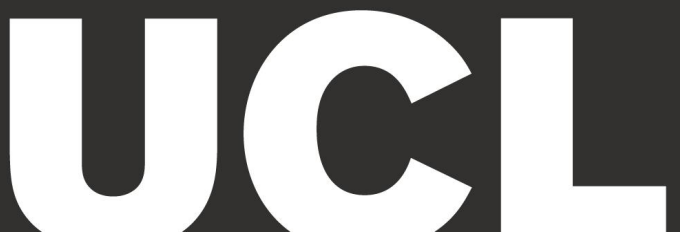
UCL DEPARTMENT OF
COMPUTER SCIENCE



Research Note

RN/13/14

Cyclic Abduction of Inductively Defined Safety and Termination Preconditions

July 12, 2013

James Brotherston Nikos Gorogiannis

Abstract

We describe a new method, called *cyclic abduction*, for automatically inferring safety and/or termination preconditions for heap-manipulating `while` programs, expressed as inductive definitions in separation logic. Cyclic abduction essentially works by searching for a *cyclic proof* of memory safety and/or termination, abducing definitional clauses of the precondition as necessary in order to advance the proof search process. This is achieved via a suite of heuristically guided automatic tactics.

We have implemented our cyclic abduction procedure as an automatic tool, Caber, that automatically infers the correct safety and termination preconditions for a range of common small programs manipulating lists and trees, and can also abduce the definitions of more exotic data structures such as cyclic or segmented lists, or trees of linked lists. To our knowledge, cyclic abduction is the first technique for automatically abducing such inductive definitions from pointer programs.

Cyclic Abduction of Inductively Defined Safety and Termination Preconditions

James Brotherston * Nikos Gorogiannis †
Dept. of Computer Science, University College London, UK

Abstract

We describe a new method, called *cyclic abduction*, for automatically inferring safety and/or termination preconditions for heap-manipulating `while` programs, expressed as inductive definitions in separation logic. Cyclic abduction essentially works by searching for a *cyclic proof* of memory safety and/or termination, abducing definitional clauses of the precondition as necessary in order to advance the proof search process. This is achieved via a suite of heuristically guided automatic tactics.

We have implemented our cyclic abduction procedure as an automatic tool, CABER, that automatically infers the correct safety and termination preconditions for a range of common small programs manipulating lists and trees, and can also abduce the definitions of more exotic data structures such as cyclic or segmented lists, or trees of linked lists. To our knowledge, cyclic abduction is the first technique for automatically abducing such inductive definitions from pointer programs.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Logics of programs, pre- and post-conditions

General Terms Verification, theory

Keywords abduction, shape analysis, cyclic proof, separation logic, inductive definitions, program verification

1. Introduction

In the last few years, a number of program analyses have appeared that employ *separation logic* [24] to establish safety and/or termination properties of heap-manipulating programs, in some cases extending to substantial code bases (see e.g. [25, 16, 21, 23]). All of these analyses rely on the use of *inductive predicates* to specify the shape of data structures stored in memory and manipulated by the program, e.g., lists or trees. However, such predicates must typically be hard-coded into the analysis (or in some cases must be provided by the user). This means that the analysis must either fail or ask the user for advice when it encounters a data structure not described by the provided definitions. For example, the well known SPACEINVADER [25] and SLAYER [5] analysers perform accurately on programs using combinations of linked lists, as found with device drivers, but report a false bug if they encounter a tree. Thus the ability to mechanically infer, or *abduce*, the inductive predicates needed to analyse individual procedures has the potential to greatly boost the automation of such verification tools.

In this paper, we consider the following form of the above synthesis or abduction problem: given a heap-manipulating `while` program P, can we define an inductive predicate in separation logic such that, given this predicate as a precondition,

- P runs without encountering a memory error (**safety**); or
- P eventually terminates safely (**termination**)?

These are highly non-trivial questions. On the one hand, the *weakest (liberal) precondition* (cf. Dijkstra [15]) can straightforwardly be extracted from P, but is useless for analysis: Deciding which program states satisfy this precondition is as hard as deciding from which states P runs safely and/or terminates! On the other hand, many preconditions (e.g. $\bot$) may be correct but too strong because they rule out the execution of some of the program. Ideally, we would like to find the weakest precondition that is expressible inside some fairly natural, "positive" class of inductive definitions and that ensures maximal program coverage. For 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 for `while` programs, expressed as inductive definitions in separation logic. Our approach is based upon heuristic proof search in a formal system of *cyclic proofs*, adapted from the cyclic termination proofs in [8].

Here, the core of a cyclic proof is a derivation tree built from *symbolic execution rules* capturing the effect of program commands (cf. [3]), and *logical rules* that manipulate the precondition. Cyclic proofs are so named because this derivation tree is allowed to contain *back-links* identifying leaves of the tree with arbitrary interior nodes, potentially creating cycles in the proof. Because such structures do not in general correspond to sound proofs, an additional (decidable) global soundness condition must be imposed upon these graphs to qualify them as genuine proofs. The difference between cyclic proofs of memory safety and those of termination is entirely a matter of the choice of soundness condition.

Given a program, our cyclic abduction procedure aims to simultaneously construct the inductive definition of a precondition in separation logic, under which this program is memory safe and/or terminating. 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:

* Research supported by an EPSRC Career Acceleration Fellowship.

† Research supported by EPSRC grant EP/H008373/1.

- 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 [10]. This tool essentially comprises a number of low-level *tactics* implementing the abduction principles above, heuristically guided by a high-level proof search strategy. CABER is able to automatically abduce safety and/or termination preconditions for a fairly wide variety of common small programs. The abduced inductive predicates include definitions of segmented, cyclic, nested and mutually defined data structures (over any number of parameters).

The remainder of this paper is structured as follows. Section 2 introduces the programming language and the fragment of separation logic preconditions we use to express program preconditions. Section 3 presents the formal system of cyclic safety/termination proofs on which our abduction technique is based. In Section 4 we present an overview of the cyclic abduction strategy, and then describe in detail the various abductive tactics from which it is built. Section 5 describes the implementation of our cyclic abduction tool CABER and our experimental evaluation of this tool. Section 6 examines related work and Section 7 concludes.

2. Programs and preconditions

In this section we present a basic language of `while` programs with heap pointers (similar to the language in [26]) and the fragment of separation logic we used to express program preconditions, based upon the *symbolic heaps* of [3].

We often use vector notation to abbreviate tuples or lists, e.g. $\mathbf{x}$ for $(x_1, \ldots, x_k)$. We write $\mathbf{x}_i$ for the ith element of the tuple $\mathbf{x}$.

2.1 Syntax of programs.

We assume infinite sets Var of *variables* and 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 field names and E over expressions:

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

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 nondeterministic condition. A *program* is a list of the field names of heap records followed by a command sequence:

$$\mathtt{fields}\ f_1, \ldots, f_k;\ C$$

2.2 Semantics of programs

We use a typical RAM model employing heaps of records. We fix a set Val of *values*, of which an infinite subset $\mathsf{Loc} \subset \mathsf{Val}$ are *locations*, i.e., the addresses of heap cells. We also assume a "nullary" value $nil \in \mathsf{Val} \setminus \mathsf{Loc}$ which is not 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 as usual: $[\![x]\!]s =_{\text{def}} s(x)$ for all $x \in \mathsf{Var}$, and $[\![\mathsf{nil}]\!]s =_{\text{def}} nil$. The semantics $[\![B]\!]s \subseteq \{\mathsf{true}, \mathsf{false}\}$ of a branching condition B in stack s is defined in the obvious way for equalities and disequalities, and $[\![\star]\!]s =_{\text{def}} \{\mathsf{true}, \mathsf{false}\}$.

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 $s[x \mapsto v]$ for 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 (Fig. 1). 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 $\mathtt{fields}\ f_1, \ldots, f_k;\ C$, we map the field names $f_1, \ldots, f_k$ onto elements of heap records by $\overline{f_j} =_{\text{def}} j$. The small-step semantics of our programs is then as usual given by a binary relation $\rightsquigarrow$ on states, presented in Figure 1. We write $\rightsquigarrow^n$ for the n-step variant of $\rightsquigarrow$, and $\rightsquigarrow^*$ for its reflexive-transitive closure. We say that a state (C, s, h) is *safe* if there is no computation $(C, s, h) \rightsquigarrow^* fault$, i.e. if running the program from (C, s, h) cannot result in a memory error. We say that (C, s, h) is *terminating* if it is safe and there is no infinite $\rightsquigarrow$-computation starting from (C, s, h).

Proposition 2.1 (Safety / termination monotonicity)**.** *If (C, s, h) is safe (resp. terminating) and $h \circ h'$ is defined then $(C, s, h \circ h')$ is also safe (terminating).*

Essentially, Proposition 2.1 holds for the same reason as in [26], i.e., extending the memory cannot lead to new memory faults. It *is* possible that extending the memory might lead to non-termination where previously there was a memory fault, but this is not problematic since terminating states are also required to be safe.

2.3 Syntax of logical preconditions

We express program preconditions using a simple fragment of separation logic with inductive definitions, based on the *symbolic heaps* of [3]. We assume an infinite set of *predicate symbols*, each with an associated arity.

Definition 2.2. *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 ranges over 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 the variable x by the expression E in the formula F. Substitution is extended pointwise to tuples; but when we write an expression of the form $F[E/\mathbf{x}_i]$, 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 2.3. An *inductive rule set* is a finite set of *inductive rules* each of the form $F \Rightarrow P(\mathbf{E})$, where F and $P(\mathbf{E})$ are formulas. We sometimes write an inductive rule as $F \overset{\mathbf{z}}{\Rightarrow} P(\mathbf{E})$, where $\mathbf{z}$ 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.

Inductive rules for a predicate P are understood as disjunctive clauses of the definition of P (cf. [8]).

2.4 Semantics of logical preconditions

The forcing relation $s, h \models_\Phi F$ for satisfaction of the formula F by the stack s and heap h under inductive rule set Φ is defined by

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

In order to remove the need for standard conjunction $\wedge$ in our logic fragment, it is convenient to interpret equalities and disequalities as holding in the empty heap. (This does not result in a loss of expressivity, since, e.g., arbitrary s, h satisfying $s(x) = s(y)$ can be represented as $x = y * \top$.

The semantics $[\![P]\!]^\Phi$ of the predicate P under Φ is, as usual, the least prefixed point of a monotone operator constructed from Φ:

Definition 2.4. Assume that Φ defines n predicates $P_1, \ldots, P_n$. Partition Φ into $\Phi_1, \ldots, \Phi_n$, where $\Phi_i \subseteq \Phi$ is the set of inductive rules of the form $F \Rightarrow P_i\mathbf{x}$. We let each Φ_i be indexed by j, and for each inductive rule $\Phi_{i,j}$ of the form $F \Rightarrow P_i\mathbf{x}$, we define the operator $\varphi_{i,j}$ by:

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

where $\models_{\mathbf{X}}$ is the satisfaction relation defined above, except that $[\![P_i]\!]^{\mathbf{X}} =_{\text{def}} X_i$, where $\mathbf{X} = (X_1, \ldots, X_n)$ is a tuple of pairs of the appropriate type. We then define $[\![\mathbf{P}]\!]^\Phi$ by:

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

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

An extremely useful fact about the fragment of separation logic we consider is that, for any given inductive rule set, it is decidable whether a formula in the fragment is *consistent*, i.e., whether or not the formula is semantically equivalent to $\bot$. The proof of this fact, which is very helpful in simplifying abduced preconditions as well as assessing their quality, appears in [9].

3. Formal cyclic safety/termination proofs

In this section we present a formal cyclic proof system for proving memory safety and/or termination of programs. The system here is adapted from the system of cyclic termination proofs in [8], with the following main differences: (i) we treat `while` programs rather than `goto` programs; and (ii) we are additionally able to consider memory safety only by imposing an alternative soundness condition on the proof graph.

A *proof judgement* is given by $F \vdash C$, where C is a command sequence and F is a formula.

Definition 3.1 (Validity)**.** Let Φ be an inductive rule set. The judgement $F \vdash C$ is *valid* (resp. *termination-valid*) w.r.t. Φ if $s, h \models_\Phi F$ implies (C, s, h) is safe (resp. terminating).

The proof rules for judgements are given in Fig. 2. Our notational convention is that 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 symbolic execution rules for commands are adaptations of standard rules for separation logic [3]. The logical rules manipulate the precondition without advancing the program. In particular, the rule (Frame) can be seen as a special case of the general *frame rule* of separation logic (see e.g. [26]), where the postcondition is omitted. Crucially, we include an unfolding (or "case analysis") rule for inductively defined predicates that unfolds a formula $P(\mathbf{E})$ in the conclusion according to the definition of P in an inductive definition set Φ:

Example 3.2. Define a unary inductive predicate $\mathsf{bt}(x)$, denoting those heaps structured as a binary tree with head pointer x, by

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

The unfolding rule for bt is the following:

$$\frac{\begin{array}{r} E = \mathsf{nil} * F \vdash C \\ E \neq \mathsf{nil} * E \mapsto (y', z') * \mathsf{bt}(y') * \mathsf{bt}(z') * F \vdash C \end{array}}{\mathsf{bt}(E) * F \vdash C}\ (\mathsf{bt})$$

where y' and z' are fresh variables.

Definition 3.3 (Pre-proof)**.** A *pre-proof* of $F \vdash C$ is a pair $(\mathcal{D}, \mathcal{L})$, where $\mathcal{D}$ is a finite derivation tree whose root is labelled by $F \vdash C$, 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 syntactically identical.

Any pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{L})$ can be understood as a graph, with an edge from the conclusion of any rule instance to each of its premises, by identifying each open leaf ℓ of $\mathcal{D}$ with $\mathcal{L}(\ell)$. A *path* in $\mathcal{P}$ is then understood in the obvious way.

Lemma 3.4. *Let Φ be an inductive rule set. Suppose the conclusion $F \vdash C$ of an instance of a rule R is invalid w.r.t. Φ, so that there exist a stack s and heap h such that $s, h \models_\Phi F$ but $(C, s, h) \rightsquigarrow^n$ fault.*

Then there is a premise $F' \vdash C'$ of this rule instance that is invalid w.r.t. Φ, i.e. there exist stack s' and heap h' 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 $m < n$.

Proof. A straightforward verification for each proof rule in Figure 2. The case of (Frame) relies upon Proposition 2.1. □

Definition 3.5 (Cyclic proof)**.** A pre-proof $\mathcal{P}$ is a *cyclic (safety) proof* if for every infinite path in $\mathcal{P}$, there are infinitely many symbolic execution rule applications along this path.

Theorem 3.6. *For any inductive rule set Φ, if there is a cyclic safety proof of $F \vdash C$, then $F \vdash C$ is valid w.r.t Φ.*

Proof. Suppose for contradiction that $F \vdash C$ has a cyclic safety proof $\mathcal{P}$ but is invalid. Using Lemma 3.4, we can construct an infinite path $(F_k \vdash C_k)_{k \geq 0}$ in $\mathcal{P}$, and an 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 the path $(F_k \vdash C_k)_{k \geq 0}$. This implies the existence of an infinite descending chain of natural numbers, which is impossible. Hence $F \vdash C$ must be valid. □

We can consider cyclic proofs of termination rather than memory safety simply by replacing the soundness condition of 3.5 with the trace-based soundness condition of [8], which essentially demands that some inductive predicate be unfolded infinitely often along every infinite path in the pre-proof. Thus, by a simple adaptation of the soundness result in [8], we also have:

Theorem 3.7. *For any inductive rule set Φ, if there is a cyclic termination proof of $F \vdash C$ — i.e., a pre-proof of $F \vdash C$ satisfying the soundness condition of [8] — then $F \vdash C$ is termination-valid w.r.t Φ.*

$$\frac{}{(x := E;\, C, s, h) \rightsquigarrow (C, s[x \mapsto [\![E]\!]s], h)}$$

$$\frac{[\![y]\!]s \in \mathit{dom}(h)}{(x := y.f;\, C, s, h) \rightsquigarrow (C, s[x \mapsto h([\![y]\!]s).\overline{f}], h)} \qquad \frac{[\![x]\!]s \in \mathit{dom}(h)}{(x.f := E;\, C, s, h) \rightsquigarrow (C, s, h[[\![x]\!]s.\overline{f} \mapsto [\![E]\!]s])}$$

$$\frac{\ell \in \mathsf{Loc} \setminus \mathit{dom}(h) \quad v_1, \ldots, v_k \in \mathsf{Val}}{(x := \mathtt{new}();\, C, s, h) \rightsquigarrow (C, s[x \mapsto \ell], h[\ell \mapsto (v_1, \ldots, v_k)])} \qquad \frac{[\![x]\!]s \in \mathit{dom}(h)}{(\mathtt{free}(x);\, C, s, h) \rightsquigarrow (C, s, (h \upharpoonright (\mathit{dom}(h) \setminus \{[\![x]\!]s\}))}$$

$$\frac{[\![y]\!]s \notin \mathit{dom}(h)}{(x := y.f;\, C, s, h) \rightsquigarrow \mathit{fault}} \qquad \frac{[\![x]\!]s \notin \mathit{dom}(h)}{(x.f := E;\, C, s, h) \rightsquigarrow \mathit{fault}} \qquad \frac{[\![x]\!]s \notin \mathit{dom}(h)}{(\mathtt{free}(x);\, C, s, h) \rightsquigarrow \mathit{fault}}$$

$$\frac{\mathsf{true} \in [\![B]\!]s}{(\mathtt{if}\, B\, \mathtt{then}\, C\, \mathtt{else}\, C'\, \mathtt{fi};\, C'', s, h) \rightsquigarrow (C;\, C'', s, h)} \qquad \frac{\mathsf{false} \in [\![B]\!]s}{(\mathtt{if}\, B\, \mathtt{then}\, C\, \mathtt{else}\, C'\, \mathtt{fi};\, C'', s, h) \rightsquigarrow (C';\, C'', s, h)}$$

$$\frac{\mathsf{true} \in [\![B]\!]s}{(\mathtt{while}\, B\, \mathtt{do}\, C\, \mathtt{od};\, C', s, h) \rightsquigarrow (C;\, \mathtt{while}\, B\, \mathtt{do}\, C\, \mathtt{od};\, C', s, h)} \qquad \frac{\mathsf{false} \in [\![B]\!]s}{(\mathtt{while}\, B\, \mathtt{do}\, C\, \mathtt{od};\, C', s, h) \rightsquigarrow (C', s, h)}$$

Figure 1. Small-step operational semantics of programs, given by the binary relation $\rightsquigarrow$ over program states.

Symbolic execution rules:

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

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

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

$$\frac{B * F \vdash C;\, C''}{B * F \vdash \mathtt{if}\, B\, \mathtt{then}\, C\, \mathtt{else}\, C'\, \mathtt{fi};\, C''} \qquad \frac{\overline{B} * F \vdash C';\, C''}{\overline{B} * F \vdash \mathtt{if}\, B\, \mathtt{then}\, C\, \mathtt{else}\, C'\, \mathtt{fi};\, C''} \qquad \frac{F \vdash C;\, C'' \qquad F \vdash C';\, C''}{F \vdash \mathtt{if} \star \mathtt{then}\, C\, \mathtt{else}\, C'\, \mathtt{fi};\, C''}$$

$$\frac{B * F \vdash C;\, \mathtt{while}\, B\, \mathtt{do}\, C\, \mathtt{od};\, C'}{B * F \vdash \mathtt{while}\, B\, \mathtt{do}\, C\, \mathtt{od};\, C'} \qquad \frac{\overline{B} * F \vdash C'}{\overline{B} * F \vdash \mathtt{while}\, B\, \mathtt{do}\, C\, \mathtt{od};\, C'} \qquad \frac{F \vdash C;\, \mathtt{while}\, B\, \mathtt{do}\, C\, \mathtt{od};\, C' \qquad F \vdash C'}{F \vdash \mathtt{while} \star \mathtt{do}\, C\, \mathtt{od};\, C'}$$

Logical rules:

$$\frac{F \vdash C}{F * G \vdash C}\ (\text{Frame}) \qquad \frac{F \vdash C}{F[E/x] \vdash C}\ \begin{array}{c}x \text{ not a program} \\ \text{variable}\end{array}\ (\text{Subst}) \qquad \frac{F' \vdash C}{F \vdash C}\ F \equiv F'\ (\text{Equiv}) \qquad \frac{G' * F \vdash C}{G * F \vdash C}\ G \vdash G'\ (\text{Cut})$$

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

Predicate unfolding rule:

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

Figure 2. 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, abductive *tactics* used in implementing this strategy. Then, in Section 5, we describe how these tactics are implemented and combined into an abductive proof search algorithm.

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 fresh predicate symbol.

Our strategy for finding such a Φ is to *search for a cyclic proof* of the judgement $P(\mathbf{x}) \vdash C$ (with respect to safety or termination as desired). Almost invariably, this search process will become "stuck" at some point, e.g., because the precondition does not contain enough information for a symbolic execution rule to fire. When this happens, we may abduce one or more new inductive rules to enable the search to proceed. In the following, we 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 command appearing at the current subgoal.*

That is, if we can close the current branch somehow then we do so, otherwise we try to advance the program along this branch via symbolic execution. (Note that when we attempt to form a back-link in the proof, violations of the soundness condition can be detected automatically by appealing to a model checker.) However, it often happens that neither of these is possible. When this situation occurs, we are allowed to use the logical rules and/or to abduce inductive rules in order to enable the proof search to make progress, as described by our second principle.

Principle 2. *We may abduce inductive rules and/or deploy the logical and predicate unfolding rules in the following circumstances:*

(a) *in order to symbolically execute a branching command;*
(b) *in order to symbolically execute a dereferencing command;*
(c) *in order to form a back-link;*
(d) *as part of a generalisation attempt.*

When we abduce inductive rules for a predicate occurring in the current subgoal, we always immediately apply the unfolding rule for that predicate to the subgoal.

We may abduce inductive rules only *for predicate symbols that are currently undefined.*

We explain in the following subsections exactly how inductive rules are abduced in the situations (a), (b), and (c) described by Principle 2 (generalisation (d) being covered in Section 4.6). The reason for restricting abduction to undefined predicates is that adding inductive rules to Φ_P adds new premises to the unfolding rule (P), rendering any existing applications of (P) in the derivation unsound. We will indicate a combined abduction-and-unfolding step in our derivations with $\mathcal{A}(P)$.

Our final principle sets out the general idea behind performing *generalisation* in a proof search.

Principle 3. *Before applying symbolic execution to a* `while` *loop, one should normally attempt to generalise the precondition F appearing at the subgoal in question. That is to say, we should attempt to find some weaker precondition 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 this more general F'.*

Generalisation is well known to be a necessary (and difficult) step in inductive theorem proving [11], and unsurprisingly it shows up in our abduction proofs also. Section 4.6 presents tactics for generalising the precondition when we encounter a `while` loop.

4.2 Tactics: an overview

A *tactic* in our setting is a more general version of a tactic as it usually appears in automated theorem proving. Here, a tactic is essentially a general proof transformer: it updates the current *proof state* by applying some (possibly nondeterministic) combination of *atomic inferences*. The differences between our setting and that of a traditional theorem prover reside in the underlying notions of "proof state" and "atomic inference". First, since we employ a cyclic notion of proof, with back-links joining leaves to arbitrary proof nodes, it is not sufficient to restrict our attention to the current subgoal only: our proof state must reflect the entire pre-proof, and forming a back-link between nodes must count as a valid atomic inference step. Second, since we are allowed to abduce new inductive rules in the proof search, the current inductive rule set must form part of the proof state, and adding new inductive rules to this set must also count as a valid inference step. (As usual, all proof rules of our system, in Figure 2, are also valid atomic inferences.)

Taking the above into account, our formal proof states are triples comprised of the following elements:

$\mathcal{P}$**:** A partial pre-proof, representing the portion of proof constructed so far. By "partial", we mean that some of the leaves of $\mathcal{P}$ may be open; we call these the *open subgoals* of $\mathcal{P}$.

Φ**:** An inductive rule set, containing all the inductive rules abduced so far in the proof search.

ℓ**:** A distinguished open leaf of the pre-proof $\mathcal{P}$, representing the subgoal on which the next step of the proof search is to operate.

A *tactic* is then simply a transformer on proof states.

Example 4.1. In order to demonstrate how our abductive tactics are applied in practice, we will frequently refer to the following running example in the remainder of this section. Figure 3 shows an abductive cyclic proof of the following program:

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

The proof in Figure 3 abduces the binary tree predicate from Example 3.2 as a safety and termination precondition for this program. For ease of presentation, we number each individual command (sequence) in the program, and we write the judgements in the proof by referring to these indices rather than the commands themselves.

4.3 Abductive tactic for deterministic branching commands

Our proof rules for deterministic `if` and `while` commands (Fig. 2) mirror the operational semantics in that they require the precondition to determine the status of the branching condition. We introduce an abductive tactic, `abduce_branch`, enabling us to proceed 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

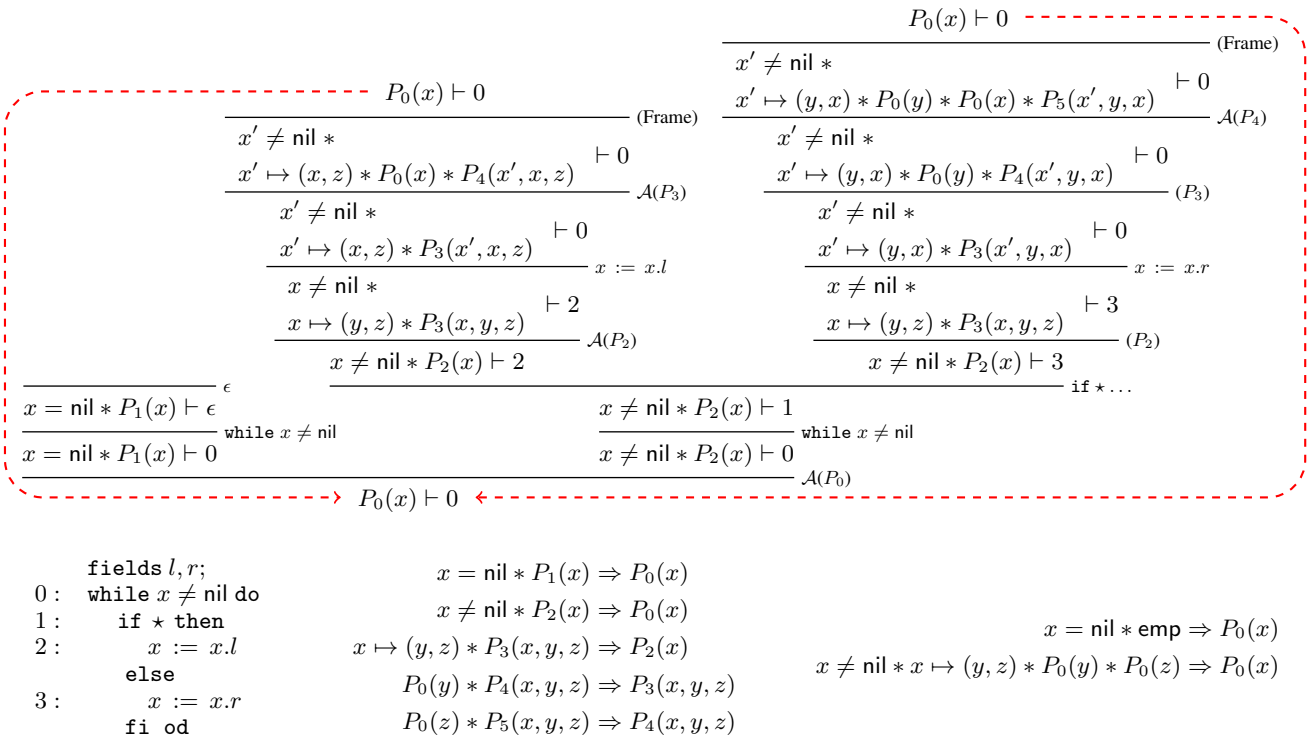


Figure 3. Top: abductive proof for a binary tree search program (shown bottom left). Bottom center: inductive rules abduced during the proof. Bottom right: simplified inductive rules.

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 the predicate symbol P is currently undefined in Φ, and such that 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 $y = \mathbf{E}_j$ (where $k \neq j$). Then, `abduce_branch` adds the following inductive rules for P to Φ:

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

where P', P'' are fresh predicate symbols and $\mathbf{z}$ is a tuple of appropriately many arbitrary variables. Next, the tactic unfolds the indicated occurrence of $P(\mathbf{E})$ in ℓ as follows:

$$\frac{B * F * P'(\mathbf{E}) \vdash \mathcal{C} \qquad \overline{B} * F * P''(\mathbf{E}) \vdash \mathcal{C}}{F * P(\mathbf{E}) \vdash \mathcal{C}}\, \mathcal{A}(P)$$

Finally, the tactic applies the appropriate symbolic execution rule for $\mathcal{C}$ to each of the new subgoals (note that this step is now guaranteed to succeed).

The choice of subformula $P(\mathbf{E})$ and indices k, j is nondeterministic; `abduce_branch` returns the results of all such choices.

Example 4.2. When searching for a proof of the program in Example 4.1, our initial goal is of the form

$$P_0(x) \vdash \texttt{while}\ x \neq \mathsf{nil}\ \texttt{do} \ldots$$

where P_0 is undefined. As the symbolic execution of the `while` command fails, we can call `abduce_branch` which selects $P_0(x)$ as the only viable subformula of the goal and abduces the following inductive rules for P_0:

$$\begin{array}{rcl} x = \mathsf{nil} * P_1(x) & \Rightarrow & P_0(x) \\ x \neq \mathsf{nil} * P_2(x) & \Rightarrow & P_0(x) \end{array}$$

The tactic then unfolds $P_0(x)$ and applies symbolic execution for the `while` command in the two resulting subgoals. This is shown in Fig. 3 as the application of $\mathcal{A}(P_0)$ followed by the normal symbolic executions for `while`).

4.4 Abductive tactic for dereferencing assignments

The symbolic execution rules for commands that dereference a memory address (Fig. 2) require the precondition to guarantee that this address is indeed allocated (otherwise the program will encounter a memory fault, as per the operational semantics in Fig. 1). Here we present an abductive tactic, `abduce_deref`, that enables the symbolic execution of such commands to proceed by abducing the allocation of the appropriate address.

Formally, suppose `abduce_deref` is applied to the proof state $(\mathcal{P}, \Phi, \ell)$, where the command $\mathcal{C}$ in the judgement labelling the current subgoal ℓ is of the form $\texttt{free}(x); C$, $x.f := E; C$ or $y := x.f; C$. First, `abduce_deref` non-deterministically selects a subformula of the form $P(\mathbf{E})$ appearing at ℓ, where P is currently undefined in Φ, and such that x occurs in the tuple $\mathbf{E}$. Thus, we may write the judgement appearing at ℓ as $F * P(\mathbf{E}) \vdash \mathcal{C}$ where $x = \mathbf{E}_k$ (for some k). Then `abduce_branch` adds the following inductive rule for P 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, arbitrary variables such that $|\mathbf{x}| = |\mathbf{E}|$, and $|\mathbf{y}|$ is the number of fields in the program. The

tactic then unfolds the selected occurrence of $P(\mathbf{E})$ in ℓ as follows:

$$\frac{F * x \mapsto \mathbf{y}' * P'(\mathbf{E} \sqcup \mathbf{y}') \vdash \mathcal{C}}{F * P(\mathbf{E}) \vdash \mathcal{C}}\,\mathcal{A}(P)$$

where $\mathbf{y}'$ is a tuple of $|\mathbf{y}|$ fresh variables. Finally, `abduce_deref` updates $\mathcal{P}$ again by applying the symbolic execution rule for $\mathcal{C}$ (which is now guaranteed to succeed).

Similarly to `abduce_branch` in the previous subsection, the selection of subformula $P(\mathbf{E})$ and index k is essentially nondeterministic, so `abduce_deref` returns the list of proof states corresponding to the results of all such selections.

Example 4.3. When searching for an abductive proof of the binary tree traversal program in Example 4.1, after applying the `abduce_branch` tactic we arrive at the following subgoal on the middle branch (see Figure 3):

$$x \neq \mathsf{nil} * P_2(x) \vdash x := x.l; \ldots$$

where P_2 is undefined (at the current proof state). We cannot apply the symbolic execution rule for the dereferencing assignment $x := x.l$, as it would require a formula of the form $x \mapsto (E, E')$ to appear in the precondition. Thus we call `abduce_deref`, which selects the only suitable formula $P_2(x)$ in the precondition and abduces the following inductive rule for P_2:

$$x \mapsto (y, z) * P_3(x, y, z) \Rightarrow P_2(x).$$

The tactic then unfolds $P_2(x)$ and applies the symbolic execution rule for the command $x := x.l$ as shown in Figure 3.

A similar situation occurs when we reach the similar subgoal on the right hand branch:

$$x \neq \mathsf{nil} * P_2(x) \vdash x := x.r$$

However, there is one crucial difference: having already abduced a definition for P_2 when tackling the left-hand subgoal, it is no longer undefined. In such situations, when no suitable undefined predicate is available, `abduce_deref` attempts to apply symbolic execution for $x := x.r$ by first unfolding a previously defined predicate instance, in this case $P_2(x)$. (Calcagno et al. [12] employ a similar abduction tactic for their fixed inductive predicates.)

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 node in the current preproof labelled by a judgement of the form $F' \vdash C$, provided that:

1. the judgement $F \vdash C$ is derivable from $F' \vdash C$, so that the source and target of the back-link can be made syntactically identical as required by Definition 3.3; and
2. the addition of this back-link does not create an infinite path violating the soundness condition on cyclic proofs (for safety or termination, as appropriate).

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 the proof state $(\mathcal{P}, \Phi, \ell)$. First, the tactic selects a node n of $\mathcal{P}$, distinct from ℓ, such that the commands in the judgements labelling n and ℓ are identical. Then it tries to manipulate ℓ using logical rules so as to obtain a precondition identical to the one at n. 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 a copy of n.

We write ℓ as $F_1 * P(\mathbf{E}) \vdash C$, where P is undefined in Φ, and n as $F_2 \vdash C$. Then `abduce_backlink` will abduce an inductive rule of the following 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, where we view spatial 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 as follows:

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

The role of the fresh predicate P' is to allow a part of the heap to be abduced at a later stage, possibly on a completely different branch of the proof (see the second part of Example 4.4).

As with our other abductive tactics, `abduce_backlink` acts nondeterministically: there may be several viable choices of undefined predicate P, "matching formula" F' and substitution θ.

Two further caveats apply. First, according to Principle 1, we must ensure that the proposed back-link does not violate the relevant soundness condition on cyclic proofs. We verify this is the case by calling a model checker when `abduce_backlink` attempts to form a back-link. Second, `abduce_backlink` is also allowed to try unfolding a defined predicate in the subgoal ℓ if no undefined predicates are available (similarly to `abduce_deref`).

Example 4.4. In Example 4.1, upon reaching the subgoal

$$x' \neq \mathsf{nil} * x' \mapsto (x, z) * P_3(x', x, z) \vdash 0$$

on the middle branch of the proof in Fig. 3, we notice that the `while` command in the subgoal matches that in earlier nodes, so we decide to try to form a back-link. Thus we call `abduce_backlink`, which selects the only suitable formula $P_3(x', x, z)$ in the subgoal and abduces the following inductive rule for P_3:

$$P_0(y) * P_4(x, y, z) \Rightarrow P_3(x, y, z)$$

The tactic then unfolds $P_3(x', x, z)$, applies (Frame) and forms a back-link from the resulting subgoal to the root node of the proof as shown in Fig. 3. Note that, in this case, the use of substitution is not required and the chosen "matching formula" F' is $P_0(y)$.

When we arrive at a similar subgoal on the rightmost branch,

$$x' \neq \mathsf{nil} * x' \mapsto (y, x) * P_3(x', y, x) \vdash 0$$

the situation is slightly different because P_3 is no longer undefined. In this situation, `abduce_backlink` first unfolds P_3 according to its existing definition, and then proceeds in the usual way by abducing a suitable inductive rule for the undefined P_4. (It is for this reason that it was essential to abduce the fresh predicate component $P_4(x, y, z)$ of the previous inductive rule for P_3, even though on the middle branch this component was discarded by (Frame).)

We observe that `abduce_backlink` is "forgetful" in that it uses the frame rule to discard parts of the precondition. Another alternative would instead be to use the (Cut) rule to establish logical entailments enabling the current subgoal to be identified with the desired target node. In some cases, such reasoning is necessary in order to abduce the "ideal" predicate as precondition. We implemented an alternative back-linking tactic that attempts to apply (Cut) in order to unify the current subgoal with a chosen target node. The resulting logical entailments are passed to the separation

logic instantiation of CYCLIST (described in [10]) which attempts to discharge them. Unfortunately, because such entailments might be invalid, and often require inductive reasoning to establish even in the case that they are valid, such calls to CYCLIST are extremely expensive (many such calls might be made during a proof attempt and any of them might time-out). Thus, at the time of writing, we have found the computational cost of this tactic to be prohibitive.

4.6 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 specified target judgement often fail because the judgement specifies a too-precise relationship between program variables which is not preserved by the body of the loop. One possible solution, which is typical of inductive theorem proving in general, is to *generalise* the precondition upon encountering the `while` loop so as to "forget" such variable relationships. Here we present a tactic, `ex_gen`, that 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 $E = E'$ or $y \mapsto \mathbf{E}$ by a fresh (implicitly existentially quantified) variable w. (This step implicitly uses an application of (Cut), and is easily seen to be sound.) This tactic also exhibits nondeterminism in that it may generalise over any subset of variables modified by the loop body, and present in the precondition.

Example 4.5. Figure 4 shows an abductive cyclic proof of the following program, which traverses a list of lists:

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

When the proof search for this program reaches the subgoal

$$x \neq \mathsf{nil} * x \mapsto (y, z) * P_3(x, y, z) \vdash \texttt{while}\ z \neq \mathsf{nil}\ \texttt{do}\ \ z := z.down\ \ \texttt{od}$$

we call the tactic `ex_gen`. Since z is modified by the body of the `while` loop, and the precondition contains the subformula $x \mapsto (y, z)$, the tactic replaces the z in this subformula by the fresh variable w, as shown in the application of EX-GEN in Fig. 4. Observe that this generalisation step is needed in order to form the back-link on the right-hand branch (as $x \mapsto (y, z)$ does not hold after execution of the loop body, but $x \mapsto (y, w)$ does).

Other, more complex types of generalisation are also possible (and indeed might be needed for some proofs). For example, we implemented a "segmenting" generalisation tactic, `seg_gen`, that identifies a "cursor" variable in the loop body and abduces inductive rules for an undefined predicate in the precondition, based on forming segments from this predicate with this cursor as a midpoint. Figure 5 shows an application of this tactic in the context of an abductive cyclic proof of the following program, which traverses a list from y after initially assigning x to y:

$$\begin{array}{l}\texttt{fields}\ next\,;\\ y := x;\ \texttt{while}\ y \neq \mathsf{nil}\ \texttt{do}\ \ y := y.next\ \ \texttt{od}\end{array}$$

After symbolically executing $y := x$ we arrive at the subgoal:

$$x = y * P_0(x) \vdash \texttt{while}\ y \neq \mathsf{nil}\ \texttt{do}\ y := y.next\ \texttt{od}$$

Our tactic `seg_gen` identifies y as a potential cursor variable, since it is modified by the loop body and appears in the loop guard, and abduces the following inductive rule for the undefined P_0:

$$P_1(x, x) * P_1(x, \mathsf{nil}) \Rightarrow P_0(x)$$

The tactic then unfolds $P_0(x)$ and rewrites using the equality $x = y$ as shown by the application of SEG-GEN in Fig. 5. This enables us to later form a back-link with the more general $P_1(x, y) * P_1(y, \mathsf{nil})$ as the precondition which describes the general invariant of the `while` loop (i.e., a list segment from x to y together with a list segment from y to nil). However, this requires us to establish the entailment $P_1(x, y') * y' \mapsto y \vdash P_1(x, y)$, as shown by the application of (Cut) on branch, which requires inductive reasoning. In order to find the proof in Fig. 5 automatically in reasonable time, we require a better *lemma speculation* mechanism, as discussed in the previous subsection.

4.7 Simplification of inductive rule sets

When an abductive proof search using the tactics above succeeds, the returned set of abduced inductive rules will typically be too large and unwieldy for human consumption. We can apply some fairly straightforward simplifications to our abduced rule sets in order to make them more easily readable. Our simplification method, outlined below, is used to produce the simplified inductive rules shown in Figures 3–5.

1. Elimination of undefined predicates. When our search tactics abduce inductive rules, these rules always contain an undefined component involving a fresh predicate symbol. Thus, the final set of abduced inductive rules will typically contain several undefined predicates. Clearly, such predicates may be interpreted however we wish; thus we interpret them all as the empty heap emp.

For example, during the abductive proof in Figure 3 we abduce the inductive rule $P_0(z) * P_5(x, y, z) \Rightarrow P_4(x, y, z)$, where P_5 is still undefined at the end of the proof. Thus we instantiate $P_5(x, y, z)$ to emp whereby, taking into account that $F * \mathsf{emp} \equiv F$, we obtain the simplified rule $P_0(z) \Rightarrow P_4(x, y, z)$.

2. In-lining. In most cases, our abductive tactics abduce a single inductive rule for an undefined predicate. Consequently, the final inductive rule set typically contains many mutually recursive predicates, many of which can be straightforwardly eliminating by in-lining. First, we order the predicate symbols by the order in which their definitions were abduced during the proof search. Then, for any predicate Q with only a single inductive rule, whenever $Q > P$ according to our ordering, we replace all subformulas of the form $Q(\mathbf{E})$ in the inductive rules for P by the definition of $Q(\mathbf{E})$.

For example, in the abductive proof in Figure 3, the predicate P_2 is defined by a single inductive rule,

$$x \mapsto (y, z) * P_3(x, y, z) \Rightarrow P_2(x)$$

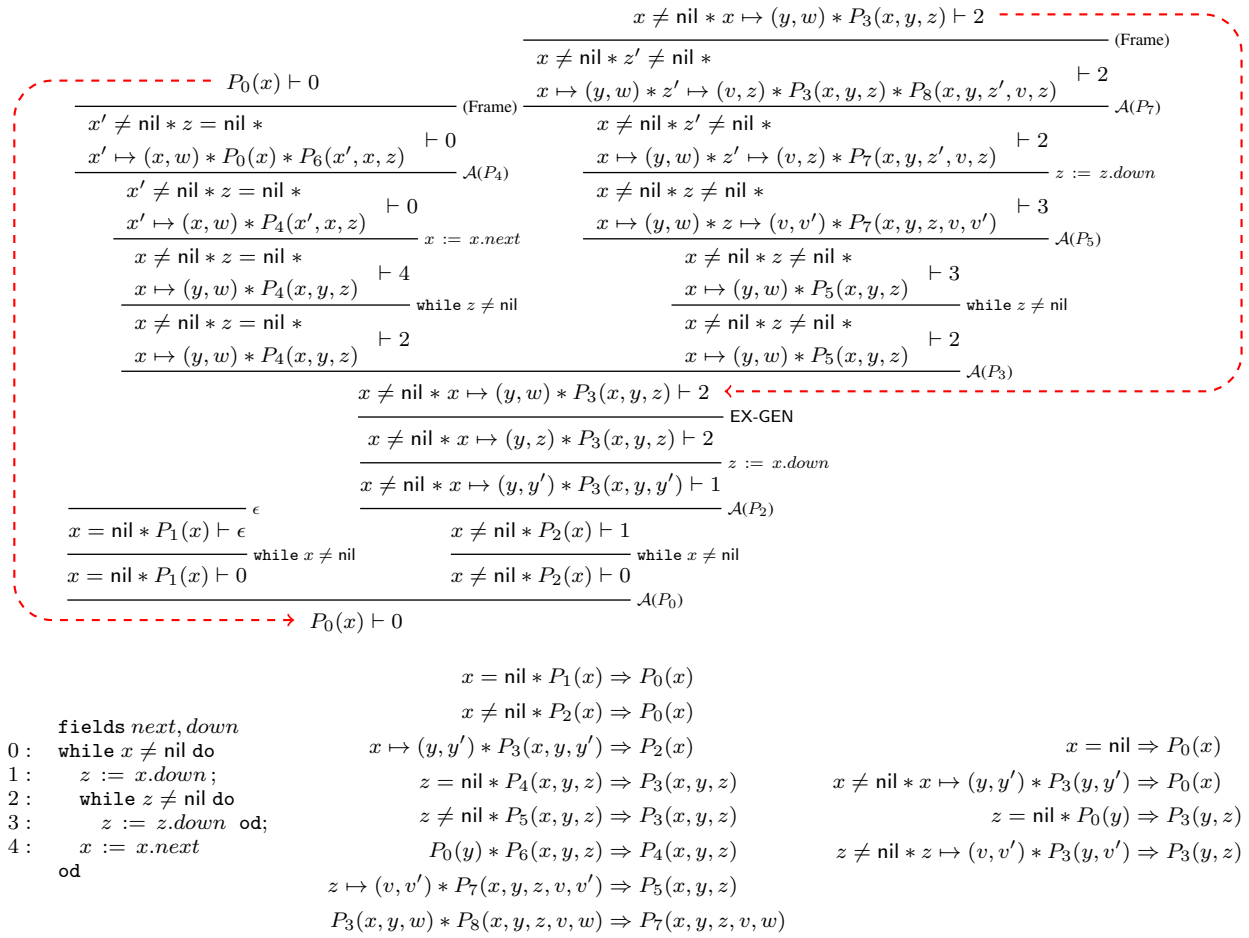
and $P_2 > P_0$ according to the abduction order. Thus, we replace $P_2(x)$ by its definition in the second inductive rule for P_0 to obtain the in-lined inductive rule

$$x \neq \mathsf{nil} * x \mapsto (y, z) * P_3(x, y, z) \Rightarrow P_0(x)$$

Repeating this process for the predicates P_3 and P_4 produces the inductive rule shown in Figure 3,

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

3. Elimination of redundant parameters. A parameter of a predicate P is said to be *redundant* (in the inductive rule set Φ) if it does not occur on the left hand side of any inductive rule in Φ_P, except possibly as the same parameter of P itself. Our final simplification step is to remove any such redundant parameters from the inductive rule set.

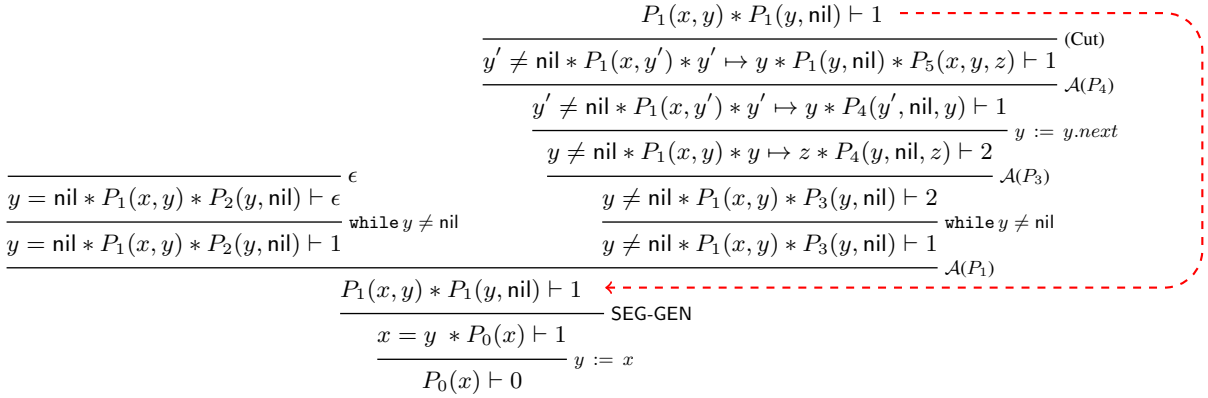


```
    fields next, down
0 : while x ≠ nil do
1 :   z := x.down ;
2 :   while z ≠ nil do
3 :     z := z.down od;
4 :   x := x.next
    od
```

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

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

Figure 4. Top: abductive proof for list-of-lists traversal. Bottom, left to right: program; predicates found; simplified predicates.



```
    fields next ;
0 : y := x;
1 : while y ≠ nil do
2 :   y := y.next
    od
```

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

$$P_1(x, x) * P_1(x, \mathsf{nil}) \Rightarrow P_0(x)$$
$$x = y \ * \mathsf{emp} \Rightarrow P_1(x, y)$$
$$x \neq y \ * x \mapsto z * P_1(z, y) \Rightarrow P_1(x, y)$$

Figure 5. Top: abductive proof for list traversal program (bottom left) employing segment generalisation. Bottom centre: inductive rules abduced during the proof. Bottom right: the simplified predicates.

For example, given an inductive rule of the form

$$y \neq \mathsf{nil} * x \mapsto x' * P(y, x', z) \Rightarrow P(x, y, z)$$

the third parameter (z) of P is redundant, and thus we eliminate this parameter to obtain the simplified

$$y \neq \mathsf{nil} * x \mapsto x' * P(y, x') \Rightarrow P(x, y).$$

5. Implementation and evaluation

In this section, we describe our automatic tool, CABER (from "Cyclic ABducER"), that implements our cyclic abduction strategy and the individual tactics described in the previous section, and we report on its performance on a variety of example programs.

5.1 Implementation platform

CABER is built on top of the open-source theorem prover CYCLIST, a generic framework (written in OCaml and C++) for constructing cyclic theorem provers [10]. One of its existing instantiations is a prover for a simpler version of the system in [8] which, given a precondition in separation logic with (fixed) inductive predicates, searches for termination proofs for pointer programs. CABER extends this prover with the abductive tactics described in Sec. 4, and modifies the proof search algorithm in CYCLIST in the following ways: First, we incorporated the set of inductive definitions into the proof search state, allowing it to be modified during proof search. This also enables CABER to back-track over different choices of inductive rules, as required by our non-deterministic abductive tactics. Second, when the CYCLIST prover forms a back-link it checks that the soundness condition has not been violated (Def. 3.5) and, if so back-tracks. In order to check for safety and, optionally, for termination, CABER provides a different soundness check for each one as detailed in Section 3. Third, our abduction method requires that when a closed proof is found, the inductive rules found are checked for satisfiability through the algorithm detailed in [9], and if found to be unsatisfiable to trigger back-tracking in the proof search. The implementation of CABER amounts to about 3000 lines of OCaml code, excluding other minor changes to CYCLIST.

5.2 Proof search algorithm

Algorithm 6 gives an overview of the abductive proof search, suppressing the boundary between theorem prover and tactics. The inputs to procedure `prove` are: $\mathcal{P}$, a (partial) cyclic pre-proof; Φ, a set of inductive rules; and S, the list of open subgoals in $\mathcal{P}$. A successful call `prove`($\mathcal{P}$, Φ, S) returns $(\mathcal{P}', \Phi')$ where $\mathcal{P}'$ is a proof that extends $\mathcal{P}$, and which is valid w.r.t. the inductive rule set $\Phi' \supseteq \Phi$. To search for a proof of a program C with precondition $P\mathbf{x}$, we would make the call `prove`($\mathcal{P}_0$, $\emptyset$, $[n]$) where $\mathcal{P}_0 =_{\text{def}} \{n : P\mathbf{x} \vdash C\}$ is the partial proof with just the sequent n.

Procedures `abduce_backlink` and `abduce_symex` implement the tactics described in Sec. 4.5 and Secs. 4.3/4.4 (`abduce_symex` combines tactics `abduce_branch` and `abduce_deref` into one), inventing new inductive rules and subsequently unfolding a predicate P in the current subgoal. They all return a list of pairs of a proof and an inductive rule set, representing alternative ways to progress the search. Procedure `ex_gen`, which implements the generalisation tactic described in Sec. 4.6, does not define new predicates, so returns a list of pairs of a proof and a new subgoal. Procedure `sound` uses CYCLIST's model checker to ensure that the introduction of a back-link has not violated the soundness condition (Sec. 3). Procedure `sat` uses the algorithm described in [9] to decide if a set of inductive rules is satisfiable.

5.3 Experimental evaluation

We ran CABER on a set of test programs, listed in Fig. 7. The test suite includes programs manipulating lists, trees, cyclic structures and higher-order structures, like lists-of-lists and trees-of-lists. Additionally, we obtained under permission the programs used to test the MUTANT termination checker [4]. The MUTANT programs are loops extracted from the Windows kernel that manipulate list-like structures of varying complexity. We run the abductive prover twice on each program, using the safety soundness condition and the termination soundness condition respectively.

```
sym_exec(P: proof, Φ: inductive rules, S: open goals of P) :
begin
    (g, T) := (head(S), tail(S)) if rule for cmd(g) cannot fire at g
    then return nil (P', S') := symex_rule(P, g) return prove(P',
    Φ, concat(S', T))

prove(P: proof, Φ: inductive rules, S: open goals of P) :
begin
    if S = [] then
        if sat(Φ) then return (P, Φ) else return nil
    (g, T) := (head(S), tail(S)) if axiom applies to g then
        return prove(close_by_axiom(P, g), Φ, T)
    foreach g' ∈ P such that cmd(g)=cmd(g') do
        foreach (P', Φ') ∈ abduce_backlink(P, Φ, g, g') do
            if sound(P') then
                r := prove(P', Φ', T)  if r ≠ nil then return r
    r := sym_exec(P, Φ, S)  if r ≠ nil then return r  foreach
    (P', Φ', g') ∈ abduce_symex(P, Φ, g) do
        r := sym_exec(P', Φ', concat(g',T))  if r ≠ nil then return
        r
    if cmd(g) = while then
        foreach (P', g') ∈ ex_gen(P, g) do
            r := prove(P', Φ, concat(g', T))  if r ≠ nil then return
            r
    return nil
```

Figure 6. Abductive proof search algorithm. Procedure `cmd`(g) returns the command in goal g; `symex_rule`($\mathcal{P}$, g) applies the appropriate symbolic execution rule to goal g and returns a proof that extends $\mathcal{P}$ with that application, plus a new list of subgoals.

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 syntactically identical except when proving termination fails entirely (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 unlikely even to be decidable, let alone practical for experimental evaluation. Another possibility is to use code coverage, or one of its variants, as a measure of solution quality; however, because we insist every predicate in our definitions is satisfiable (cf. Sections 2.4 and 5) it is easy to see that coverage will always be 100% provided a proof is found. To overcome these difficulties we classify solutions manually in three categories. Category A consists of definitions that are identical to the standard one, modulo predicate name substitution. As an example the predicate below defines a nil-terminated singly-linked list, but the name P is arbitrary. This is the output obtained from tests #1, #7–#9 and MUTANT tests #1–#3, #6 and #7.

$$\begin{aligned} x = \mathsf{nil} &\Rightarrow P(x) \\ x \neq \mathsf{nil} * x \mapsto x' * P(x') &\Rightarrow P(x) \end{aligned} \qquad \text{(L)}$$

Category B consists of definitions provably equivalent to the standard one, but possibly unfolded, or having a significantly different recursion scheme. For example, the following definition is clearly equivalent to (L) and derives by unfolding (L) once. These predi-

#	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

Figure 7. Experimental results for the CABER tool. T/O indicates timeout (30s). See Sec. 5.3 for explanation of the Definitions Class column.

cate were ontained from tests #2–#4 and MUTANT test #5.

$$\begin{array}{r} x = \mathsf{nil} \Rightarrow Q(x) \\ x \neq \mathsf{nil} * x \mapsto x' * Q(x') \Rightarrow Q(x) \\ x = \mathsf{nil} \Rightarrow P(x) \\ x \neq \mathsf{nil} * x \mapsto x' * Q(x') \Rightarrow P(x) \end{array} \quad \text{(LU)}$$

Finally, category C consists of definitions that are strictly stronger than the standard one, typically because they allow recursion up to a fixed finite level. For example, the following definition is stronger than (L)/(LU) in that it only allows lists of length < 2. These predicates, here folded for brevity, were obtained from test #6.

$$\begin{array}{r} x = \mathsf{nil} \Rightarrow Q(x) \\ x \neq \mathsf{nil} * x \mapsto x' * Q(x') * Q(x') \Rightarrow Q(x) \\ x = \mathsf{nil} \Rightarrow P(x) \\ x \neq \mathsf{nil} * x \mapsto x' * Q(x') \Rightarrow P(x) \end{array} \quad \text{(L1)}$$

Arguably, the difference between categories A and B is a little arbitrary. For instance, better simplification techniques (Section 4.7) would turn B instances to A instances. What is more important is the distribution of tests between A/B and C.

Overall, 14 out of 30 tests (47%) produce predicates syntactically isomorphic to the standard ones (A), 11 tests (37%) produce semantically equivalent, but not syntactically isomorphic predicates (B), 4 tests (13%) produce predicates that are strictly stronger than the standard ones (C), and one test (3%) fails entirely. Notable cases in categories A and B are traversing a cyclic list (program 9 in Fig. 7), traversing a list of lists (12), searching binary and ternary search trees (10, 15) and traversing 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 too strong definitions, and MUTANT tests #8 fails altogether. The cause behind these are due to the need for better abstraction techniques, as discussed in Section 4.6. Manual inspection reveals that the tactic `seg_gen` would produce a sufficiently strong precondition for a proof by hand. However, such a proof requires the use of the (Cut) rule to prove non-trivial inductive theorems such as the following.

$$\mathsf{lseg}(x, y) * y \mapsto z * \mathsf{ls}(z) \vdash \mathsf{lseg}(x, z) * \mathsf{ls}(z)$$

As briefly discussed in Sec. 4.6 we implemented this approach by calling the separation logic entailment prover in CYCLIST from within the abductive proof search, but found it to be computationally prohibitive. An improved lemma speculation facility could help significantly by avoiding many unsuccessful calls to the entailment checker; we plan to investigate this direction in future work.

6. Related work

Our proof-theoretic approach to the abduction of inductive definitions is close in spirit to *inductive recursion synthesis* in AI (for a survey see [17]). The main novelties of our approach, compared to existing inductive recursion synthesis techniques, are: (a) that we abduce program preconditions in separation logic, while the techniques surveyed in [17] are confined to the setting of first-order logic; and (b) that we employ cyclic proof to abduce induction schemas, whereas inductive recursion synthesis typically relies upon fixed recursion schemas.

There have also been a number of 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 [18]. However, the inductive recursion synthesis and recurrence detection algorithms employed in [18] are highly complex. Finally, 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].

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 does not find termination preconditions but assumes a fixed precondition and uses it to infer further predicates; more importantly, their approach assumes memory safety, while we guarantee it. Several authors have also considered the problem of *conditional* termination of integer programs (e.g., [14, 6]). Here, the heap is not usually considered — an exception being Moy and Marché [22] — and the abduced preconditions are linear combinations of (in)equalities between integer expressions, rather than inductively defined predicates, as in our case.

7. Conclusions and future work

In this paper, we introduce an entirely new technique, cyclic abduction, for automatically abducing the inductive definitions of data structures that serve as safety and/or termination preconditions for heap-manipulating `while` programs. To the best of our knowledge, ours is the first known technique for automatically inferring such inductive definitions without recoursing to user-provided definitions or hard-coded abstract predicate domains. At the time of writing, our automatic cyclic abduction tool CABER is capable of inferring the correct definitions for many common small programs manipulating lists and trees, and in some cases can also infer infer the definitions of more exotic data structures such as cyclic lists, segmented data structures, or a tree of linked lists. In particular, CABER abduces the correct termination precondition for over 90% of the successful tests reported for the MUTANT tool in [4], where the precondition previously had to be supplied by hand.

Cyclic abduction is based upon heuristically-guided proof search in a formal cyclic proof theory adapted from the one in [8]. For fundamental computability reasons, no general solutions are possible, thus we cannot do any better than a heuristic search; our method will not always succeed in discovering the ideal preconditions (or perhaps any preconditions at all). On the other hand, we believe our formal logical setting is advantageous in at least three respects. Firstly, we can search either for safety or termination preconditions in the *same* formal system, simply by choosing an appropriate soundness condition over pre-proofs. Secondly, it is straightforward to develop and test new heuristics and tactics by implementing (perhaps nondeterministic) combinations of inference rules and back-links. Thirdly, when our proof search succeeds, it yields both a suitable precondition for the program *and* an explicit proof of safety and/or termination under this precondition.

In this paper we consider a basic `while` language that is quite adequate for writing small programs but less so for large ones since it does not include procedure calls. We can straightforwardly extend our proof system to consider postconditions as well as preconditions, whereby the standard Hoare logic proof rule for procedure calls (see e.g. [12]) can straightforwardly be included. However, to abduce the preconditions of procedure calls successfully we would need to establish inductive entailments between formulas at procedure call sites. This is a challenging problem likely requiring the development of good heuristics for *lemma speculation*, as also arises in general inductive theorem proving [11].

Naturally, there are some limitations on what can be done using cyclic abduction. Most conspicuously, the only source of information for abduction is the program itself, which means that the recursion in the abduced predicates will typically reflect the traversal of data structures in the program. Similarly, we are currently unable to abduce numerical information about data structures as employed e.g. in [21], such as the length of a list, unless that information is explicitly manipulated by the program. However, we can attempt to establish the equivalence of an abduced predicate to one already known by calling a suitable inductive theorem prover for separation logic (e.g. CYCLIST [10]). Further current limitations (that however might be overcome by better heuristics) include: a blow up in the search space in the presence of too many record fields and/or temporary variables in the program; and difficulty in correctly abducing suitably segmented predicates when several pointer variables are used to traverse the same data structure.

Finally, we note that the problem of inferring predicates for use in program analysis is by no means limited to the setting we consider here. For example, [20] attempts to infer instrumentation predicates for use in the TVLA system, while [1] attempts predicate abstraction over large C programs. A possible avenue for future work would be to investigate whether cyclic abduction as presented here can be usefully adapted to such settings.

References

[1] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *International Journal on Software Tools for Technology Transfer*, 5, 2003.

[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. Technical Report RN/13/15, Univ. College London, 2013.

[10] J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In *Proc. APLAS-10*, LNCS. Springer, 2012.

[11] A. Bundy. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning*, chapter 13, . Elsevier Science, 2001.

[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] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *Proc. CAV-20*. Springer, 2008.

[15] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[16] D. Distefano and M. Parkinson. jStar: Towards practical verification for Java. In *Proc. OOPSLA-23*. ACM, 2008.

[17] P. Flener and S. Yilmaz. Inductive synthesis of recursive logic programs: achievements and prospects. *The Journal of Logic Programming*, 41(2-3), 1999.

[18] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *Proc. PLDI-28*, June 2007.

[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] A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement via inductive learning. In *Proc. CAV-17*. Springer, 2005.

[21] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. POPL-37*. ACM, 2010.

[22] Y. Moy and C. Marché. Modular inference of subprogram contracts for safety checking. *Journal of Symbolic Computation*, 45, 2010.

[23] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *Proc. POPL-37*. ACM, 2010.

[24] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS-17*. IEEE Computer Society, 2002.

[25] 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.

[26] H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *Proc. FOSSACS-5*. Springer, 2002.