

Automated Cyclic Entailment Proofs in Separation Logic

James Brotherston¹, Dino Distefano^{2,3}, and Rasmus L. Petersen²

¹ Dept. of Computing, Imperial College London

² Dept. of Computer Science, Queen Mary University of London

³ Monoidics Ltd

Abstract. We present a general automated proof procedure, based upon *cyclic proof*, for inductive entailments in separation logic. Our procedure has been implemented via a deep embedding of cyclic proofs in the HOL Light theorem prover. Experiments show that our mechanism is able to prove a number of non-trivial entailments involving inductive predicates.

1 Introduction

Separation logic [18] has recently become a very popular formalism for the verification of imperative, memory-manipulating programs. Proofs of programs in separation logic are based on the Hoare triples $\{P\}C\{Q\}$ familiar from first-order approaches to verification. However, the pre- and post-conditions of triples may contain a special *separating conjunction* $*$, which allows the disjointness of portions of heap memory to be expressed. Thus the formula $A * B$ denotes those heaps which can be separated into two disjoint parts satisfying respectively A and B . This characteristic feature enables one to construct proofs that are highly modular, and thus separation logic scales well to large programs. Indeed, there are now several tools based upon separation logic that are capable of verifying code on an industrial scale [8, 21, 13].

In this paper, we address the issue of automatically proving *entailments* $F_1 \models F_2$ between formulas in separation logic. In automatic verification tools based on Hoare triples, the obligation to prove such entailments typically arises via the standard *rule of consequence*:

$$\frac{\{P'\}C\{Q'\}}{\{P\}C\{Q\}} P \models P', Q' \models Q \text{ (Consq)}$$

This rule might be applied during a proof search, e.g., to remove redundant information from the precondition P , or to convert P and Q into a format which matches a rule for the command C . Other activities in which entailments need to be proved include abstraction [12] and discharging the guards of conditional commands during symbolic execution. Thus, effective procedures for establishing entailments are at the foundation of automatic verification based on separation logic. Due to the intense use of dynamically-allocated data structure in real-world

software (e.g., system code [21]), in practice, the pre- and postconditions occurring in separation logic proofs typically contain inductively defined predicates. Thus any proof-theoretic approach to establishing entailments is essentially a problem of *inductive theorem proving*, which is known to present serious difficulties for automated search (see [7] for an overview). Moreover, in the case of separation logic, the induction hypotheses required for an inductive proof are often not even expressible in the fragments of the logic handled by automatic tools since they require unsupported operators like the spatial implication $-*$.

Unfortunately, due to the current lack of off-the-shelf general theorem provers, most of the existing automated verification tools have to appeal to their own theorem prover for checking the validity of entailments. Because building them is a difficult and time-consuming activity, these provers tend to be rather *ad-hoc* and often do not provide support for inductive methods. Here, we present a prototype theorem prover for entailments of separation logic that uses *cyclic proof* to handle inductive theorems. Cyclic proof has recently been mooted as an alternative to the default approach of explicit inductive proof that offers potential advantages for automated proof search [4, 6]. Cyclic proofs differ from explicit induction proofs in two main respects. First, explicit induction rules are replaced by simple unfolding or “case split” rules for the inductively defined predicates. Second, proofs are allowed to contain cycles, and thus can be seen as infinite derivation trees. To ensure that such structures correspond to sound proofs, a global *soundness condition* is imposed on cyclic proofs guaranteeing the well-foundedness of all reasoning. The main attraction of cyclic proofs is that, unlike in standard induction proofs, the induction hypotheses are not supplied explicitly via the application of an induction rule. Instead, they are constructed implicitly via the discovery of a valid cyclic proof. This allows a much more exploratory approach to automated proof search.

Our theorem prover is implemented in HOL Light [15] and supports both fully automatic and interactive proof. The implementation of a cyclic proof system in HOL Light, or indeed any of the mainstream theorem provers, presents several non-trivial technical obstacles stemming from the fact that such provers take a *local* viewpoint of proofs, whereas cyclic proof is necessarily *global*. To overcome this mismatch, we employ a *deep embedding* of our mathematical cyclic proof system, i.e., a HOL Light representation in which cyclic proofs themselves are first-class objects. The main advantage of a fully explicit representation of this type is that we can easily impose the correct global soundness conditions on proofs. Although we employ a fairly simple such condition in this paper, we can easily impose more general conditions in order to improve completeness at the expense of speed. We have evaluated our implementation on a series of examples, drawn from the literature. Although our prover is only a prototype, the results are encouraging for their coverage as well as their performance.

The remainder of this paper is structured as follows. Section 2 introduces of our separation logic fragment. Section 3 introduces our cyclic proof machinery. Section 4 describes the implementation of our proof procedure and evaluates its performance. Section 5 compares with related work and Section 6 concludes.

2 Syntax and semantics

In this section we introduce the separation logic formulas that we shall consider throughout the paper, and their standard semantics with respect to a fixed heap model. We assume a fixed, infinite set \mathcal{V} of first-order variables and a fixed finite set of predicate symbols, each with associated arity.

Definition 1 (Formulas). *Formulas* are given inductively by the grammar:

$$F ::= \top \mid \perp \mid x = y \mid x \neq y \mid \mathbf{emp} \mid x \mapsto y \mid x \overset{2}{\mapsto} y, z \mid F \vee F \mid F * F \mid P\mathbf{x}$$

where x, y range over \mathcal{V} , P ranges over predicate symbols and \mathbf{x} ranges over tuples of variables of appropriate length to match the arity of P . We consider formulas up to associativity and commutativity of $*$ and \vee .

The fragment of separation logic considered here is relatively simple and does not include, for example, function symbols, plain conjunction (\wedge) or spatial implication (\multimap). These features are not typically employed in separation logic verification (in fact even \vee is often removed as well) because the complexity rapidly becomes unmanageable. In fact, it has been shown that unrestricted separation logic is undecidable even in the purely propositional setting [5].

Definition 2 (Inductive rule set). An *inductive rule set* is a finite set of *inductive rules* each of the form $F \Rightarrow P\mathbf{x}$ where F and $P\mathbf{x}$ are formulas with P a predicate symbol.

From now on we assume a fixed inductive rule set Φ .

Semantics. Let L be an infinite set of *locations*, and V be a set of *values*. Then $H = L \rightarrow_{\text{fin}} V$, the set of all finite partial functions from L to V , is called the set of *heaps*. We write $\text{dom}(h)$ to denote the *domain* of the heap h , i.e., the set $\{l \in L \mid h(l) \text{ is defined}\}$. Composition of heaps, $h_1 \circ h_2$, is defined as the union of h_1 and h_2 if their domains are disjoint, and undefined otherwise. The *empty heap* e is the heap such that $e(l)$ is undefined for all $l \in L$. It is easy to see that $\langle H, \circ, e \rangle$ is a *separation algebra* (cf. [9]), i.e., a cancellative partial commutative monoid. We sometimes choose to work with heaps of the form $H = L \rightarrow_{\text{fin}} V \times V$, where a pair of values is stored at each location, rather than a single value. In such cases we use the predicate $\overset{2}{\mapsto}$ in place of \mapsto . The set of *stacks* is $S = \mathcal{V} \rightarrow L \cup V$, the set of total functions from first-order variables to locations or values (which are not necessarily disjoint). *Satisfaction* of a formula F by a stack s and heap h is denoted $s, h \models F$ and defined by structural induction on F in Figure 1. Here $\llbracket P \rrbracket$ is as usual a component of the least fixed point of a monotone operator constructed from the inductive definition set Φ ; see [3, 4] for details. We say the *entailment* $F_1 \models F_2$ holds if, for all stacks $s \in S$ and heaps $h \in H$, we have $s, h \models F_1$ implies $s, h \models F_2$.

$s, h \models \top$	\Leftrightarrow	always
$s, h \models \perp$	\Leftrightarrow	never
$s, h \models x = y$	\Leftrightarrow	$s(x) = s(y)$
$s, h \models x \neq y$	\Leftrightarrow	$s(x) \neq s(y)$
$s, h \models \mathbf{emp}$	\Leftrightarrow	$h = e$
$s, h \models x \mapsto y$	\Leftrightarrow	$\mathbf{dom}(h) = \{s(x)\}$ and $h(s(x)) = s(y)$
$s, h \models x \xrightarrow{2} y, z$	\Leftrightarrow	$\mathbf{dom}(h) = \{s(x)\}$ and $h(s(x)) = (s(y), s(z))$
$s, h \models P\mathbf{x}$	\Leftrightarrow	$(s(\mathbf{x}), h) \in \llbracket P \rrbracket$
$s, h \models F_1 \vee F_2$	\Leftrightarrow	$s, h \models F_1$ or $s, h \models F_2$
$s, h \models F_1 * F_2$	\Leftrightarrow	$\exists h_1, h_2 \in H. h = h_1 \circ h_2$ and $s, h_1 \models F_1$ and $s, h_2 \models F_2$

Fig. 1. Semantics of formulae

3 Cyclic proofs of separation logic entailments

In this section we define a proof system for a class of separation logic entailment problems. Our system employs *cyclic proof* as a means of tackling proofs involving inductively defined predicates.

Our proof system employs *sequents* of the form $F \vdash G$ where F and G are separation logic formulas as given by Defn. 1. We write $F[\theta]$ for the result of applying a substitution $\theta : \mathcal{V} \rightarrow \mathcal{V}$ to the formula F , and extend substitution pointwise to tuples of variables. We give a set of basic proof rules for sequents in Figure 2. Note that we write a rule with a double-line between premise and conclusion to indicate that it is *bidirectional*, i.e., that the rôles of premise and conclusion may be reversed. We also comment that our rules have been chosen for simplicity and ease of implementation, rather than completeness and expressivity. In particular, there is no rule for rewriting with equalities; such rewriting techniques are out of the scope of the present paper, which concentrates on inductive techniques.

To the proof rules in Figure 2 we add simple *unfolding* rules for the inductive predicates in the definition set Φ . In order to do this, it is essential to know which variables occur free in our inductive rules, so that they can be instantiated correctly in these proof rules. We write an annotated inductive rule $F \xrightarrow{\mathbf{z}} P\mathbf{x}$, where \mathbf{z} is a tuple of distinct variables, to indicate that $FV(F) \cup FV(P\mathbf{x}) = \{\mathbf{z}\}$.

Definition 3 (Unfolding rules). To any predicate symbol P we associate a finite number of *right-unfolding* rules and a single *left-unfolding* rule, constructed from its inductive definition in the inductive rule set Φ . First, for each inductive rule $F \xrightarrow{\mathbf{z}} P\mathbf{x}$ there is a right-unfolding rule for P :

$$\frac{G \vdash H * F[\mathbf{y}/\mathbf{z}]}{G \vdash H * P\mathbf{x}[\mathbf{y}/\mathbf{z}]} (PR)$$

(Note that, by definition, the variables occurring in the tuple \mathbf{x} all occur in \mathbf{z} , so that in e.g. $P\mathbf{x}[\mathbf{y}/\mathbf{z}]$ all of the variables in \mathbf{x} are uniformly replaced by arbitrary variables from \mathbf{y} .)

$$\begin{array}{c}
\frac{}{F \vdash F} \text{(Id)} \quad \frac{}{\perp * F \vdash G} \text{(}\perp\text{L)} \quad \frac{}{F \vdash \top} \text{(}\top\text{R)} \quad \frac{}{F \vdash x = x} \text{(}=\text{R)} \\
\\
\frac{}{x = y * x \neq y * F \vdash G} \text{(}=\text{L)} \quad \frac{}{x \mapsto y * x \mapsto z * F \vdash G} \text{(}\mapsto\text{)} \\
\\
\frac{}{x \xrightarrow{2} y_1, y_2 * x \xrightarrow{2} z_1, z_2 * F \vdash G} \text{(}\xrightarrow{2}\text{)} \quad \frac{F \vdash H \quad H \vdash G}{F \vdash G} \text{(Cut)} \\
\\
\frac{F \vdash G}{\mathbf{emp} * F \vdash G} \text{(empL)} \quad \frac{F \vdash G}{F \vdash G * \mathbf{emp}} \text{(empR)} \quad \frac{F_1 \vdash G_1 \quad F_2 \vdash G_2}{F_1 * F_2 \vdash G_1 * G_2} \text{(}*\text{)} \\
\\
\frac{F_1 * F \vdash G \quad F_2 * F \vdash G}{(F_1 \vee F_2) * F \vdash G} \text{(}\vee\text{L)} \quad \frac{F \vdash G_i * G}{F \vdash (G_1 \vee G_2) * G} \quad i \in \{1, 2\} \text{(}\vee\text{R)}
\end{array}$$

Fig. 2. Basic proof rules.

The left-unfolding, or *case-split* rule for P has the following general schema:

$$\frac{\text{case premises}}{G * P\mathbf{v} \vdash H} \text{(Case } P\text{)}$$

where, for each inductive rule of the form $F \xrightarrow{\mathbf{z}} P\mathbf{x}$, there is a *case premise*:

$$G[(\mathbf{x}[\mathbf{y}/\mathbf{z}])/\mathbf{v}] * F[\mathbf{y}/\mathbf{z}] \vdash H[(\mathbf{x}[\mathbf{y}/\mathbf{z}])/\mathbf{v}]$$

where the variables \mathbf{y} are *fresh*, i.e. $y \notin FV(G * P\mathbf{v} \vdash H)$ for all $y \in \{\mathbf{y}\}$. We observe that the complicated-seeming variable instantiation here essentially works in two stages. First, the variables \mathbf{z} appearing in the inductive rule $F \xrightarrow{\mathbf{z}} P\mathbf{x}$ are replaced by the fresh variables \mathbf{y} , giving us a “fresh version” of the rule, $F[\mathbf{y}/\mathbf{z}] \xrightarrow{\mathbf{y}} P\mathbf{x}[\mathbf{y}/\mathbf{z}]$. Second, to obtain the case premise we uniformly replace the variables \mathbf{v} appearing in the formula to be unfolded, $P\mathbf{v}$, with the freshly instantiated variables $\mathbf{x}[\mathbf{y}/\mathbf{z}]$ appearing in the conclusion of the inductive rule⁴.

Example 1 (List segment). Define the inductive predicate \mathbf{ls} by:

$$\begin{array}{l}
\mathbf{emp} \Rightarrow \mathbf{ls} \ x \ x \\
x \mapsto x' * \mathbf{ls} \ x' \ z \Rightarrow \mathbf{ls} \ x \ z
\end{array}$$

The formula $\mathbf{ls} \ x \ y$ denotes a singly-linked list segment whose first cell is pointed to by x and whose last cell contains y . The right-unfolding rules for \mathbf{ls} are:

$$\frac{G \vdash H * \mathbf{emp}}{G \vdash H * \mathbf{ls} \ y \ y} \text{(lsR}_1\text{)} \quad \frac{G \vdash H * y \mapsto y' * \mathbf{ls} \ y' \ v}{G \vdash H * \mathbf{ls} \ y \ v} \text{(lsR}_2\text{)}$$

⁴ We could write this premise more simply as $G * \mathbf{v} = \mathbf{x}[\mathbf{y}/\mathbf{z}] * F[\mathbf{y}/\mathbf{z}] \vdash H$. However, our formulation above allows us to do without rules for equality on the left.

The case-split rule for **1s** is:

$$\frac{G[y/v, y/v'] * \mathbf{emp} \vdash H[y/v, y/v'] \quad G[y/v, y'/v'] * y \mapsto y'' * \mathbf{1s} y'' y' \vdash H[y/v, y'/v']}{G * \mathbf{1s} v v' \vdash H} \text{ (Case } \mathbf{1s})$$

where y, y', y'' are suitably fresh. (Note that both v and v' are replaced by the same variable y in the first premise.)

Example 2 (Binary trees). Define the inductive predicate **btr** by:

$$\begin{aligned} & \mathbf{emp} \Rightarrow \mathbf{btr}(x) \\ x \xrightarrow{2} y, z * \mathbf{btr}(y) * \mathbf{btr}(z) & \Rightarrow \mathbf{btr}(x) \end{aligned}$$

The formula $\mathbf{btr}(x)$ denotes a binary tree whose first cell is pointed to by x . The right-unfolding rules for **btree** are:

$$\frac{G \vdash H * \mathbf{emp}}{G \vdash H * \mathbf{btr}(v)} \text{ (btr}R_1) \quad \frac{G \vdash H * v \xrightarrow{2} v_1, v_2 * \mathbf{btr}(v_1) * \mathbf{btr}(v_2)}{G \vdash H * \mathbf{btr}(v)} \text{ (btr}R_2)$$

The case-split rule for **btr** (where y, y_1, y_2 are suitably fresh) is:

$$\frac{G[y/v] * \mathbf{emp} \vdash H[y/v] \quad G[y/v] * y \xrightarrow{2} y_1, y_2 * \mathbf{btr}(y_1) * \mathbf{btr}(y_2) \vdash H[y/v]}{G * \mathbf{btr}(v) \vdash H} \text{ (Case } \mathbf{btr})$$

Our proof system allows proofs to be *cyclic*: that is, our proofs are derivation trees with “back edges”, subject to a global syntactic condition ensuring soundness. The following definitions are adapted from, but somewhat stronger than, their analogues in [3].

Definition 4 (Pre-proof). A *bud* in a derivation tree \mathcal{D} is a sequent occurrence in \mathcal{D} which is not the conclusion of any proof rule. A *companion* C for a bud B is any (strict) ancestor of B in \mathcal{D} of which B is a substitution instance, i.e. $C = B[\theta]$ for some θ . A *pre-proof* of a sequent S is given by $(\mathcal{D}, \mathcal{R})$, where \mathcal{D} is a derivation tree whose root is S and \mathcal{R} is a function assigning a companion to every bud of \mathcal{D} . The *graph* $\mathcal{G}(\mathcal{P})$ of a pre-proof \mathcal{P} is the (possibly cyclic) graph obtained from \mathcal{D} by adding an edge from each bud to its companion.

The restriction that buds must be assigned ancestor nodes as companions is for convenience of implementation. It is shown in [2] that this restriction does not reduce the power of the system, although it may lead to bigger proofs.

A *path* in a pre-proof graph is a sequence of sequent occurrences $(F_i \vdash G_i)_{i \geq 0}$ such that, for all $i \geq 0$, it holds that $F_{i+1} \vdash G_{i+1}$ is a premise of the rule instance in \mathcal{D} with conclusion $F_i \vdash G_i$.

Definition 5 (Trace). Let $(F_i \vdash G_i)_{i \geq 0}$ be a path in the graph of a pre-proof \mathcal{P} . A *trace following* $(F_i \vdash G_i)_{i \geq 0}$ is a sequence $(A_i)_{i \geq 0}$ such that, for all $i \geq 0$, A_i is a subformula occurrence of the form $P\mathbf{x}$ in the formula F_i , and either:

- (i) A_{i+1} is the subformula occurrence in F_{i+1} corresponding to A_i in F_i (defined in the obvious way analogous to [3, 4]), or
- (ii) $F_i \vdash G_i$ is the conclusion of an instance of a case-split rule (Case P), A_i is the formula $P\mathbf{v}$ unfolded by the rule and A_{i+1} is a subformula of the formula $F[\mathbf{y}/\mathbf{z}]$ obtained by the unfolding, in which case i is said to be a *progress point* of the trace.

We remark that, in particular, condition (i) means that formulas can only be traced through the left-hand premise of an instance of (Cut) and not its right-hand premise. An *infinitely progressing trace* is a (necessarily infinite) trace having infinitely many progress points.

Definition 6 (Cyclic proof). A pre-proof \mathcal{P} is a *cyclic proof* if it satisfies the *global trace condition*: for every infinite path $(F_i \vdash G_i)_{i \geq 0}$ in $\mathcal{G}(\mathcal{P})$, there is an infinitely progressing trace following some tail $(F_i \vdash G_i)_{i \geq n}$ of the path.

Theorem 7 (Soundness). *If there is a cyclic proof of $F \vdash G$, then $F \models G$.*

Proof. (Sketch) The proof runs along the lines given in [3, 6, 4]. Briefly, suppose for contradiction that there is a cyclic proof \mathcal{P} of $F \vdash G$ but $F \not\models G$, so that for some stack s and heap h we have $s, h \models F$ but $s, h \not\models G$. Then, by local soundness of the proof rules, we would be able to construct an infinite path $(F_i \vdash G_i)_{i \geq 0}$ in $\mathcal{G}(\mathcal{P})$ (with $F_0 \vdash G_0 = F \vdash G$) such that $F_i \not\models G_i$ for all $i \geq 0$. Since \mathcal{P} is a cyclic proof, there exists an $n \geq 0$ and an infinitely progressing trace following $(F_i \vdash G_i)_{i \geq n}$. It is a standard fact that the least fixed point interpretation of the inductive predicates can be generated by an ordinal-indexed chain of *approximants* (cf. [1]). The fact, guaranteed by the trace condition, that some occurrence of an inductive predicate is unfolded infinitely often using the case-split rules then induces an infinite decreasing chain of the ordinals indexing this chain of approximants, which contradicts their well-foundedness. \square

Example 3 (cf. [3]). The following is a cyclic pre-proof of $\mathbf{ls} \ x \ x' * \mathbf{ls} \ x' \ y \vdash \mathbf{ls} \ x \ y$.

$$\begin{array}{c}
\frac{}{\mathbf{ls} \ x \ y \vdash \mathbf{ls} \ x \ y} \text{(Id)} \quad \frac{}{x \mapsto z \vdash x \mapsto z} \text{(Id)} \quad \frac{}{\mathbf{ls} \ z \ x' * \mathbf{ls} \ x' \ y \vdash \mathbf{ls} \ z \ y} \text{(\dagger)} \quad \frac{}{\mathbf{ls} \ z \ x' * \mathbf{ls} \ x' \ y \vdash \mathbf{ls} \ z \ y} \text{(*)} \\
\frac{}{\mathbf{ls} \ x \ y \vdash \mathbf{ls} \ x \ y} \text{(empL)} \quad \frac{x \mapsto z * \mathbf{ls} \ z \ x' * \mathbf{ls} \ x' \ y \vdash x \mapsto z * \mathbf{ls} \ z \ y}{x \mapsto z * \mathbf{ls} \ z \ x' * \mathbf{ls} \ x' \ y \vdash \mathbf{ls} \ x \ y} \text{(lsR}_2\text{)} \\
\frac{\mathbf{emp} * \mathbf{ls} \ x \ y \vdash \mathbf{ls} \ x \ y \quad x \mapsto z * \mathbf{ls} \ z \ x' * \mathbf{ls} \ x' \ y \vdash \mathbf{ls} \ x \ y}{\mathbf{ls} \ x \ x' * \mathbf{ls} \ x' \ y \vdash \mathbf{ls} \ x \ y} \text{(Case ls)}
\end{array}$$

The pairing of a suitable companion with the only bud in this pre-proof is denoted by (\dagger) . A trace from the companion to the bud is denoted by the underlined formulas, with a progress point at the displayed application of (Case \mathbf{ls}).

We remark that the standard inductive proof of $\mathbf{ls} \ x \ x' * \mathbf{ls} \ x' \ y \vdash \mathbf{ls} \ x \ y$ is by induction on $\mathbf{ls} \ x \ x'$ using the induction hypothesis $\mathbf{ls} \ x' \ y \multimap \mathbf{ls} \ x \ y$, where \multimap is the multiplicative implication of separation logic. Not only is this induction hypothesis not a subformula occurring in the goal sequent, but it is not even expressible in our formula language (or in most available verification tools).

4 Implementation of the cyclic prover

The proof system described in Section 3 has been implemented in HOL Light as a *deep embedding*, meaning that proofs as well as sequents are represented explicitly in our implementation. Thus we provide HOL datatypes for formulas (with a sequent being represented as a pair of formulas) and pre-proofs, with the proof rules captured by a HOL relation on sequents.

The main obstacles when implementing a cyclic prover all stem from the fact that the activities of constructing cycles and verifying the soundness condition are *global* in nature, whereas (like most theorem provers) HOL Light’s internal view of proofs is inherently *local*. Thus, while one can typically implement a proof system simply by encoding each proof rule, we have to explicitly represent (portions of) pre-proofs in order to allow us to identify suitable companions for buds and to ensure that the resulting pre-proof satisfies the soundness condition that all infinite paths have infinitely progressing traces .

Our solution is to first tag each occurrence of an inductive predicate in our sequents, in order to assist in the construction of traces. We then augment each node with information about the current branch and any progress points in the traces along it. This gives us enough explicit information in a proof tree to enable the formation of “downlinks” from buds to companions, and to ensure the soundness condition on cyclic proofs.

The next subsections describe the various components of the implementation.

4.1 Representation of pre-proofs

As with the proof system in Section 3, the entire implementation is parameterized by a set of inductive definitions, so an OCaml datatype for inductive definitions has been designed and a list of such is a parameter to the whole implementation.

The type `formula` is implemented as in Definition 1 except for atomic formulas of the form $P\mathbf{x}$, which have the constructor `Ind num inductive`. The type `inductive` is generated from the input list of inductive definitions and simply has an entry $P\mathbf{x}$ for each inductive predicate. The argument of type `num` is a tag used to track occurrences so that traces can be established; in the following we write atomic formulas $P_i\mathbf{x}$ with the subscript i denoting the tag. Section 4.2 shows how traces are constructed from tags. When searching for a cyclic proofs, unique tags are assigned to all inductive predicates of the root node.

In the implemented system, our sequents are augmented with two extra pieces of information indicated by α and π . The component α is called the *ancestry* of the current node. It records the entire branch from the root of the proof tree to the node, in the form of a (finite) list of entailments $F_1 \vdash G_1, \dots, F_n \vdash G_n$, with $F_n \vdash G_n$ being the root of the tree. We write $F \vdash G :: \alpha$ for the ancestry obtained by adding $F \vdash G$ to the beginning of the list α , and write α_n for the n -th element of α (if it exists). The component $\pi \in \mathbb{N}$ is called the *progress pointer*. It is the smallest natural number n such that α_n is the conclusion of a case-split rule (denoting the closest progress point in the sense of Definition 5). If no such number exists (no progress happens on the current branch), the progress pointer

is $|\alpha| + 1$, that is, it points past the end of the ancestry. Thus the nodes of the proof tree are augmented sequents, written as

$$(\alpha, \pi) : F \vdash G$$

It is relatively easy to track both ancestries and progress pointers syntactically, and the implemented rules, with two exceptions, are just the ones from Figure 2, augmented with ancestry and progress pointers. The exceptions are the rule (Id) and a link formation rule which we describe below.

The ancestry is quite simple, the proven sequent is just tacked on when moving from the conclusion up to the premise of the rule. The progress pointer is incremented in all rules that do not make progress, because the distance from the current node to the nearest progress point has increased by one (again, reading the rule from conclusion to premise). For a rule that does make progress, the progress pointer is set to 1, meaning that a downlink has to point at least 1 step beyond this rule. So, for example, the rule (**empR**) now looks like this

$$\frac{((F \vdash G * \mathbf{emp}) :: \alpha, \pi + 1) : F \vdash G}{(\alpha, \pi) : F \vdash G * \mathbf{emp}} \text{ (c_emp_R)}$$

in one direction while the rule (**1s-Case**) looks like this:

$$\frac{\begin{array}{l} ((G * \mathbf{1s}_i v v' \vdash H) :: \alpha, 1) : G[y/v, y/v'] * \mathbf{emp} \vdash H[y/v, y/v'] \\ ((G * \mathbf{1s}_i v v' \vdash H) :: \alpha, 1) : G[y/v, y'/v'] * y \mapsto y'' * \mathbf{1s}_i y'' y' \vdash H[y/v, y'/v'] \end{array}}{(\alpha, \pi) : G * \mathbf{1s}_i v v' \vdash H} \text{ (c_1s_Case)}$$

Note that the formula $\mathbf{1s} y'' y'$ in the second premise, obtained by unfolding $\mathbf{1s} v v'$ in the conclusion, inherits the tag i , in keeping with the rules for forming traces in Defn. 5. Also the progress pointer in both premises is reset to 1.

The rule (Id) is slightly different because it has to account for the tags on inductive predicates. Since tags are only relevant for the purpose of constructing traces, they should be ignored when applying (Id). We define a binary predicate **matches** on formulas to implement equality up to change of tags, whence $F \text{ matches } G$ holds if F and G are equal when all their tags are erased. The implemented form of (Id) is then

$$\frac{F \text{ matches } F'}{(\alpha, \pi) : F \vdash F'} \text{ (c_Id)}$$

The ancestry information alone is enough to form cycles, but the progress pointer allows us to only form cycles which contains at least one progress point: In order to find a companion for $(\alpha, \pi) : F \vdash G$, it suffices to find a substitution θ and an n such that $n > \pi$, α_n is defined and $\alpha_n = (F \vdash G)[\theta]$. However, because traces only involve predicates occurring on the left of sequents, it suffices that G and the right hand side of α_n are equal up to predicate tags. Thus, the proof rule for link formation in the implemented system is

$$\frac{|\alpha| > n > \pi \quad \exists \theta. \alpha_n = (F \vdash G')[\theta] \quad G \text{ matches } G'[\theta]}{(\alpha, \pi) : F \vdash G} \text{ (c_downlink)}$$

where $|\alpha|$ is the length of the ancestry.

4.2 Soundness of the implementation

We now describe how the soundness of the implemented system follows from the soundness of the system in Section 3.

There is a map \mathcal{E} from proofs in the implemented system to pre-proofs in the system from Section 3 in the sense of Definition 4. That is, for any proof tree T in the implemented system, $\mathcal{E}(T) = (\mathcal{D}, \mathcal{R})$, where:

- \mathcal{D} is the derivation tree (in the proof system of section 3) obtained by stripping the ancestry and progress pointer from each node of T and turning every node occurring as the conclusion of an instance of `(c_downlink)` into a bud of \mathcal{D} ;
- \mathcal{R} is a function from the buds of \mathcal{D} to suitable companions, built from the applications of `(c_downlink)` in the obvious way.

The main theorem of this section is that for every proof P in the implemented system, the pre-proof $\mathcal{E}(P)$ is actually a cyclic proof:

Theorem 8 (Soundness of the implementation). *If there is a proof of $([], 1) : F \vdash G$ in the implemented system, then $F \models G$.*

Proof. (Sketch) Given a proof P of $([], 1) : F \vdash G$, we show that $\mathcal{E}(P) = (\mathcal{D}, \mathcal{R})$ is a cyclic proof. $\mathcal{E}(P)$ is clearly a pre-proof by construction, so it just remains to show that it satisfies the global soundness condition of Defn. 6. Essentially, the argument is that our tagging of inductive predicates and the conditions on the “downlink” rule `(c_downlink)` ensure that there is a “trace manifold” for $\mathcal{E}(P)$, which implies the global soundness condition (see [2], ch. 7).

Let $(S_i)_{i \geq 0}$ be an infinite path in the graph $\mathcal{G}(\mathcal{E}(P))$. It is clear that there exists a tail $(S_i)_{i \geq n}$ of this path that traverses some strongly connected component \mathcal{C} of $\mathcal{G}(\mathcal{E}(P))$, which must be constructed from finite paths of the form $\mathcal{R}(B) \dots B$ from companions to buds. Specifically, there is a non-empty (finite) set \mathbf{B} of buds which are visited infinitely often on $(S_i)_{i \geq n}$. Let B be the bud in \mathbf{B} with the lowest companion, i.e. such that $\mathcal{R}(B)$ is closest to the root of \mathcal{D} . By inspection of the `(c_downlink)` rule, there is some tagged atomic formula $P_i \mathbf{x}$ occurring in both $\mathcal{R}(B)$ and B whose case-split rule is applied on the path $\mathcal{R}(B) \dots B$. There must be an infinitely progressing trace following $(S_i)_{i \geq n}$, with all predicates tagged by i . A trace must exist because all tags on the left of sequents must be identical to apply `(c_downlink)` and our tagging discipline for other rules follows the method for constructing traces in Defn. 5. Moreover, this trace is infinitely progressing because our choice of $\mathcal{R}(B)$ to be the lowermost companion in \mathcal{C} visited infinitely often ensures that the path $(S_i)_{i \geq n}$ passes through the case-split rule that unrolls a predicate tagged by i . \square

4.3 Proof Search

Split entailments. To better manage the sizes of the generated proofs, the entailment relation has been split into two: the augmented entailment $(\alpha, \pi) : P \vdash Q$ and a basic one $P \vdash_{basic} Q$ which is not augmented (and so \vdash_{basic} is actually a subset of \vdash). The idea is to relay all reasoning using the associativity, commutativity and unit of $*$ to \vdash_{basic} . Such rules as (**empR**) are then found in this lightweight entailment rather than in the augmented one.

For the augmented entailment to make use of (**empR**), cut rules to inject \vdash_{basic} -reasoning into proofs are provided:

$$\frac{P \vdash_{basic} R \quad (\alpha, \pi) : R \vdash Q}{(\alpha, \pi) : P \vdash Q} \text{ (basicL)} \quad \frac{(\alpha, \pi) : P \vdash R \quad R \vdash_{basic} Q}{(\alpha, \pi) : P \vdash Q} \text{ (basicR)}$$

It is important that \vdash_{basic} does not destroy the tracings, and so it is limited to reorganizing terms. Its id-rule, for instance, does not use the **matches**-predicate, and there is no cut rule. It can be shown that this careful re-factoring of the entailment relation does not change the truth of Theorem 8.

Tactics. The prover is a collection of HOL tactics. There is a tactic for each rule of the implemented proof system; tactics are generated for the unfolding rules given by the inductive definitions. The left rules introduce fresh variables and perform the (potentially unifying) substitutions, while the right rules introduce existential metavariables for any extra exposed variables.

Since a rule might not be directly applicable until some rearrangements have been performed, a layer of intelligent rule applications is built upon these, using \vdash_{basic} -reasoning to set up rule applications. For the right rules, this amounts to bringing the conclusion to the front on the right hand side.

On this, a layer of “active” rule applications is constructed. Right rules, for instance, typically expose new state on the right side. An active version of the right rule, will try to match this on the left hand side (resolving existential metavariables if necessary) and invoke a tactic to eliminate common state. The active rule will fail, if no state can be disposed of. Elimination of common state is implemented using \vdash_{basic} -reasoning to bring both sides to similar forms and then using the rules ($*$) and a version of (**c_Id**) which resolves existential metavariables. Finally, a rule for link formation is implemented that searches through the ancestry for sequents of which the current node is a substitution instance and if one is found, applies (**c_downlink**).

With these basic tactics at hand, one can conveniently use the system interactively or implement an automatic tactic. We implemented a backtracking proof search which applies any rule it can, with two constraints: Certain rules, like right unfolding rules, have to be active and the rules are organized into groups with different priorities:

1. $\{(\mathbf{c_Id}), (\mathbf{c_downlink})\}$
2. right rules
3. left rules

Rules not in these groups are only invoked as part of auxiliary reasoning for the rules in these groups.

4.4 Experiments and Performance

Table 2 reports several lemmas that has been proven automatically by our cyclic prover, while Table 1 shows the definitions of the inductive predicates appearing in Table 2. The implementation was tested on a MacBook with a 2.4 GHz Intel Core Duo and 2 GB of 667 MHz DDR2 SDRAM running Mac OS 10.5.8.

We also proved a more sophisticated lemma interactively. It makes use of lemma 3 from Table 2:

Example 4. The following is a cyclic pre-proof of $\mathbf{RList} \ x \ r \vdash \mathbf{List} \ x \ y$.

$$\frac{\frac{\frac{}{x \mapsto y \vdash x \mapsto y} \text{(Id)}}{x \mapsto y \vdash \mathbf{List} \ x \ y} \text{(ListR}_1\text{)} \quad \frac{\frac{\frac{}{z \mapsto y \vdash z \mapsto y} \text{(Id)}}{z \mapsto y * \mathbf{RList} \ x \ z \vdash z \mapsto y * \mathbf{List} \ x \ z} \text{(*)}}{z \mapsto y * \mathbf{RList} \ x \ z \vdash \mathbf{List} \ x \ y} \text{lemma 3}}{z \mapsto y * \mathbf{RList} \ x \ z \vdash \mathbf{List} \ x \ y} \text{(Cut)}}{\frac{}{\mathbf{RList} \ x \ y \vdash \mathbf{List} \ x \ y} \text{(†)}} \text{(Case rls)}$$

It seems that a static analyzer could benefit from remembering earlier proven lemmas and allow the automatic tactic to use these.

Most of the lemmas in Table 2 were proven with a bound of 3 on the depth of backtracking. Lemmas 10 through 12 required higher bounds, due to the mutual recursion (5, 7 and 5 respectively), and a few of the tree lemmas required a bound of 4 (lemmas 13, 14 and 16). The relatively low bound needed to prove lemmas is due to the split entailment relations.

5 Related work

There is a good body of work in the literature which relates to ours in several aspects.

Tuerk developed Holfoot [19] a general framework for Separation Logic in HOL. This general system has been shown useful in proving several interesting pointer manipulating programs in an automatic way. Holfoot does not support cyclic proofs, and we hope that our work may become useful for bringing cyclic proofs in such general framework.

Nguyen *et al.* [16] describe an extension of entailment checking technique which employs fold/unfold mechanism for user defined inductive predicated introduced in earlier work [17]. This extension is a mechanism to automatically prove and apply lemmas provided by the user. When proving the lemmas, the theorem prover may apply the lemma itself. The recursive application of the lemmas is done on the root node of inductive predicates. Therefore, this method represents a simple kind of cycles tailored to their specific verification system. While the emphasis of that paper is in the application of lemmas, the emphasis of our work is rather on the definition and implementation of cyclic proof as well as in proving the soundness of our system. Here we have focussed at developing a general cyclic entailment checker which could eventually become a off-the-shelf

Predicate	Definition
RList (nonempty list segment)	$x \mapsto y \Rightarrow \text{RList } x \ y$ $\text{RList } x \ y * y \mapsto z \Rightarrow \text{RList } x \ z$
List (nonempty list segment)	$x \mapsto z \Rightarrow \text{List } x \ z$ $\text{List } z \ y * x \mapsto z \Rightarrow \text{List } x \ y$
ListE / ListO (nonempty list segment of even / odd length)	$x \mapsto z \Rightarrow \text{ListO } x \ z$ $\text{ListO } z \ y * x \mapsto z \Rightarrow \text{ListE } x \ y$ $\text{ListE } z \ y * x \mapsto z \Rightarrow \text{ListO } x \ y$
PeList (list segment)	$\text{Emp} \Rightarrow \text{PeList } x \ x$ $\text{PeList } z \ y * x \mapsto z \Rightarrow \text{PeList } x \ y$
DLL (doubly linked list segment)	$\text{Emp} \Rightarrow \text{DLL } a \ a \ b \ b$ $\text{DLL } x \ b \ c \ a * a \xrightarrow{2} x, d \Rightarrow \text{DLL } a \ b \ c \ d$
SLL (singly linked list segment in binary heap)	$\text{Emp} \Rightarrow \text{SLL } a \ a$ $\text{SLL } x \ b * a \xrightarrow{2} x, d \Rightarrow \text{SLL } a \ b$
BSLL (reverse SLL)	$\text{Emp} \Rightarrow \text{BSLL } c \ c$ $\text{BSLL } c \ x * x \xrightarrow{2} a, d \Rightarrow \text{BSLL } c \ d$
BinTree (binary tree)	$\text{Emp} \Rightarrow \text{BinTree } a$ $\text{BinTree } b * \text{BinTree } c * a \xrightarrow{2} b, c \Rightarrow \text{BinTree } a$
BinTreeSeg (binary tree segment)	$\text{Emp} \Rightarrow \text{BinTreeSeg } a \ a$ $\text{BinTreeSeg } c \ b * \text{BinTree } d * a \xrightarrow{2} c, d \Rightarrow \text{BinTreeSeg } a \ b$ $\text{BinTree } c * \text{BinTreeSeg } d \ b * a \xrightarrow{2} c, d \Rightarrow \text{BinTreeSeg } a \ b$
BinListFirst (list in cell 1 of binary heap)	$\text{Emp} \Rightarrow \text{BinListFirst } a$ $\text{BinListFirst } b * a \xrightarrow{2} b, c \Rightarrow \text{BinListFirst } a$
BinListSecond (list in cell 2 of binary heap)	$\text{Emp} \Rightarrow \text{BinListSecond } a$ $\text{BinListSecond } c * a \xrightarrow{2} b, c \Rightarrow \text{BinListSecond } a$
BinPath (path in binary heap)	$\text{Emp} \Rightarrow \text{BinPath } a \ a$ $\text{BinPath } c \ b * a \xrightarrow{2} c, d \Rightarrow \text{BinPath } a \ b$ $\text{BinPath } c \ b * a \xrightarrow{2} d, c \Rightarrow \text{BinPath } a \ b$

Table 1. Definitions of predicates

prover for verification tools or theorem provers. The parametric character gives the flexibility to easily tune the expressivity w.r.t. speed.

Chang *et al.* [11, 10] propose a shape analysis guided by data structure invariants, called invariant checkers, provided by the programmer. Invariant checkers describe inductive predicates. While their emphasis is on defining expressive and precise shape analyses for a large variety of data structures, our emphasis here is on solving entailment questions which could be used to help such kind of analyses. Therefore, our automated cyclic proofs could be used to support or enhance various operations performed in their shape analysis (such as, approximation test, helping termination of fixed point computation, widening, etc.)

Recently, Voicu and Li [20] have implemented a technique for cyclic proofs in the Coq proof assistant. Their system seems to be oriented towards inductive lemmas for arithmetic, whereas ours is specialized in separation logic, and there-

Lemma nr	Time in seconds	Proven
1	2.37	$x \mapsto y^* \text{RList } y \ z \vdash \text{RList } x \ z$
2	2.37	$\text{RList } x \ z^* \text{RList } z \ y \vdash \text{RList } x \ y$
3	2.56	$z \mapsto y^* \text{List } x \ z \vdash \text{List } x \ y$
4	2.45	$\text{List } z \ y^* \text{List } x \ z \vdash \text{List } x \ y$
5	2.78	$z \mapsto y^* \text{PeList } x \ z \vdash \text{PeList } x \ y$
6	1.96	$\text{PeList } z \ y^* \text{PeList } x \ z \vdash \text{PeList } x \ y$
7	3.54	$\text{DLL } u \ v \ x \ y \vdash \text{SLL } u \ v$
8	3.82	$\text{DLL } u \ v \ x \ y \vdash \text{BSLL } x \ y$
9	8.86	$\text{DLL } w \ v \ x \ z^* \text{DLL } u \ w \ z \ y \vdash \text{DLL } u \ v \ x \ y$
10	5.44	$\text{ListO } z \ y^* \text{ListO } x \ z \vdash \text{ListE } x \ y$
11	11.2	$\text{ListE } x \ z^* \text{ListE } z \ y \vdash \text{ListE } x \ y$
12	5.57	$\text{ListO } z \ y^* \text{ListE } x \ z \vdash \text{ListO } x \ y$
13	4.40	$\text{BinListFirst } x \vdash \text{BinTree } x$
14	4.43	$\text{BinListSecond } x \vdash \text{BinTree } x$
15	4.21	$\text{BinPath } z \ y^* \text{BinPath } x \ z \vdash \text{BinPath } x \ y$
16	7.00	$\text{BinPath } x \ y \vdash \text{BinTreeSeg } x \ y$
17	8.78	$\text{BinTreeSeg } z \ y^* \text{BinTreeSeg } x \ z \vdash \text{BinTreeSeg } x \ y$
18	8.61	$\text{BinTreeSeg } x \ y^* \text{BinTree } y \vdash \text{BinTree } x$

Table 2. Experimental results.

fore, aims at proofs for complex dynamically-allocated data structures found in real software.

Finally, there is a large body of work on automated theorem proving using explicit induction; we mention IsaPlanner [14] as one contemporary such tool that employs Bundy’s *rippling* technique to remove differences between the hypotheses of an induction and its goal. We think it far from unlikely that these techniques might usefully transfer to the setting of cyclic proof.

6 Conclusion and Future Work

In this paper we have introduced a sound automatic entailment checking for separation logic with inductive predicates based on cyclic proofs. In our description we have focussed on the soundness of our method and in describing carefully the details of the implementation. Our procedure represents a relevant first step towards the construction of off-the-shelf theorem provers based for separation logic. Our entailment checker has been implemented in HOL Light and has shown great potential by proving non trivial lemmas of several kinds of inductive predicates corresponding to popular data structures.

The automatic entailment checking introduced in this paper opens up several direction and in the future we plan to enhance its expressivity and effectiveness looking at different possibilities. We intend to experiment with weakening the soundness condition to admit more sophisticated cyclic proofs using nested cycles. This will require also an improvement of the automated search by implementing more sophisticated search tactics. We intend to extend the expressivity

of formulae by adding constructs such as quantifiers and arithmetic operations. In doing this we will also investigate the possibility to integrate our procedure with STM solvers and arithmetic provers. Finally, we plan to explore how our prover could be integrated with automatic verification tools such as jStar [13].

Acknowledgments. We would like to thank Thomas Tuerk and Peter O’Hearn for intense helpful discussions. We thank Huu Hai Nguyen and Wei-Ngan Chin for providing us with example lemmas to add to our experiments. We acknowledge EPSRC and Royal Academy of Engineering for support.

References

1. P. Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pp. 739–782. North-Holland, 1977.
2. J. Brotherston. *Sequent Calculus Proof Systems for Inductive Definitions*. PhD thesis, University of Edinburgh, November 2006.
3. J. Brotherston. Formalised inductive reasoning in the logic of bunched implications. In *Proceedings of SAS-14*, volume 4634 of *LNCS*, pp. 87–103. Springer, 2007.
4. J. Brotherston, R. Bornat, and C. Calcagno. Cyclic proofs of program termination in separation logic. In *Proceedings of POPL-35*, pp. 101–112. ACM, 2008.
5. J. Brotherston and M. Kanovich. Undecidability of propositional separation logic and its neighbours. In *Proceedings of LICS-25*, pp. 137–146. IEEE, 2010.
6. J. Brotherston and A. Simpson. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 2010.
7. A. Bundy. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning*, volume I, chapter 13, pp. 845–911. Elsevier Science, 2001.
8. C. Calcagno, D. Distefano, P. O’Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of POPL-36*, pp. 289–300, 2009.
9. C. Calcagno, P. O’Hearn, and H. Yang. Local action and abstract separation logic. In *Proceedings of LICS-22*, pages 366–378. IEEE, 2007.
10. B.-Y. Evan Chang and X.Rival. Relational inductive shape analysis. In *POPL 2008*, pp. 247–260, 2008.
11. B.-Y. Evan Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In *SAS’07*, LNCS 4634, pp. 384–401, Springer 2007.
12. D. Distefano, P. O’Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS’06*, pp. 287–302, LNCS 3920, Springer 2006.
13. D. Distefano and M. Parkinson. jStar: Towards practical verification for Java. In *Proceedings of OOPSLA*, pp. 213–226. ACM, 2008.
14. L. Dixon and J. Fleuriot. Higher order rippling in isaplanner. In *Theorem Proving in Higher Order Logics ’04*, LNCS 3223. Springer, 2004.
15. J. Harrison. HOL Light: An overview. In *TPHOLs 2009*, LNCS 5674, pp. 60–66.
16. H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *Proceedings of CAV 2008*, LNCS 5123, pp. 355–369. Springer, 2008.
17. H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, pp. 251–266, Springer 2007.
18. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th LICS*, 2002.
19. T. Tuerk. A formalisation of smallfoot in hol. In *Theorem Proving in Higher Order Logics*, LNCS, pp. 469–484. Springer, 2009.
20. R. Voicu and M. Li. Descente infinie proofs in Coq. In *Proceedings of the 1st Coq Workshop*. Technische Universität München, 2009.
21. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O’Hearn. Scalable shape analysis for systems code. In *Proceedings of CAV*, 2008.