\!\!* ls(e_3, nil)))) \quad (4)\end{aligned}$$

To challenge our prover further, we build larger formulae generated from Table 3, Formulae 1, 4 (and some formulae similar to 4) and some formulae in [18], totalling 26 formulae inexpressible in symbolic heaps. We use the "clone" method [1] to generate larger formulae, but we make the formulae "harder" by randomly switching the order of starred subformulae. We call these test suites "MClones". The test results are shown in Table 4. The MClones 1 set contains 26 original formulae. The experiment method is the same as before, except that the timeout is set to 500 seconds. The successful rate drops as the number of cloned subformulae increases. The average time used to prove a formulae, however, fluctuates, because we do not count the timed out attempts. In both experiments,

	Test suite with 164 valid formulae						Test suite with 39 invalid formulae					
Test suite	Separata+		Cyclist$_{\mathrm{SL}}$		Smallfoot		Separata+		Cyclist$_{\mathrm{SL}}$		Smallfoot	
	proved	avg. time	proved	avg. time	proved	avg. time	dis-proved	avg. time	dis-proved	avg. time	dis-proved	avg. time
Clones 1	164	0.01	147	0.04	164	0.00	39	0.09	0	-	39	0.00
Clones 2	160	0.02	137	0.17	164	0.00	23	3.37	0	-	39	0.00
Clones 3	159	0.07	126	0.48	164	0.01	9	1.78	0	-	39	0.01
Clones 4	159	0.30	117	0.11	164	0.03	6	7.89	0	-	39	0.02
Clones 5	158	0.03	115	0.13	164	0.15	2	0.52	0	-	39	0.10
Clones 6	158	0.08	114	0.29	164	0.65	2	20.10	0	-	39	0.40
Clones 7	158	0.18	106	0.01	162	0.75	0	-	0	-	39	0.00
Clones 8	158	0.42	106	0.01	160	0.83	0	-	0	-	38	2.10
Clones 9	158	0.89	106	0.01	157	0.36	0	-	0	-	38	5.37
Clones 10	157	1.19	106	0.01	157	0.83	0	-	0	-	32	3.54

Asterix proved every test set with an average of 0.01s and 100% successful rate.
Table 2. Experiment 1: the Clones benchmark. Times are in seconds.

	Formula
T3.1	$ls(e_1, e_2) \land \top^* \land \neg(e_1 = e_2)$
T3.2	$\neg((e_1 \mapsto e_2) \mathrel{-\!\!*} \neg\top) \land ((e_1 \mapsto e_2) * \top)$
T3.3	$(ls(e_1, e_2) * \neg ls(e_2, e_3)) \land ls(e_1, e_3)$
T3.4	$ls(e_1, e_2) \land ls(e_1, e_3) \land \neg\top^* \land \neg(e_2 = e_3)$
T3.5	$\neg(ls(e_1, e_2) \mathrel{-\!\!*} \neg ls(e_1, e_2)) \land \neg\top^*$
T3.6	$\neg((e_3 \mapsto e_4) \mathrel{-\!\!*} \neg ls(e_1, e_4)) \land ((e_3 = e_4) \lor \neg ls(e_1, e_3))$
T3.7	$\neg(\neg((e_2 \mapsto e_3) \mathrel{-\!\!*} \neg ls(e_2, e_4)) \mathrel{-\!\!*} \neg ls(e_1, e_4)) \land \neg ls(e_1, e_3)$
T3.8	$\neg(\neg((e_2 \mapsto e_3) \mathrel{-\!\!*} \neg ls(e_2, e_4)) \mathrel{-\!\!*} \neg ls(e_3, e_1)) \land (e_2 = e_4)$
T3.9	$\neg((e_1 \mapsto e_2) \mathrel{-\!\!*} \neg ls(e_1, e_3)) \land (\neg ls(e_2, e_3) \lor ((\top \land ((e_1 \mapsto e_4) * \top)) \lor (e_1 = e_3)))$
T3.10	$\neg((ls(e_1, e_2) \land \neg(e_1 = e_2)) \mathrel{-\!\!*} \neg ls(e_3, e_4)) \land \neg(e_3 = e_1) \land (e_4 = e_2) \land \neg ls(e_3, e_1)$
T3.11	$\neg(e_3 = e_4) \land \neg(ls(e_3, e_4) \mathrel{-\!\!*} \neg ls(e_1, e_2)) \land (e_4 = e_2) \land \neg ls(e_1, e_3)$
T3.12	$\neg((ls(e_1, e_2) \land \neg(e_1 = e_2)) \mathrel{-\!\!*} \neg ls(e_3, e_4)) \land \neg(e_3 = e_2) \land (e_3 = e_1) \land \neg ls(e_2, e_4)$
T3.13	$\neg(\neg((e_2 \mapsto e_3) \mathrel{-\!\!*} \neg ls(e_2, e_4)) \mathrel{-\!\!*} \neg ls(e_3, e_1)) \land (\neg ls(e_4, e_1) \lor (e_2 = e_4))$

Separata+ proved the negation of each listed formula within 0.01 second.
Table 3. Selected formulae from [36, Table 3] translated via $A \mathrel{-\!\!\circledast} B \equiv \neg(A \mathrel{-\!\!*} \neg B)$.

Test suite	Separata+		Test suite	Separata+	
	proved	avg. time		Proved	avg. time
MClones 1	26/26	2.96s	MClones 6	18/26	16.44s
MClones 2	23/26	8.76s	MClones 7	17/26	3.97s
MClones 3	20/26	7.00s	MClones 8	15/26	2.93s
MClones 4	20/26	0.62s	MClones 9	16/26	8.43s
MClones 5	20/26	22.35s	MClones 10	14/26	10.71s

Table 4. Mutated clones benchmark for formulae in Table 3.

the first test suite (Clones 1 and MClones 1) contains the original formulae in program verification. These formulae can be easily proved by Separata+.

7 Conclusion

We have presented a labelled sequent calculus LS_{SL} for Reynolds's SL. The syntax allows all the logical connectives in SL including $*, -\!*$ and quantifiers, the predicate $\mapsto$ and equality. It is impossible to obtain a finite, sound and complete sequent system for this logic, so we focused on soundness, usefulness, and efficiency. With the extension to handle data structures, our proof method is sound, complete, and terminating for the widely used symbolic heaps fragment. Our prover Separata+ showed comparable results as that for Smallfoot on proving valid formulae, although Separata+ does not perform well when the formula is invalid, which may be due to our inference rules having to cover a larger fragment. However, Separata+ can deal with many formulae that, to our knowledge, no other provers for Reynolds's SL can prove. Some of these formulae are taken from existing (manual) proofs to verify algorithms/programs. These indicate that our method would be useful, at least as a part of the tool chain, for program verification with more sophisticated use of separation logic.

Acknowledgment. The third author is partly supported by NTU start-up grant M4081190.020.

References

1. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137. Springer, 2005.
2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, volume 3780 of *LNCS*, pages 52–68, 2005.
3. J. Berdine and P. W. O'Hearn. Strong update, disposal, and encapsulation in bunched typing. *Electron. Notes Theor. Comput. Sci.*, 158:81–98, May 2006.
4. R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. In *MFPS*, volume 155 of *ENTCS*, pages 247–276, 2006.
5. R. Brochenin, S. Demri, and E. Lozes. On the almighty wand. *Inform. and Comput.*, 211:106–137, 2012.
6. S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, Apr. 2007.
7. J. Brotherston, D. Distefano, and R. L. Petersen. Automated cyclic entailment proofs in separation logic. CADE'11, pages 131–146. Springer-Verlag, 2011.
8. J. Brotherston, N. Gorogiannis, and R. Petersen. A generic cyclic theorem prover. In *APLAS-10*, volume 7705 of *LNCS*, pages 350–367. Springer, 2012.
9. C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS*, volume 4634 of *LNCS*, 2007.
10. C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FSTTCS*, volume 2245 of *LNCS*, pages 108–119, 2001.
11. S. Demri, D. Galmiche, D. Larchey-Wendling, and D. Mèry. Separation logic with one quantified variable. In *CSR*. LNCS 8476, Moscow, 2014.

12. S. Demri and M.Deters. Expressive completeness of separation logic with two variables and no separating conjunction. In *CSL/LICS*. Vienna, 2014.
13. D. Distefano and P. Matthew. jStar: Towards practical verification for java. In *ACM Sigplan Notices*, volume 43, pages 213–226. ACM, 2008.
14. R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, volume 5904 of *LNCS*, pages 161–177, 2009.
15. D. Galmiche and D. Méry. Tableaux and resource graphs for separation logic. *J. Logic Comput.*, 20(1):189–231, 2007.
16. A. Hobor and J. Villard. The ramifications of sharing in data structures. POPL '13, pages 523–536, New York, NY, USA, 2013. ACM.
17. Z. Hóu. Separata+. `http://users.cecs.anu.edu.au/~zhehou/`.
18. Z. Hóu. *Labelled Sequent Calculi and Automated Reasoning for Assertions in Separation Logic*. PhD thesis, The Australian National University, 2015. Submitted.
19. Z. Hóu, R. Clouston, R. Goré, and A. Tiu. Proof search for propositional abstract separation logics via labelled sequents. In *POPL*, pages 465–476. ACM, 2014.
20. Z. Hóu, A. Tiu, and R. Goré. A labelled sequent calculus for BBI: Proof theory and proof search. In *Tableaux*, LNCS, 2013. page 172-187.
21. J. Jensen. Techniques for model construction in separation logic. Report, 2013.
22. J. B. Jensen and L. Birkedal. Fictional separation logic. In *ESOP*, volume 7211 of *LNCS*, pages 377–396, 2012.
23. N. R. Krishnaswami. Reasoning about iterators with separation logic. In *SAVCBS*, pages 83–86. ACM, 2006.
24. W. Lee and S. Park. A proof system for separation logic with magic wand. POPL '14, pages 477–490, New York, NY, USA, 2014. ACM.
25. E. Maclean, A. Ireland, and G. Grov. Proof automation for functional correctness in separation logic. *Journal of Logic and Computation*, 2014.
26. T. Maeda, H. Sato, and A. Yonezawa. Extended alias type system using separating implication. TLDI '11, pages 29–42, New York, NY, USA, 2011. ACM.
27. J. A. Navarro Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. PLDI '11, pages 556–566, USA, 2011. ACM.
28. J. A. Navarro Pérez and A. Rybalchenko. Separation logic modulo theories. In *APLAS*, volume 8301 of *LNCS*, pages 90–106. Springer, 2013.
29. P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bull. Symbolic Logic*, 5(2):215–244, 1999.
30. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. CSL '01, pages 1–19. Springer-Verlag, 2001.
31. M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in hoare logics. In *In 21st LICS*, 2006.
32. B. Reus and J. Schwinghammer. Separation logic for higher-order store. In *CSL*, volume 4207 of *LNCS*, pages 575–590. Springer Berlin Heidelberg, 2006.
33. J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000.
34. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE, 2002.
35. G. Stewart, L. Beringer, and A. W. Appel. Verified heap theorem prover by paramodulation. In *ICFP*, pages 3–14. ACM, 2012.
36. A. Thakur, J. Breck, and T. Reps. Satisfiability modulo abstraction for separation logic with linked lists. Technical report, University of Wisconsin, 2014.
37. V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, volume 4703 of *LNCS*, pages 256–271, 2007.