Cyclic Abduction of Inductively Defined Safety and Termination Preconditions

James Brotherston[1] and Nikos Gorogiannis[2]

[1] Dept. of Computer Science, University College London
[2] Dept. of Computer Science, Middlesex University London

Abstract. We introduce *cyclic abduction*: a new method for automatically inferring safety and termination preconditions of heap-manipulating `while` programs, expressed as inductive definitions in separation logic. Cyclic abduction essentially works by searching for a *cyclic proof* of the desired property, abducing definitional clauses of the precondition as necessary in order to advance the proof search process.
We provide an implementation, Caber, of our cyclic abduction method, based on a suite of heuristically guided tactics. It is often able to automatically infer preconditions describing lists, trees, cyclic and composite structures which, in other tools, previously had to be supplied by hand.

1 Introduction

Whether a given pointer program is *memory-safe*, or eventually *terminates*, under a given precondition, are well-known (and undecidable) problems in program analysis. In this paper, we consider the even more difficult problem of *inferring* reasonable safety / termination preconditions, in *separation logic* [21] with inductive definitions, for such heap-aware programs.

Analyses of heap-manipulating programs based upon separation logic now extend, in some cases, to substantial code bases (see e.g. [22,20]), and rely on the use of *inductive predicates* to specify the shape of data structures stored in memory. However, such predicates are typically hard-coded into these analyses, which must therefore either give up or ask the user for advice when they encounter a data structure not described by the hard-coded predicates. For example, the well known SpaceInvader [22] and SLAyer [5] analysers perform accurately on programs using combinations of linked lists, but report a false bug if they encounter a tree. Thus automatically inferring, or *abducing*, the inductive predicates needed to analyse individual procedures has the potential to greatly boost the automation of such verifiers.

The abduction of safety or termination preconditions is a highly non-trivial problem. At one end of the scale, the *weakest (liberal) precondition* (cf. Dijkstra [14]) can straightforwardly be extracted from a program P, but is useless for analysis: Deciding which program states satisfy this precondition is as hard as deciding from which states P runs safely / terminates! At the other end, many correct preconditions are too strong in that they rule out the execution

of some or all of the program. Thus we are required to perform a fine balancing act: find the weakest precondition that is at least somewhat "natural". Unfortunately, for fundamental computability reasons, we cannot hope to obtain such a precondition in general, so we must instead look for reasonable approximating heuristics.

Our main contribution is a new method, *cyclic abduction*, for inferring safety and/or termination preconditions, expressed as inductive definitions in separation logic, for heap-manipulating `while` programs. Our approach is based upon heuristic search in a formal system of *cyclic proofs*, adapted from the cyclic termination proofs in [8]. A cyclic proof is a derivation tree possibly containing *back-links*, which identify leaves of the tree with arbitrary interior nodes. This can create potentially unsound cycles in the reasoning, and so a (decidable) global soundness condition must be imposed upon these derivations to qualify them as genuine proofs. In fact, we can consider cyclic proofs of memory safety or of termination as desired, simply by imposing two different soundness conditions.

Given a program, cyclic abduction aims to simultaneously construct an inductively defined precondition in separation logic and a cyclic proof of safety or termination for the program under this precondition. Broadly speaking, we search for a cyclic proof that the program has the desired property, and when the proof search gets stuck, we abduce (i.e., guess) part of the precondition in order to proceed. Approximately, the main abduction principles are:

- symbolically executing *branching commands* in the derivation leads to *conditional disjunction* in the definitions;
- symbolically executing *dereferencing commands* in the derivation forces us to include *pointer formulas* in the definitions;
- forming *back-links* in the derivation leads to the instantiation of *recursion* in the definitions; and
- encountering a *loop* in the program alerts us to the possibility that we may need to *generalise* the precondition.

We have implemented our abduction procedure as an automatic tool, Caber, that builds on the generic cyclic theorem prover Cyclist [11]. Caber is able to automatically abduce safety and/or termination preconditions for a fairly wide variety of common small programs, including the majority of those tested in the Mutant tool, where the (list-based) preconditions previously had to be supplied by hand [2]. Caber can abduce definitions of a range of data structures such as lists, trees, cyclic structures, or composites such as trees-of-lists.

The remainder of this paper is structured as follows. Section 2 introduces the programming language and our language of logical preconditions. Section 3 presents our formal system of cyclic safety/termination proofs on which our abduction technique is based. In Section 4 we present our cyclic abduction strategy in detail, and Section 5 describes the implementation of Caber and its experimental evaluation. Section 6 examines related work and Section 7 concludes.

Due to space limitations, we have had to omit quite a few details. These can be found in an earlier technical report [10].

2 Programs and preconditions

In this section we present a basic language of `while` programs with heap pointers, and the fragment of separation logic we use to express program preconditions. We often use vector notation to abbreviate tuples or lists, e.g. $\mathbf{x}$ for $(x_1, \ldots, x_k)$, and we write $\mathbf{x}_i$ for the ith element of the tuple $\mathbf{x}$.

Syntax of programs. We assume infinite sets Var of *variables* and Fld of *field names*. An *expression* is either a variable or the constant nil. *Branching conditions* B and *command sequences* C are defined as follows, where x, y range over Var, f over Fld and E over expressions:

$$\begin{aligned} B &::= \star \mid E = E \mid E \neq E \\ C &::= \epsilon \mid x := E;\, C \mid y := x.f;\, C \mid x.f := E;\, C \mid \\ &\quad\ \texttt{free}(x);\, C \mid x := \texttt{new}();\, C \mid \\ &\quad\ \texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C\, \texttt{fi};\, C \mid \texttt{while}\, B\, \texttt{do}\, C\, \texttt{od};\, C \end{aligned}$$

where $y := x.f$ and $x.f := E'$ respectively read from and write to field f of the heap cell with address x, and $\star$ represents a non-deterministic condition. A *program* is simply a list of field names followed by a command sequence: $\texttt{fields}\ f_1, \ldots, f_k;\ C$.

Program semantics. We use a RAM model employing heaps of records. We fix a set Val of *values* and an infinite subset $\mathsf{Loc} \subset \mathsf{Val}$ of *locations*, i.e., addresses of heap cells. The "nullary" value $nil \in \mathsf{Val} \setminus \mathsf{Loc}$ will not be the address of any heap cell. A *stack* is a function $s : \mathsf{Var} \to \mathsf{Val}$. The semantics $[\![E]\!]s$ of expression E in stack s is defined by $[\![x]\!]s =_{\text{def}} s(x)$ for $x \in \mathsf{Var}$, and $[\![\mathsf{nil}]\!]s =_{\text{def}} nil$.

A *heap* is a partial function $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} (\mathsf{Val}\ \mathsf{List})$ mapping finitely many locations to tuples of values (i.e. records); we write $dom(h)$ for the *domain* of heap h, i.e. the set of locations on which h is defined, and e for the empty heap that is undefined everywhere. If h_1 and h_2 are heaps with $dom(h_1) \cap dom(h_2) = \emptyset$, we define $h_1 \circ h_2$ to be the union of h_1 and h_2; otherwise, $h_1 \circ h_2$ is undefined.

We write $s[x \mapsto v]$ to denote the stack defined exactly as s except that $(s[x \mapsto v])(x) = v$, and adopt a similar update notation for heaps.

We employ a standard small-step operational semantics of our programs. A *(program) state* is either a triple (C, s, h) where C is a command sequence, s a stack and h a heap, or the special state *fault*, used to catch memory errors. Given a program $\texttt{fields}\ f_1, \ldots, f_k;\ C$, we map the field names $f_1, \ldots, f_k$ onto elements of heap records by $\overline{f_j} =_{\text{def}} j$. The semantics of programs is then standard, given by a relation $\rightsquigarrow$ on states (omitted here for space reasons, but see [10]). We write $\rightsquigarrow^n$ for the n-step variant of $\rightsquigarrow$, and $\rightsquigarrow^*$ for its reflexive-transitive closure. A state (C, s, h) is *safe* if there is no computation $(C, s, h) \rightsquigarrow^* fault$, and *terminating* if it is safe and there is no infinite $\rightsquigarrow$-computation starting from (C, s, h).

As in [23,8], extending the heap memory cannot lead to new memory faults under our semantics, and so the following proposition holds:

Proposition 1. *If (C, s, h) is safe (resp. terminating) and $h \circ h'$ is defined then $(C, s, h \circ h')$ is also safe (terminating).*

Syntax of preconditions. We express preconditions using the *symbolic heap* fragment of separation logic [3] extended with inductive definitions. We assume an infinite set of *predicate symbols*, each with associated arity.

Definition 1. *Formulas* are given by the following grammar:

$$F ::= \top \mid \bot \mid E = E \mid E \neq E \mid \mathsf{emp} \mid x \mapsto \mathbf{E} \mid P(\mathbf{E}) \mid F * F$$

where $x \in \mathsf{Var}$, E ranges over expressions, P over predicate symbols and $\mathbf{E}$ over tuples of expressions (matching the arity of P in $P(\mathbf{E})$). We write $F[E/x]$ for the result of replacing all occurrences of variable x by the expression E in formula F. Substitution is extended pointwise to tuples; but when we write $F[E/\mathbf{x}_i]$, we mean that E should be substituted for the ith component of $\mathbf{x}$ *only*.

We define $\equiv$ to be the least equivalence on formulas closed under associativity and commutativity of $*$ and $F * \mathsf{emp} \equiv F$.

Definition 2. An *inductive rule set* is a finite set of *inductive rules* each of the form $F \overset{\mathbf{z}}{\Rightarrow} P(\mathbf{E})$, where F and $P(\mathbf{E})$ are formulas and $\mathbf{z}$ (often suppressed) is a tuple listing the set of all variables appearing in F and $\mathbf{E}$. If Φ is an inductive rule set we define Φ_P to be the set of all *inductive rules for* P in Φ, i.e. those of the form $F \Rightarrow P(\mathbf{E})$. We say P is *undefined* if Φ_P is empty.

Semantics of preconditions. Satisfaction $s, h \models_\Phi F$ of the formula F by stack s and heap h under inductive rule set Φ is defined by structural induction:

$$\begin{array}{lll}
s,h \models_\Phi \top & \Leftrightarrow & \text{always} \\
s,h \models_\Phi \bot & \Leftrightarrow & \text{never} \\
s,h \models_\Phi E_1 = E_2 & \Leftrightarrow & [\![E_1]\!]s = [\![E_2]\!]s \text{ and } h = e \\
s,h \models_\Phi E_1 \neq E_2 & \Leftrightarrow & [\![E_1]\!]s \neq [\![E_2]\!]s \text{ and } h = e \\
s,h \models_\Phi \mathsf{emp} & \Leftrightarrow & h = e \\
s,h \models_\Phi E \mapsto \mathbf{E} & \Leftrightarrow & \mathit{dom}(h) = \{[\![E]\!]s\} \text{ and } h([\![E]\!]s) = [\![\mathbf{E}]\!]s \\
s,h \models_\Phi P(\mathbf{E}) & \Leftrightarrow & ([\![\mathbf{E}]\!]s, h) \in [\![P]\!]^\Phi \\
s,h \models_\Phi F_1 * F_2 & \Leftrightarrow & h = h_1 \circ h_2 \text{ and } s,h_1 \models_\Phi F_1 \text{ and } s,h_2 \models_\Phi F_2
\end{array}$$

Note that we interpret (dis)equalities as holding in the empty heap. The semantics $[\![P]\!]^\Phi$ of the predicate P under Φ is defined as follows:

Definition 3. Assume that Φ defines predicates $P_1, \ldots, P_n$ with respective arities $a_1, \ldots, a_n$. We let each Φ_{P_i} be indexed by j, and for an inductive rule $\Phi_{P_i,j}$ of the form $F \Rightarrow P_i\mathbf{x}$, we define an operator $\varphi_{i,j}$ by:

$$\varphi_{i,j}(\mathbf{X}) =_{\text{def}} \{(s(\mathbf{x}), h) \mid s, h \models_{\mathbf{X}} F\}$$

where $\mathbf{X} = (X_1, \ldots, X_n)$ and each $X_i \subseteq \mathsf{Val}^{a_i} \times \mathsf{Heaps}$, and $\models_{\mathbf{X}}$ is the satisfaction relation above, except that $[\![P_i]\!]^{\mathbf{X}} =_{\text{def}} X_i$. We then define

$$[\![\mathbf{P}]\!]^\Phi =_{\text{def}} \mu\mathbf{X}.\ (\textstyle\bigcup_j \varphi_{1,j}(\mathbf{X}), \ldots, \bigcup_j \varphi_{n,j}(\mathbf{X}))$$

We write $[\![P_i]\!]^\Phi$ for the ith component of $[\![\mathbf{P}]\!]^\Phi$.

For any inductive rule set, the *satisfiability* of a formula in our fragment is decidable [9], which is very helpful in evaluating abduced preconditions. On the other hand, *entailment* between formulas in the fragment is undecidable [1].

3 Formal cyclic safety/termination proofs

Here we present our formal cyclic proof system, adapted from the cyclic termination proofs in [8], for proving memory safety and/or termination of programs. We can consider memory safety rather than termination simply by imposing an alternative soundness condition on proofs.

A *proof judgement* is given by $F \vdash C$, where C is a command sequence and F is a formula. The proof rules for judgements are given in Fig. 1. By convention, the primed variables x', x'' etc. appearing in the premises of rules are chosen *fresh*, and we write $\overline{B}$ to mean $E \neq E'$ if B is $E = E'$, and vice versa. The rule (Frame) can be seen as a special case of the general *frame rule* of separation logic (cf. [23]), where the postcondition is omitted; its soundness depends on Proposition 1. We also include a rule for unfolding a formula of the form $P(\mathbf{E})$ according to the definition of P in a given inductive rule set Φ. (Predicate folding is a special case of *lemma application*, handled by the (Cut) rule.)

Definition 4. The judgement $F \vdash C$ is *valid* (resp. *termination-valid*) w.r.t. inductive rule set Φ if $s, h \models_\Phi F$ implies (C, s, h) is safe (resp. terminating).

Lemma 1. *Suppose the conclusion $F \vdash C$ of an instance of a rule R from Figure 1 is invalid w.r.t. Φ, i.e. $s, h \models_\Phi F$ but $(C, s, h) \rightsquigarrow^n$ fault for some stack s, heap h and $n \in \mathbb{N}$. Then there is a premise $F' \vdash C'$ of this rule instance and stack s', heap h' and $m \in \mathbb{N}$ such that $s', h' \models_\Phi F'$, but $(C', s', h') \rightsquigarrow^m$ fault. Moreover, $m \leq n$, and if R is a symbolic execution rule then $m < n$.*

Definition 5. A *pre-proof* of $F \vdash C$ is a pair $(\mathcal{D}, \mathcal{L})$, where $\mathcal{D}$ is a finite derivation tree with $F \vdash C$ at its root, and $\mathcal{L}$ is a "*back-link*" function assigning to every open leaf ℓ of $\mathcal{D}$ a node $\mathcal{L}(\ell)$ of $\mathcal{D}$ such that the judgements at ℓ and $\mathcal{L}(\ell)$ are identical. A pre-proof $(\mathcal{D}, \mathcal{L})$ can be seen as a graph by identifying each open leaf ℓ of $\mathcal{D}$ with $\mathcal{L}(\ell)$; a *path* in $\mathcal{P}$ is then understood as usual.

Definition 6. A pre-proof $\mathcal{P}$ is a *cyclic (safety) proof* if there are infinitely many symbolic execution rule applications along every infinite path in $\mathcal{P}$.

We can treat termination rather than safety by replacing the soundness condition of Defn. 6 with the condition in [8], which essentially demands that some inductive predicate is unfolded infinitely often along every infinite path in the pre-proof. Thus, by a simple adaptation of the soundness result in [8]:

Theorem 7. *For any inductive rule set Φ, if there is a cyclic safety (resp. termination) proof of $F \vdash C$, then $F \vdash C$ is valid (resp. termination-valid) w.r.t. Φ.*

Proof. We just consider safety here, and refer to [8] for the termination case. Suppose $F \vdash C$ has a cyclic safety proof $\mathcal{P}$ but is invalid. By Lemma 1, there is an infinite path $(F_k \vdash C_k)_{k \geq 0}$ in $\mathcal{P}$, and an infinite sequence $(n_k)_{k \geq 0}$ of natural numbers such that $n_{k+1} < n_k$ whenever $F_k \vdash C_k$ is the conclusion of a symbolic execution rule instance, and $n_{k+1} = n_k$ otherwise. Since $\mathcal{P}$ is a cyclic safety proof, there are infinitely many symbolic executions along $(F_k \vdash C_k)_{k \geq 0}$. Thus $(n_k)_{k \geq 0}$ is an infinite descending chain of natural numbers, contradiction. □

Symbolic execution rules:

$$\frac{x = E[x'/x] * F[x'/x] \vdash C}{F \vdash x := E;\, C} \qquad \frac{}{F \vdash \epsilon}$$

$$\frac{x = \mathbf{E}_{\overline{f}}[x'/x] * (y \mapsto \mathbf{E} * F)[x'/x] \vdash C}{y \mapsto \mathbf{E} * F \vdash x := y.f;\, C}\ |\mathbf{E}| \geq \overline{f} \qquad \frac{x \mapsto \mathbf{E}[E/\mathbf{E}_{\overline{f}}] * F \vdash C}{x \mapsto \mathbf{E} * F \vdash x.f := E;\, C}\ |\mathbf{E}| \geq \overline{f}$$

$$\frac{x \mapsto (x'_1, \ldots, x'_k) * F[x'/x] \vdash C}{F \vdash x := \texttt{new}();\, C} \qquad \frac{F \vdash C}{x \mapsto \mathbf{E} * F \vdash \texttt{free}(x);\, C}$$

$$\frac{B * F \vdash C;\, C''}{B * F \vdash \texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C'\, \texttt{fi};\, C''} \qquad \frac{B * F \vdash C;\, \texttt{while}\, B\, \texttt{do}\, C\, \texttt{od};\, C'}{B * F \vdash \texttt{while}\, B\, \texttt{do}\, C\, \texttt{od};\, C'}$$

$$\frac{\overline{B} * F \vdash C';\, C''}{\overline{B} * F \vdash \texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C'\, \texttt{fi};\, C''} \qquad \frac{\overline{B} * F \vdash C'}{\overline{B} * F \vdash \texttt{while}\, B\, \texttt{do}\, C\, \texttt{od};\, C'}$$

$$\frac{F \vdash C;\, C'' \qquad F \vdash C';\, C''}{F \vdash \texttt{if}\, \star\, \texttt{then}\, C\, \texttt{else}\, C'\, \texttt{fi};\, C''} \qquad \frac{F \vdash C;\, \texttt{while}\, B\, \texttt{do}\, C\, \texttt{od};\, C' \qquad F \vdash C'}{F \vdash \texttt{while}\, \star\, \texttt{do}\, C\, \texttt{od};\, C'}$$

Logical rules:

$$\frac{F \vdash C}{F * G \vdash C}\ (\text{Frame}) \qquad \frac{F \vdash C}{F[E/x] \vdash C}\ x \text{ not in } C\ (\text{Subst}) \qquad \frac{F' \vdash C}{F \vdash C}\ F \equiv F'\ (\text{Equiv})$$

$$\frac{(t_1 = t_2 * F)[t_2/x, t_1/y] \vdash C}{(t_1 = t_2 * F)[t_1/x, t_2/y] \vdash C}\ (=) \qquad \frac{G' * F \vdash C}{G * F \vdash C}\ G \vdash G'\ (\text{Cut})$$

$$\frac{}{t_1 = t_2 * t_1 \neq t_2 * F \vdash C}\ (\neq) \qquad \frac{}{x \mapsto \mathbf{E} * x \mapsto \mathbf{E}' * F \vdash C}\ (\mapsto)$$

Predicate unfolding rule:

$$\frac{(\mathbf{E} = \mathbf{E_j}[\mathbf{x_j}/\mathbf{z_j}] * F_j[\mathbf{x_j}/\mathbf{z_j}] * F \vdash C)_{1 \leq j \leq k}}{P(\mathbf{E}) * F \vdash C}\ \begin{array}{l} \Phi_P = \{F_1 \overset{\mathbf{z_1}}{\Rightarrow} P(\mathbf{E_1}), \ldots, F_k \overset{\mathbf{z_k}}{\Rightarrow} P(\mathbf{E_k})\} \\ \forall x_j \in \{\mathbf{x_j}\}.\ x_j \text{ is fresh} \end{array}\ (P)$$

Fig. 1. Hoare logic rules for proof judgements.

4 Cyclic abduction: basic strategy & tactics

We now turn to the main contribution of this paper: our *cyclic abduction* method for inferring inductive safety and/or termination preconditions of programs. Here, we first explain the high-level strategy for abducing such preconditions, and then develop a number of automatic *tactics* implementing this strategy.

4.1 Overview of abduction strategy. The typical initial problem we are faced with is: given a program with code C and input variables $\mathbf{x}$, find an inductive definition set Φ such that the judgement $P(\mathbf{x}) \vdash C$ is (termination-)valid wrt. Φ, where P is a predicate symbol.

Our strategy for finding such a Φ is to *search for a cyclic proof* of the judgement $P(\mathbf{x}) \vdash C$, abducing inductive rules as necessary to enable the search to progress. We now set out informally the main principles governing this process.

Principle 1. *The first priority of the search procedure is to close the current branch of the derivation tree, preferably by* applying an axiom*, or else by* forming a back-link *to some other node. (The formation of back-links must respect the relevant soundness condition on cyclic proofs.)*

If closing the branch is not possible, the second priority is to apply the symbolic execution rule *for the (foremost) command appearing at the current subgoal.*

Principle 2. *We may abduce inductive rules and/or deploy the logical rules as "helper functions" to serve the priorities laid out in Principle 1, i.e., in order to form a back-link or to apply the symbolic execution rule for a command.*

We may abduce inductive rules only *for predicate symbols that are in the current subgoal, and currently undefined. When we abduce inductive rules for a predicate, we always immediately unfold that predicate in the current subgoal.*

Principle 3. *Before symbolically executing a* `while` *loop, one can attempt to generalise the precondition F appearing at the subgoal in question. That is to say, we can try to find a formula F' such that $F' \vdash F$ is a valid entailment, and, by applying (Cut), proceed with the proof search using the precondition F' in place of F. If necessary, we may abduce inductive rules in order to obtain F'.*

4.2 Tactics. A *tactic* in our setting, as is standard in automated theorem proving, is simply a transformer on *proof states*. However, since we employ cyclic proofs with back-links joining leaves to arbitrary proof nodes, our proof state must reflect the entire pre-proof rather than just the current subgoal. Furthermore, since we are allowed to abduce new inductive rules in the proof search, the current inductive rule set must also form part of the proof state. Thus our proof states are comprised of the following elements:

$\mathcal{P}$: A partial pre-proof, representing the portion of proof constructed so far.
Some of the leaves of $\mathcal{P}$ may be open; we call these the *open subgoals* of $\mathcal{P}$.
Φ: The set of inductive rules abduced so far in the proof search.
ℓ: The open subgoal of $\mathcal{P}$ on which to operate next.

Example 1. Figure 2 shows an abductive cyclic proof of a program that non-deterministically traverses l and r fields of pointer x until it reaches nil; as expected, the abduced predicate defines binary trees. We will often refer to this proof, which satisfies both the safety *and* the termination soundness condition, as a running example in order to illustrate our abductive tactics.

4.3 Abductive tactic for branching commands. Our proof rules for deterministic `if` and `while` commands (Fig. 1) require the precondition to determine the status of the branching condition. We introduce an abductive tactic, `abduce_branch`, that fires whenever the symbolic execution of such a rule fails.

Suppose `abduce_branch` is applied to the proof state $(\mathcal{P}, \Phi, \ell)$ where the command sequence $\mathcal{C}$ in the judgement appearing at the current subgoal ℓ is of the form $\texttt{while}\, B\, \texttt{do}\, C\, \texttt{od}; C'$ or $\texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C'\, \texttt{fi}; C''$ (where $B \neq \star$). For simplicity we assume B is an equality or disequality between two program variables x, y (the case where one of the two terms is nil is very similar). First, `abduce_branch` selects a subformula of the form $P(\mathbf{E})$ appearing in ℓ such that P is undefined in Φ, and x and y occur in the tuple $\mathbf{E}$. Thus, we may write the judgement appearing at ℓ as $F * P(\mathbf{E}) \vdash \mathcal{C}$ where $x = \mathbf{E}_k$ and $y = \mathbf{E}_j$ (and $k \neq j$). Then, `abduce_branch` adds the following inductive rules for P to Φ:

$$B[\mathbf{z}_k/x, \mathbf{z}_\ell/y] * P'(\mathbf{z}) \Rightarrow P(\mathbf{z})$$
$$\overline{B}[\mathbf{z}_k/x, \mathbf{z}_\ell/y] * P''(\mathbf{z}) \Rightarrow P(\mathbf{z})$$

where P', P'' are fresh predicate symbols and $\mathbf{z}$ is a tuple of appropriately many arbitrary variables. `abduce_branch` then unfolds the indicated occurrence of $P(\mathbf{E})$ in ℓ, and applies the appropriate symbolic execution rule for $\mathcal{C}$ to each of the new subgoals (this step is now guaranteed to succeed).

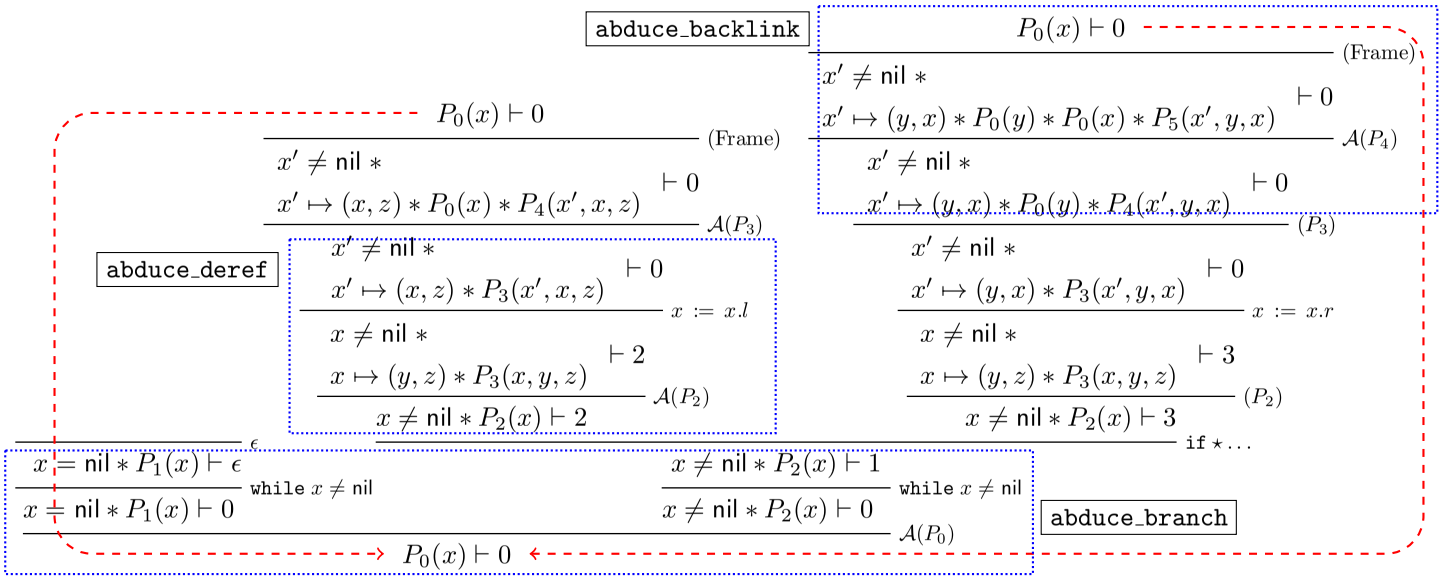
The proof in Figure 2 begins by applying `abduce_branch` in order to symbolically execute the `while` command, abducing a suitable definition of predicate P_0.

4.4 Abductive tactic for dereferencing assignments. The symbolic execution rules for commands that dereference a memory address (Fig. 1) require the precondition to guarantee that this address is indeed allocated. The tactic `abduce_deref` enables the symbolic execution of such commands by abducing the allocation of the appropriate address.

Formally, suppose `abduce_deref` is applied to the proof state $(\mathcal{P}, \Phi, \ell)$, where the first command $\mathcal{C}$ in the judgement at ℓ is of the form $\texttt{free}(x)$ or $x.f := E$ or $y := x.f$. First, `abduce_deref` selects a subformula of the form $P(\mathbf{E})$ appearing at ℓ, where P is undefined in Φ, and x occurs in the tuple $\mathbf{E}$ at position k (i.e., $x = \mathbf{E}_k$). Then, the inductive rule below is added to Φ:

$$P'(\mathbf{x} \sqcup \mathbf{y}) * x_k \mapsto \mathbf{y} \Rightarrow P(\mathbf{x})$$

where $\sqcup$ is tuple concatenation, P' is a fresh predicate symbol, and $\mathbf{x}$ and $\mathbf{y}$ are tuples of distinct, fresh variables such that $|\mathbf{x}| = |\mathbf{E}|$, and $|\mathbf{y}|$ is the number of fields in the program. `abduce_deref` then unfolds the selected occurrence of




```
    fields l, r;
0 : while x ≠ nil do
1 :   if ⋆ then
2 :     x := x.l
      else
3 :     x := x.r
      fi od
```

$$x = \mathsf{nil} * P_1(x) \Rightarrow P_0(x)$$
$$x \neq \mathsf{nil} * P_2(x) \Rightarrow P_0(x)$$
$$x \mapsto (y, z) * P_3(x, y, z) \Rightarrow P_2(x)$$
$$P_0(y) * P_4(x, y, z) \Rightarrow P_3(x, y, z)$$
$$P_0(z) * P_5(x, y, z) \Rightarrow P_4(x, y, z)$$

$$x = \mathsf{nil} * \mathsf{emp} \Rightarrow P_0(x)$$
$$x \neq \mathsf{nil} * x \mapsto (y, z) * P_0(y) * P_0(z) \Rightarrow P_0(x)$$

Fig. 2. Top: abductive proof for a binary tree search program (shown bottom left). Note that judgements refer to the indices attached to program commands; and we write $\mathcal{A}(P)$ to indicate a combined abduction-and-unfolding proof step. Bottom centre: inductive rules abduced during the proof. Bottom right: simplified inductive rules.

$P(\mathbf{E})$ in ℓ (introducing fresh variables as appropriate), and applies the symbolic execution rule for $\mathcal{C}$ to the resulting subgoal (which will now succeed).

In the case of the proof in Figure 2, we apply `abduce_deref` when attempting to symbolically execute the command $x := x.l$ on the middle branch at line 2, abducing a suitable definition for P_2 in the process. A similar situation arises when we attempt to symbolically execute the command $x := x.r$ on the right hand branch, with the crucial difference that here the only predicate in the precondition, P_2, has already been defined. In this case, `abduce_deref` is able to succeed by unfolding $P_2(x)$ according to its existing definition.

4.5 Abductive tactic for forming back-links. In principle, we may attempt to form a back-link from an open subgoal labelled by $F \vdash C$ to any other proof node labelled by $F' \vdash C$, provided that: (a) $F \vdash F'$ is a valid entailment; and (b) the addition of this back-link does not violate the soundness condition on cyclic proofs. Here we present a tactic, `abduce_backlink`, that attempts to form such back-links automatically during the proof search.

Formally, suppose that `abduce_backlink` is applied to proof state $(\mathcal{P}, \Phi, \ell)$. First, the tactic non-deterministically selects a node ℓ' of $\mathcal{P}$, distinct from ℓ, such that the command sequences at ℓ' and ℓ are identical. Then it tries to manipulate ℓ using logical rules so as to obtain a precondition identical to the one at ℓ'. More precisely, for any predicate P in ℓ that is undefined in Φ, `abduce_backlink` attempts to abduce inductive rules for P such that after unfolding P, the logical rules (Frame) and (Subst) can be used to obtain an identical copy of ℓ'.

We write ℓ as $F_1 * P(\mathbf{E}) \vdash C$, where P is undefined in Φ, and ℓ' as $F_2 \vdash C$. Then `abduce_backlink` abduces an inductive rule of the form $F' * P'(\mathbf{z}) \Rightarrow P(\mathbf{z})$ where P' is a fresh predicate, and F' is chosen so as to satisfy

$$F_2[\theta] \subseteq_{\text{multiset}} F_1 * F'[\mathbf{E}/\mathbf{z}]$$

for some substitution θ of expressions for *non-program variables* only (here we view formulas as $*$-separated multisets). Providing we can find suitable F' (which is essentially a unification problem), `abduce_backlink` transforms $\mathcal{P}$ by applying rules to ℓ and inserting a new back-link to ℓ' as follows:

$$\cfrac{\vdots}{\cfrac{F_2 \vdash C}{\vdots}} \qquad \cfrac{\cfrac{\cfrac{F_2 \vdash C}{F_2[\theta] \vdash C}\,(\text{Subst})}{F_1 * F'[\mathbf{E}/\mathbf{z}] * P'(\mathbf{E}) \vdash C}\,(\text{Frame})}{F_1 * P(\mathbf{E}) \vdash C}\,(P)$$

As with our other tactics, `abduce_backlink` is also allowed to try unfolding a defined predicate in the subgoal ℓ if no undefined predicates are available. Finally, `abduce_backlink` ensures that the proposed back-link does not violate the relevant soundness condition on cyclic proofs by calling a model checker.

In the middle branch of the proof in Figure 2, we call `abduce_backlink` after symbolically executing $x := x.l$. The tactic proceeds by abducing a suitable definition for P_3 and applying (Frame), allowing a back-link to the root of the

proof. For the similar goal on the rightmost branch, `abduce_backlink` instead unfolds P_3 and then abduces a suitable inductive rule for the undefined P_4.

We observe that `abduce_backlink` is "forgetful" in that it uses (Frame) to discard parts of the precondition. An alternative would be to use (Cut) with an entailment theorem prover to establish the required logical entailment $F \vdash F'$ (such steps are needed for some proofs). We did implement such a tactic, calling on the separation logic entailment prover in Cyclist [11], but found the costs to be prohibitive in the absence of a sophisticated lemma speculation mechanism.

4.6 Tactic for existential generalisation. Symbolically executing `while` loops creates a potentially infinite branch of the proof search, unless it can be closed either by an axiom or, more commonly, by forming a back-link. However, naive attempts to back-link to a target judgement often fail because the judgement specifies a too-precise relationship between program variables which is not preserved by the loop body. One solution, typical of inductive theorem proving in general, is to *generalise* the precondition of a `while` loop so as to "forget" such variable relationships. The tactic `ex_gen` implements this principle.

Formally, suppose `ex_gen` is applied to the proof state $(\mathcal{P}, \Phi, \ell)$, where the judgement labelling current subgoal ℓ is of the form $F \vdash \texttt{while}\ B\ \texttt{do}\ C\ \texttt{od};\ C'$. Then for every program variable x modified by the loop body C, `ex_gen` replaces every occurrence of x in a subformula of F of the form $E = E'$, $E \neq E'$ or $y \mapsto \mathbf{E}$ by a fresh (existentially quantified) variable w. (This step uses (Cut), and is easily seen to be sound.) This tactic may generalise over *any* subset of variables modified by the loop body and present in F.

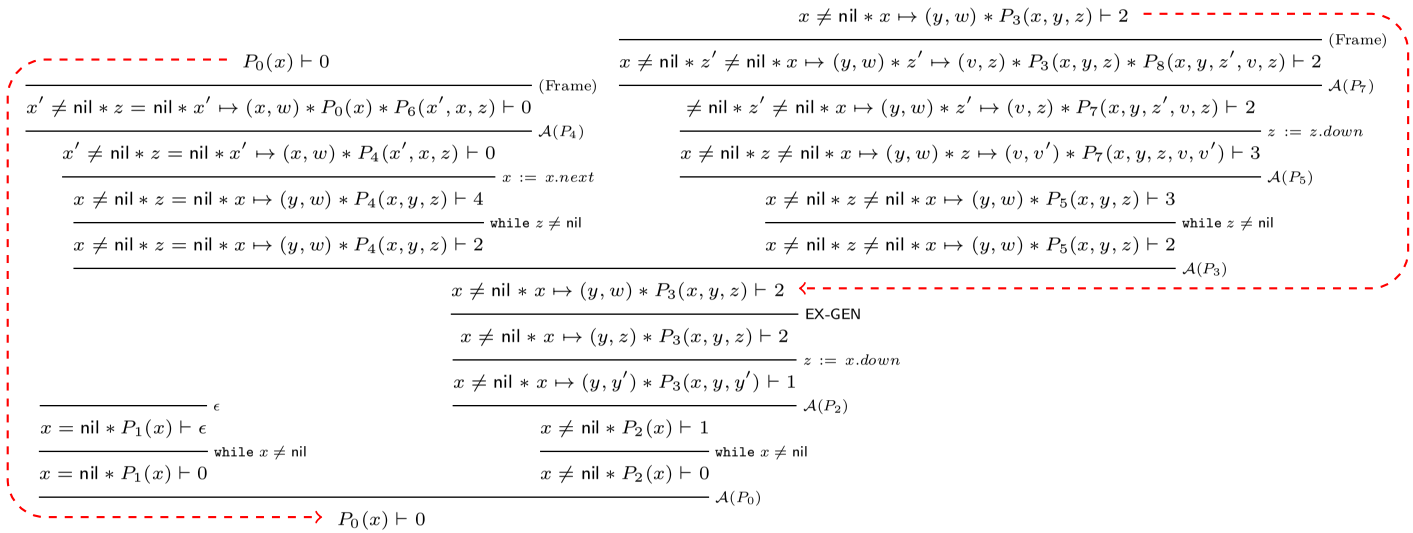
Example 2. Figure 3 shows the proof of a program with two nested `while` loops; the outer loop traverses *next* pointers while the inner loop traverses *down* pointers. Here, the abduced precondition defines a list of lists.

Consider the goal $x \neq \mathsf{nil} * x \mapsto (y, z) * P_3(x, y, z) \vdash 2$ in Figure 3. Since z is modified by the inner loop body, and the precondition contains $x \mapsto (y, z)$, we call `ex_gen`, which replaces $x \mapsto (y, z)$ by $x \mapsto (y, w)$, where w is a fresh variable. This generalisation will be needed later in order to form a backlink (as $x \mapsto (y, z)$ does not hold after executing the loop body, but $\exists w.\ x \mapsto (y, w)$ does).

Other, more complex types of generalisation are also possible (and are needed for some proofs), but are outside the scope of what we can cover in a single paper.

4.7 Simplification of inductive rule sets. When an abductive proof search succeeds, the returned set of abduced inductive rules will typically be too complex for human consumption. We apply some fairly straightforward simplifications to improve readability (as shown in Figures 2 and 3).

First, all undefined predicates are interpreted as the empty memory `emp` (this being a safe and spatially minimal interpretation). Second, we in-line the definitions of predicates defined by a single inductive rule; to ensure this process terminates, the definition of Q may only be in-lined into the body of P when Q was abduced later in the search than P. Finally, we eliminate any parameters from a predicate that are unused by its definition and therefore redundant.



$$
\begin{array}{l}
\texttt{fields } next, down; \\
0: \texttt{while } x \neq \mathsf{nil} \texttt{ do} \\
1: \quad z := x.down\,; \\
2: \quad \texttt{while } z \neq \mathsf{nil} \texttt{ do} \\
3: \quad\quad z := z.down \ \texttt{od}; \\
4: \quad x := x.next \\
\texttt{od}
\end{array}
$$

$$
\begin{array}{r}
x = \mathsf{nil} * P_1(x) \Rightarrow P_0(x) \\
x \neq \mathsf{nil} * P_2(x) \Rightarrow P_0(x) \\
x \mapsto (y, y') * P_3(x, y, y') \Rightarrow P_2(x) \\
z = \mathsf{nil} * P_4(x, y, z) \Rightarrow P_3(x, y, z) \\
z \neq \mathsf{nil} * P_5(x, y, z) \Rightarrow P_3(x, y, z) \\
P_0(y) * P_6(x, y, z) \Rightarrow P_4(x, y, z) \\
z \mapsto (v, v') * P_7(x, y, z, v, v') \Rightarrow P_5(x, y, z) \\
P_3(x, y, w) * P_8(x, y, z, v, w) \Rightarrow P_7(x, y, z, v, w)
\end{array}
$$

$$
\begin{array}{r}
x = \mathsf{nil} \Rightarrow P_0(x) \\
x \neq \mathsf{nil} * x \mapsto (y, y') * P_3(y, y') \Rightarrow P_0(x) \\
z = \mathsf{nil} * P_0(y) \Rightarrow P_3(y, z) \\
z \neq \mathsf{nil} * z \mapsto (v, v') * P_3(y, v') \Rightarrow P_3(y, z)
\end{array}
$$

Fig. 3. Top: abductive proof for list-of-lists traversal. Bottom, left to right: program; predicates found; simplified predicates.

5 Implementation and evaluation

We have implemented our cyclic abduction strategy as an experimental tool, CABER (from "Cyclic ABducER"). CABER is built on top of the open-source theorem prover CYCLIST, a generic framework for constructing cyclic theorem provers [11]. It essentially provides an instantiation of the proof system in Section 3 (based on an earlier version in [11]), and an abductive proof search algorithm implementing the tactics in Section 4. Safety versus termination is handled via a prover switch. When a proof is found, we check that the abduced predicates are satisfiable, using the method in [9]. The implementation of CABER amounts to about 3000 lines of OCaml code, excluding minor changes to CYCLIST.

# Program	LOC	Time (ms)	Search Depth	Defs. Class	Term. Proved
1 List traverse	3	20	3	A	✓
2 List insert	14	8	7	B	✓
3 List copy	12	0	8	B	✓
4 List append	10	12	5	B	✓
5 Delete last from list	16	12	9	B	✓
6 Filter list	21	48	11	C	✓
7 Dispose list	5	4	5	A	✓
8 Reverse list	7	8	7	A	✓
9 Cyclic list traverse	5	4	5	A	✓
10 Binary tree search	7	8	4	A	✓
11 Binary tree insert	18	4	7	B	✓
12 List of lists traverse	7	8	5	B	✓
13 Traverse even-length list	4	8	4	A	✓
14 Traverse odd-length list	4	4	4	A	✓
15 Ternary tree search	10	8	5	A	✓
16 Conditional diverge	3	4	3	B	×
17 Traverse list of trees	11	12	6	B	✓
18 Traverse tree of lists	17	68	7	A	✓
19 Traverse list twice	8	64	9	B	✓

# Program	LOC	Time (ms)	Search Depth	Defs. Class	Term. Proved
1 MUTANT test #1	4	4	3	A	✓
2 MUTANT test #2	6	8	5	A	✓
3 MUTANT test #3	6	8	7	A	✓
4 MUTANT test #4	11	52	8	C	✓
5 MUTANT test #5	16	16	12	B	✓
6 MUTANT test #6	6	4	5	A	✓
7 MUTANT test #7	8	4	7	A	✓
8 MUTANT test #8	30	×	×	×	×
9 MUTANT test #9	13	16	13	B	✓
10 MUTANT test #10	21	4	13	C	✓
11 MUTANT test #11	17	292	13	C	T/O

Fig. 4. Experimental results for the CABER tool. T/O indicates timeout (30s). See below for explanation of "Defs. Class" column.

Our experimental evaluation of CABER is summarised in Fig. 4. The test suite includes programs manipulating lists, trees, cyclic structures and higher-order structures like lists-of-lists and trees-of-lists. We also obtained under permission the programs used to test the MUTANT termination checker [4]. These are loops extracted from the Windows kernel that manipulate list-like structures of varying complexity. Our tests were performed on a x64 Linux system with an Intel i5 CPU at 3.4GHz and 4Gb of RAM. Run-times were generally very low, with no test taking more than 300 ms, apart from MUTANT test #11 whose termination proof times out. The definitions abduced by the safety- and termination-proving runs on each program were identical, except on test #16 and MUTANT test #11.

Evaluating the quality of abduced definitions is not trivial. In principle, definitions could be partially ordered by entailment (cf. [12]) but for our language this is known to be undecidable [1]. Instead, we manually classify solutions into three categories. A solution is rated "A" if it is syntactically equal to the standard precondition for that example, "B" if it is at least *provably* equal to the standard precondition, and "C" if it is strictly stronger than the standard precondition.

Out of 30 tests in total, 14 tests (47%) produce predicates rated "A", 11 tests (37%) produce predicates rated "B", and 4 tests (13%) produce predicates rated "C", with one test (3%) failing entirely. Categories A and B include cyclic list traversal (program 9 in Fig. 4), list of lists traversal (12), searching binary and ternary search trees (10, 15) and traversal of even- and odd-length lists (13, 14). The last four programs typically cannot be handled by (safety-checking) tools such as SpaceInvader and SLAyer. Test #6 and Mutant tests #4, #10, #11 produce C-rated definitions, and Mutant test #8 fails altogether. The common cause behind these (partial) failures is essentially the need for better abstraction and lemma speculation techniques, as discussed briefly in Section 4.

6 Related work

Our approach to the abduction of inductive definitions is close in spirit, if not so much in execution, to *inductive recursion synthesis* in AI (for a survey see [16]). The main novelties of our approach, compared to this technique, are: (a) that we abduce Hoare-style preconditions for imperative programs in separation logic, rather than inputs to functional programs in first-order logic; and (b) that we employ a cyclic proof search to abduce induction schemas.

Our abductive tactics for symbolic execution are similar to the approach taken in [12], which performs abduction for separation logic over a *fixed* signature of (higher-order) lists. In a different setting, Dillig et al. [15] abduce loop invariants as Boolean combinations of integer inequalities. In contrast, we directly abduce the inductive definitions of arbitrary data structures on-the-fly, by refining the meaning of predicate symbols during proof search.

There have also been a number of previous efforts to synthesise inductive predicates of separation logic for use in program analysis. Lee et al. present a shape analysis using an abstract domain of shape graphs based on a grammar of heaps [19]. The main limitation of the technique is the restriction of the inferred predicates to at most two parameters. Later, Berdine et al. developed a shape analysis employing a higher-order list predicate, from which various list-like data structures can be synthesised [2]. Again, the choice of abstract domain limits the class of predicates that can be discovered; for example, predicates defining trees cannot be expressed in this domain. Guo et al. leverage inductive recursion synthesis to infer inductive loop invariants in a shape analysis based on separation logic [17]. Chang and Rival propose a shape analysis whose abstract domain is parameterised by "invariant checkers", which are essentially inductive definitions provided by the user [13]. Finally, He et al. build on the bi-abductive techniques proposed in [12] to infer procedure specifications based on user-defined predicates [18]. The main differences between these works and our own is that they only consider safety and not termination; and they are generally based upon pre-defined recursion schemas or abstract domains, rather than inferring predicate definitions directly as we do. Guo et al. [17], based on inductive recursion synthesis techniques, is a notable exception to the latter rule.

Recently, Brockschmidt et al. developed a termination prover for Java programs based on term rewriting [7] that also performs some inference of heap predicates during analysis. In contrast to our work, their analysis assumes memory safety, while we guarantee it. Several authors have also considered the problem of inferring termination preconditions for integer programs (e.g., [6]). The heap is not usually considered, and the abduced preconditions are generally linear combinations of inequalities between integer expressions.

7 Conclusions and future work

In this paper we lay the foundations of a new technique, cyclic abduction, for inferring the inductive definitions of data structures manipulated by `while` pointer programs. This problem is far more challenging than the already difficult one of inferring pre/postconditions based on fixed predicates. Presently, our prototype tool Caber infers correct preconditions for small programs manipulating data structures such as lists, trees, cyclic lists and compositions of these. In particular, Caber abduces the correct termination preconditions, previously supplied by hand, for over 90% of the tests reported for Mutant in [4].

We note that cyclic abduction is subject to the same fundamental limitation as most static analyses: For computability reasons, there is no general solution to the abduction problem, and thus *we cannot do better than a heuristic search.*

The main avenue for future work is to improve the abduction heuristics in order to cover larger and more difficult examples than Caber is currently able to handle automatically. In particular, the `while` language in this paper does not feature procedure calls. There is no difficulty in extending the proof system in Section 3 to programs with procedures, adding postconditions to judgements to capture the effect of procedure calls. However, the abduction problem becomes much more difficult, as preconditions and postconditions must be abduced simultaneously. We know how to achieve this for some simple examples, but have not yet implemented it. For more complicated examples, we need to establish inductive entailments between formulas at procedure call sites, again highlighting the need for good lemma speculation techniques.

Current limitations of the implementation, which are however not fundamental, include: search space explosion in the presence of too many record fields and/or temporary variables in the program; the absence of heuristics for abducing information not explicitly manipulated by the program (e.g. numerical information [20]) and difficulty in abducing suitably segmented structures when several pointers traverse the same data structure.

Our approach is very "pure" in that the only source of information for abduction is the text of the program itself. Thus the recursion in the abduced predicates will typically reflect the manipulation of data structures by the program. In principle, one could compare abduced predicates to a "library" of known structures using a suitable inductive theorem prover for separation logic.

Although by no means a silver bullet, we believe that cyclic abduction offers a promising and natural approach to automatic specification inference.

References

1. T. Antonopoulos, N. Gorogiannis, C. Haase, M. Kanovich, and J. Ouaknine. Foundations for decision problems in separation logic with general inductive predicates. In *Proc. FoSSaCS 2014*. Springer, 2014.
2. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, and H. Yang. Shape analysis for composite data structures. In *Proc. CAV-19*. Springer, 2007.
3. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Proc. APLAS-3*. Springer, 2005.
4. J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV-18*. Springer, 2006.
5. J. Berdine, B. Cook, and S. Ishtiaq. Slayer: memory safety for systems-level code. In *Proc. CAV-23*. Springer, 2011.
6. M. Bozga, R. Iosif, and F. Konečný. Deciding conditional termination. In *Proc. TACAS-18*. Springer, 2012.
7. M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated termination proofs for Java programs with cyclic data. In *Proc. CAV-24*. Springer, 2012.
8. J. Brotherston, R. Bornat, and C. Calcagno. Cyclic proofs of program termination in separation logic. In *Proc. POPL-35*. ACM, 2008.
9. J. Brotherston, C. Fuhs, N. Gorogiannis, and J. Navarro Pérez. A decision procedure for satisfiability in separation logic with inductive predicates. In *Proceedings of CSL-LICS*. ACM, 2014. To appear.
10. J. Brotherston and N. Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. Technical Report RN/13/14, University College London, 2013.
11. J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In *Proc. APLAS-10*. Springer, 2012.
12. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6), December 2011.
13. B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *Proc. POPL-35*. ACM, 2008.
14. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
15. I. Dillig, T. Dillig, B. Li, and K. McMillan. Inductive invariant generation via abductive inference. In *Proceedings of OOPSLA*. ACM, 2013.
16. P. Flener and S. Yilmaz. Inductive synthesis of recursive logic programs: achievements and prospects. *The Journal of Logic Programming*, 41(2-3), 1999.
17. B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *Proc. PLDI-28*, June 2007.
18. G. He, S. Qin, W.-N. Chin, and F. Craciun. Automated specification discovery via user-defined predicates. In *Proc. ICFEM-15*. Springer, 2013.
19. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *Proc. ESOP-14*. Springer, 2005.
20. S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. POPL-37*. ACM, 2010.
21. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS-17*. IEEE Computer Society, 2002.
22. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *Proc. CAV-20*. Springer, 2008.
23. H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *Proc. FOSSACS-5*. Springer, 2002.