# Automated Cyclic Entailment Proofs in Separation Logic

James Brotherston[1], Dino Distefano[2], and Rasmus L. Petersen[2]

[1] Dept. of Computing, Imperial College London
[2] Dept. of Computer Science, Queen Mary University of London

**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 [19] 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: The formula $F * G$ denotes those heaps which can be separated into two disjoint parts satisfying $F$ and $G$, respectively. 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, 23, 13].

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

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

This rule might be applied during a proof search 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 [23]), 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 "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 formal 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 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. Our implementation approach should also transfer to other, similar cyclic proof systems as described in [2].

The remainder of this paper is structured as follows. Section 2 introduces 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:

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

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 write $FV(F)$ to denote the set of variables occurring in formula $F$. 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 ($-\!\!*$). These features are not typically employed in separation logic verification tools (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].

The definitions of our predicate symbols are supplied by "inductive rule sets" in the style of [3, 2], which are based on Martin-Löf's "productions" [16].

**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 $\Phi$.

*Semantics.* Let $L$ be an infinite set of *locations*, and $V$ be a set of *values*. Then $H = L \rightharpoonup_{\text{fin}} V$, the set of all finite partial functions from $L$ to $V$, is called the set of *heaps*. (We sometimes choose to work instead with heaps of the form $H = L \rightharpoonup_{\text{fin}} V \times V$, where a pair of values is stored at each location.) We write $\mathtt{dom}(h)$ 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.

The set of *stacks* is $S = \mathcal{V} \to L \cup V$, 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. There, $[\![P]\!]$ is as usual a component of the least fixed point of a monotone operator constructed from the inductive definition set $\Phi$; 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 $s, h \models F_2$.

$$
\begin{array}{rcl}
s,h \models \top & \Leftrightarrow & \text{always} \\
s,h \models \bot & \Leftrightarrow & \text{never} \\
s,h \models x = y & \Leftrightarrow & s(x) = s(y) \\
s,h \models x \neq y & \Leftrightarrow & s(x) \neq s(y) \\
s,h \models \mathbf{emp} & \Leftrightarrow & h = e \\
s,h \models x \mapsto y & \Leftrightarrow & \mathtt{dom}(h) = \{s(x)\} \text{ and } h(s(x)) = s(y) \\
s,h \models x \overset{2}{\mapsto} y,z & \Leftrightarrow & \mathtt{dom}(h) = \{s(x)\} \text{ and } h(s(x)) = (s(y), s(z)) \\
s,h \models P\mathbf{x} & \Leftrightarrow & (s(\mathbf{x}), h) \in [\![P]\!] \\
s,h \models F_1 \vee F_2 & \Leftrightarrow & s,h \models F_1 \text{ or } s,h \models F_2 \\
s,h \models F_1 * F_2 & \Leftrightarrow & \exists h_1, h_2 \in H.\ h = h_1 \circ h_2 \text{ and } s,h_1 \models F_1 \text{ and } s,h_2 \models F_2
\end{array}
$$

**Fig. 1.** Semantics of separation logic formulae. Note that the $\mapsto$ and $\overset{2}{\mapsto}$ predicates are interpreted only in heaps of type $L \rightharpoonup_{\text{fin}} V$ and $L \rightharpoonup_{\text{fin}} V \times V$ respectively.

## 3 Cyclic proofs of separation logic entailments

In this section we define a formal cyclic proof system for a class of separation logic entailment problems 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 $\theta : \mathcal{V} \to \mathcal{V}$ 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 the premise and conclusion are interchangeable (so that a "double-line rule" effectively abbreviates two normal rules). 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 $\Phi$. In order to formulate these, it is essential to know which variables occur free in our inductive rules, so that they can be instantiated correctly. We write an annotated inductive rule $F \overset{\mathbf{z}}{\Rightarrow} P\mathbf{x}$, where $\mathbf{z}$ is a tuple of distinct variables, to indicate that $FV(F) \cup \{\mathbf{x}\} = \{\mathbf{z}\}$.

**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 $\Phi$. First, for each inductive rule $F \overset{\mathbf{z}}{\Rightarrow} P\mathbf{x}$ there is a right-unfolding rule for $P$:

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

where $\mathbf{y}$ is any tuple of variables of the same length as $\mathbf{z}$. (Note that $\{\mathbf{x}\} \subseteq \{\mathbf{z}\}$ by definition, so that in $P\mathbf{x}[\mathbf{y}/\mathbf{z}]$ all of the variables in $\mathbf{x}$ are uniformly replaced by arbitrary variables from $\mathbf{y}$.)

$$\frac{}{F \vdash F}\,(\mathrm{Id}) \qquad \frac{}{\bot * F \vdash G}\,(\bot\mathrm{L}) \qquad \frac{}{F \vdash \top}\,(\top\mathrm{R}) \qquad \frac{}{F \vdash x = x}\,(=\mathrm{R})$$

$$\frac{}{x = y * x \neq y * F \vdash G}\,(=\mathrm{L}) \qquad \frac{}{x \mapsto y * x \mapsto z * F \vdash G}\,(\mapsto)$$

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

$$\frac{F \vdash G}{\mathbf{emp} * F \vdash G}\,(\mathbf{emp}\mathrm{L}) \qquad \frac{F \vdash G}{F \vdash G * \mathbf{emp}}\,(\mathbf{emp}\mathrm{R}) \qquad \frac{F_1 \vdash G_1 \quad F_2 \vdash G_2}{F_1 * F_2 \vdash G_1 * G_2}\,(*)$$

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

**Fig. 2.** Basic proof rules. A rule written with a double-line between premise and conclusion indicates that the premise and conclusion are interchangeable.

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

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

where, for each inductive rule of the form $F \overset{\mathbf{z}}{\Rightarrow} P\mathbf{x}$, 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. $y \notin FV(G * P\mathbf{v}) \cup FV(H)$ 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 $F \overset{\mathbf{z}}{\Rightarrow} P\mathbf{x}$ are replaced by the fresh variables $\mathbf{y}$, giving us a "fresh version" of the rule, $F[\mathbf{y}/\mathbf{z}] \overset{\mathbf{y}}{\Rightarrow} P\mathbf{x}[\mathbf{y}/\mathbf{z}]$. 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 rule[3].

*Example 1 (List segment).* Define the inductive predicate `ls` by:

$$\mathbf{emp} \overset{x}{\Rightarrow} \mathtt{ls}\,x\,x \qquad x \mapsto x' * \mathtt{ls}\,x'\,z \overset{x,x',z}{\Rightarrow} \mathtt{ls}\,x\,z$$

(Note the variable annotations.) The formula $\mathtt{ls}\,x\,y$ 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 `ls` are:

$$\frac{G \vdash H * \mathbf{emp}}{G \vdash H * \mathtt{ls}\,y\,y}\,(\mathtt{ls}R_1) \qquad \frac{G \vdash H * y \mapsto y' * \mathtt{ls}\,y'\,v}{G \vdash H * \mathtt{ls}\,y\,v}\,(\mathtt{ls}R_2)$$

[3] 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 $\texttt{ls}$ is:

$$\frac{\begin{array}{l} G[y/v, y/v'] * \mathbf{emp} \vdash H[y/v, y/v'] \\ G[y/v, y'/v'] * y \mapsto y'' * \texttt{ls}\, y''\, y' \vdash H[y/v, y'/v'] \end{array}}{G * \texttt{ls}\, v\, v' \vdash H} \text{(Case } \texttt{ls}\text{)}$$

where $y, y', y''$ are suitably fresh. Note that both $v$ and $v'$ are replaced by the *same* fresh variable $y$ in the first premise, because the corresponding inductive rule $\mathbf{emp} \overset{x}{\Rightarrow} \texttt{ls}\, x\, x$ only has a single free variable $x$.

*Example 2 (Binary trees).* Define the inductive predicate $\texttt{btr}$ by:

$$\mathbf{emp} \overset{x}{\Rightarrow} \texttt{btr}\, x \qquad x \overset{2}{\mapsto} y, z * \texttt{btr}\, y * \texttt{btr}\, z \overset{x,y,z}{\Rightarrow} \texttt{btr}\, x$$

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

$$\frac{G \vdash H * \mathbf{emp}}{G \vdash H * \texttt{btr}\, v} (\texttt{btr}R_1) \qquad \frac{G \vdash H * v \overset{2}{\mapsto} v_1, v_2 * \texttt{btr}\, v_1 * \texttt{btr}\, v_2}{G \vdash H * \texttt{btr}\, v} (\texttt{btr}R_2)$$

The case-split rule for $\texttt{btr}$ (where $y, y_1, y_2$ are suitably fresh) is:

$$\frac{\begin{array}{l} G[y/v] * \mathbf{emp} \vdash H[y/v] \\ G[y/v] * y \overset{2}{\mapsto} y_1, y_2 * \texttt{btr}\, y_1 * \texttt{btr}\, y_2 \vdash H[y/v] \end{array}}{G * \texttt{btr}\, v \vdash H} \text{(Case } \texttt{btr}\text{)}$$

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

**Definition 4 (Pre-proof).** A *bud* in a derivation tree $\mathcal{D}$ is a sequent occurrence in $\mathcal{D}$ to which no proof rule has been applied (i.e., it is not the conclusion of any proof rule instance in $\mathcal{D}$). A *companion* for a bud $B$ is a sequent occurrence $C$ in $\mathcal{D}$ of which $B$ is a substitution instance, i.e. $C = B[\theta]$ for some substitution $\theta$. 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 $\mathcal{D}$.

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

**Definition 5 (Trace).** Let $(F_i \vdash G_i)_{i \geq 0}$ be a path in 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 $\mathcal{P}$, 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 $\mathcal{P}$ (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. □

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

$$\dfrac{\dfrac{\dfrac{}{\mathtt{ls}\,x\,y \vdash \mathtt{ls}\,x\,y}\,(\mathrm{Id})}{\mathbf{emp} * \mathtt{ls}\,x\,y \vdash \mathtt{ls}\,x\,y}\,(\mathbf{emp}\mathrm{L}) \qquad \dfrac{\dfrac{\dfrac{}{x \mapsto z \vdash x \mapsto z}\,(\mathrm{Id}) \quad (\dagger)\ \underline{\mathtt{ls}\,z\,x'} * \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,z\,y}{x \mapsto z * \underline{\mathtt{ls}\,z\,x'} * \mathtt{ls}\,x'\,y \vdash x \mapsto z * \mathtt{ls}\,z\,y}\,(*)}{x \mapsto z * \underline{\mathtt{ls}\,z\,x'} * \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y}\,(\mathtt{ls}\mathrm{R}_2)}{(\dagger)\ \underline{\mathtt{ls}\,x\,x'} * \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y}\,(\mathrm{Case}\ \mathtt{ls})$$

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

We remark that the standard inductive proof of $\mathtt{ls}\,x\,x' * \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y$ is by induction on $\mathtt{ls}\,x\,x'$ using the induction hypothesis $\mathtt{ls}\,x'\,y \mathrel{-\!\!*} \mathtt{ls}\,x\,y$, where $\mathrel{-\!\!*}$ is the multiplicative implication of separation logic. Not only is this induction hypothesis not a subformula of the goal sequent, but it is not even expressible in our formula language (or that of 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* operations on proof trees, 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 a HOL datatype following Definition 1 except for atomic formulas of the form $P\mathbf{x}$, which have the constructor `Ind` : `num` $\rightarrow$ `inductive` $\rightarrow$ `formula`. The datatype `inductive` is generated from the input list of inductive definitions and simply has an entry $P\mathbf{x}$ for each inductive predicate. The argument to `Ind` of type `num` is a tag used to track occurrences so that traces can be established; Section 4.2 describes how traces are constructed using predicate tags. When searching for a cyclic proof, unique tags are assigned to all inductive predicates of the root node.

In the implemented system, the nodes of the proof tree are "augmented sequents" containing extra information about the proof tree, written as

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

(where $F$ and $G$ are formulas). The component $\alpha$ 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 $F_1 \vdash G_1, \ldots, F_n \vdash G_n$, 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 $\alpha$, and write $\alpha_n$ for the $n^{\text{th}}$ element of $\alpha$ (if it exists). The component $\pi \in \mathbb{N}$, called the *progress pointer*, is the smallest natural number $n$ such that $\alpha_n$ is the conclusion of a case-split rule (denoting the closest progress point below the current node in the sense of Definition 5). If no such $n$ exists (so that no case-split rules are applied below the current node), we set $\pi = |\alpha| + 1$, so that $\pi$ points past the end of the ancestry.

The general transformation from the rules of Figure 2 to rules using augmented sequents in the implemented system is the following:

$$\frac{S_1 \;\dots\; S_n}{S} \quad\Longrightarrow\quad \frac{(S :: \alpha, \pi+1) : S_1 \;\dots\; (S :: \alpha, \pi+1) : S_n}{(\alpha, \pi) : S}$$

I.e., when applying a rule backwards, the sequent in the conclusion of the rule is added to the ancestry of each of its premises. The progress pointer is incremented, because the distance from the current node to the nearest conclusion of a case-split rule has increased by one (reading the rule from conclusion to premise).

Naturally, the case-split rules are exceptions. When a case-split rule is applied, the progress pointer is set to 1 in each of its premises. So, for example, the implemented version of the case-split rule (Case `ls`) looks like this:

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

where the subscript $i$ on `ls` denotes the tag assigned to the atomic formula occurrence; note that the subformula $\mathtt{ls}\, y''\, y'$ in the second premise, obtained by unfolding $\mathtt{ls}\, v\, v'$ in the conclusion, inherits the tag $i$, in keeping with the rules for forming traces in Definition 5.

The axiom rule (Id) is the other exception because, 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 $F$ `matches` $G$ holds if $F$ and $G$ are equal when all their tags are erased. The implemented form of (Id) is then as follows:

$$\frac{F \;\mathtt{matches}\; F'}{(\alpha, \pi) : F \vdash F'}\,(\mathtt{c\_Id})$$

Finally, we need to add a rule that allows us to form cycles. The ancestry information alone is enough to form cycles, but the progress pointer allows us to only form cycles which contain at least one progress point: In order to find a companion for $(\alpha, \pi) : F \vdash G$, it suffices to find a substitution $\theta$ and an $n$ such that $n > \pi$, $\alpha_n$ 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 $\alpha_n$ are equal up to predicate tags. Thus, the proof rule for link formation in the implemented system is

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

where $|\alpha|$ is the length of the ancestry. This rule ensures that if we can form a downlink from $B$ to $C$ then there is a progressing trace on the finite path $C \dots B$ in the proof tree (and this trace has identical values at $C$ and $B$).

## 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. First, we observe that there is a map $\mathcal{E}$ from proofs in the implemented system to pre-proofs in the system from Section 3. 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, progress pointer and predicate tags 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 $\mathcal{E}(P)$. There must exist a tail $(S_i)_{i\geq n}$ of this path that traverses some strongly connected component $\mathcal{C}$ of $\mathcal{E}(P)$, 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 $(S_i)_{i\geq n}$. Choose $B \in \mathbf{B}$ such that $\mathcal{R}(B)$ is as close as possible 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. (In particular, if a tagged predicate is deleted along a path then that tag cannot be restored further up the tree.) 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 infinitely often through the case-split rule that unrolls a predicate tagged by $i$. □

We note that the soundness condition used in the implemented system is much simpler than the global trace condition of the formal system (Defn. 6), and is almost certainly incomplete. More sophisticated soundness conditions could be implemented at the expense of speed. We note also that our implementation, and its soundness, does not significantly depend on specific features of the fragment of separation logic considered in this paper, and should adapt to other cyclic proof systems employing a similar soundness condition (see [2], ch. 5).

### 4.3 Automated 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 (**emp**R) are then found in this lightweight entailment rather than in the augmented one.

For the augmented entailment to make use of lightweight entailment rules such as (**emp**R), we provide cut rules to inject $\vdash_{basic}$-reasoning into our proofs:

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

It is important that $\vdash_{basic}$ does not interfere with the predicate tags, 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.* Our prover is a collection of HOL tactics arranged into layers:

1. There is a tactic for each rule of the implemented proof system, and tactics are generated for the unfolding rules given by the inductive definitions, as described in section 3. The left rules introduce fresh variables and perform the (potentially unifying) substitutions, while the right rules introduce existential metavariables for any extra exposed variables.
   Additionally, 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`).
2. Since a rule might not be directly applicable until some rearrangements have been performed, specialized tactics are using $\vdash_{basic}$-reasoning to set up rule applications. For the right-unfolding rules, this amounts to bringing the conclusion to the front on the right hand side.
3. "Advancing" rule applications. Right-unfolding rules, for instance, typically expose new state on the right side. An *advancing* version of such a 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 entire rule application fails 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.

With these 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, from a prioritized list of rule sets:

1. {(`c_Id`), link formation}   2. advancing right rules   3. case-split rules

The other rules are only invoked as part of auxiliary reasoning for the rules in these groups.

### 4.4 Experimental performance

Table 2 presents a list of lemmas that have 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, making use of Lemma 3 from Table 2:

*Example 4.* The following is a cyclic proof of $\texttt{RList}\,x\,y \vdash \texttt{List}\,x\,y$, where $R$ and $L$ below abbreviate `RList` and `List` from Table 1, respectively.

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(Id)}
\UnaryInfC{$x \mapsto y \vdash x \mapsto y$}
\RightLabel{$(L\mathrm{R}_1)$}
\UnaryInfC{$x \mapsto y \vdash Lxy$}
\AxiomC{}
\RightLabel{(Id)}
\UnaryInfC{$z \mapsto y \vdash z \mapsto y$}
\AxiomC{$(\dagger)\ \underline{Rxz} \vdash Lxz$}
\RightLabel{$(*)$}
\BinaryInfC{$z \mapsto y * \underline{Rxz} \vdash z \mapsto y * Lxz$}
\AxiomC{(Lemma 3)}
\noLine
\UnaryInfC{$\vdots$}
\noLine
\UnaryInfC{$z \mapsto y * Lxz \vdash Lxy$}
\RightLabel{(Cut)}
\BinaryInfC{$z \mapsto y * \underline{Rxz} \vdash Lxy$}
\RightLabel{(Case $R$)}
\BinaryInfC{$(\dagger)\ \underline{Rxy} \vdash Lxy$}
\end{prooftree}
$$

It seems certain that our theorem prover would benefit from remembering earlier proven lemmas and allowing the automatic tactic to use these, as is provided e.g. by the lemma application mechanism in [17].

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 substantial body of work in the literature that relates to our own work in a variety of ways.

Tuerk's Holfoot [20] is a general framework for separation logic, implemented in HOL, which has automatically proven properties of several interesting pointer manipulating programs. However, Holfoot does not currently support cyclic proof, and we hope that our work may be useful for bringing this technique into such a general verification framework. Similar remarks apply to jStar [13].

Nguyen and Chin [17] describe an extension of an entailment checking technique introduced in earlier work [18] employing a fold/unfold mechanism for user defined inductive predicates. This extension is a mechanism that automatically proves and applies lemmas provided by the user. This mechanism employs a simple version of cyclic proof tailored to their specific verification system; when proving the lemmas, the theorem prover may apply a "smaller" instance of the lemma itself, with recursive lemma application carried out on the root node of inductive predicates. 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

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

**Table 1.** Definitions of predicates

focused on developing a general cyclic entailment checker which could eventually become an off-the-shelf prover for verification tools or theorem provers. In addition, we have the flexibility to easily tune the expressivity of our prover w.r.t. speed by implementing a more general soundness condition (which can be supplied parametrically to the system).

Chang *et al.* [11, 10] propose a shape analysis guided by data structure invariants (provided by the programmer) that describe inductive predicates, called *invariant checkers*. 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 assist such analyses. We believe that our automated cyclic proof engine could be used to support or enhance various operations performed in their shape analysis (e.g. approximation testing, proving termination of fixed point computation, widening, etc.)

There are also a number of provers based upon infinite descent / cyclic proof that are oriented towards proving inductive theorems of arithmetic; we mention by way of example the Coq implementation of Voicu and Li [21], and the stan-

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

**Table 2.** Experimental results.

dalone QUODLIBET system of Wirth [22]. Our system differs from these works in that it is specialised towards separation logic (and thus aims to assist the analyses provided by automated program verification tools).

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 Conclusions and future work

In this paper we have introduced a sound automatic entailment checker for separation logic with inductive predicates based on cyclic proofs, focusing particularly on the soundness of our method and on the careful description of implementation details. The entailment checker has been implemented in HOL Light and has shown significant potential by proving a number of non-trivial lemmas for a range of inductive predicates corresponding to popular data structures. Thus our procedure represents a relevant first step towards the construction of off-the-shelf theorem provers based on separation logic. Our approach also adapts to other cyclic proof systems employing a similar soundness condition.

The automatic entailment checking procedure introduced in this paper opens up several avenues for future work, and in the future we plan to enhance its expressivity and effectiveness in a number of different directions. One direction is to experiment with weakening the soundness condition in order to admit more sophisticated cyclic proofs. Such a generalisation will necessitate more sophisticated tactics for our automated search procedure. Another direction is to extend the expressivity of our formulae by adding features such as quantifiers and arithmetic operations. In doing so it would also be natural to investigate the possibility of integrating our procedure with SMT solvers and arithmetic provers. Finally, we plan to explore the integration of our prover with automatic verification tools such as Holfoot [20] or jStar [13]. In particular, it would be interesting to see how our tool performs on the entailment questions those systems generate.

*Acknowledgments.* We thank Thomas Tuerk and Peter O'Hearn for intensive discussions, and Huu Hai Nguyen and Wei-Ngan Chin for providing us with sample lemmas to add to our experiments. We acknowledge the support of EPSRC and the Royal Academy of Engineering.

## 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. P. Martin-Löf. Haupstatz for the intuitionistic theory of iterated inductive definitions. In *Proc. Second Scandinavian Logic Symposium*, pp. 179–216, 1971.
17. H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *Proceedings of CAV 2008*, LNCS 5123, pp. 355–369. Springer, 2008.
18. 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.
19. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th LICS*, 2002.
20. T. Tuerk. A formalisation of Smallfoot in HOL. In *Theorem Proving in Higher Order Logics*, LNCS, pp. 469–484. Springer, 2009.
21. R. Voicu and M. Li. Descente infinie proofs in Coq. In *Proceedings of the 1st Coq Workshop*. Technische Universität München, 2009.
22. C.-P. Wirth. Descente Infinie + Deduction. In *Logic Journal of the IGPL* 12(1): 1–96. Oxford University Press, 2004.
23. 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.