Automatic Cyclic Termination Proofs for Recursive Procedures in Separation Logic

Reuben N. S. Rowe James Brotherston

Department of Computer Science
University College London, UK
{r.rowe,j.brotherston}@ucl.ac.uk

Abstract

We describe a formal verification framework and tool implementation, based upon cyclic proofs, for certifying the safe termination of imperative pointer programs with recursive procedures. Our assertions are *symbolic heaps* in separation logic with user defined inductive predicates; we employ *explicit approximations* of these predicates as our termination measures. This enables us to extend cyclic proof to programs with procedures by relating these measures across the pre- and postconditions of procedure calls.

We provide an implementation of our formal proof system in the CYCLIST theorem proving framework, and evaluate its performance on a range of examples drawn from the literature on program termination. Our implementation extends the current state-of-the-art in cyclic proof-based program verification, enabling automatic termination proofs of a larger set of programs than previously possible.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Theory, Verification

Keywords Automated proof search, Cyclic proof, Explicit approximation, Imperative programming, Proof Certificates, Separation logic, Termination

1. Introduction

Establishing that a given program eventually *terminates* was identified as a fundamental problem in computer science by Turing well before the actual physical development of stored-program computers [34]. It has been understood at least since Floyd's landmark paper [19] that proving termination depends on identifying a suitable well-founded *termination measure* (a.k.a. "ranking function") that decreases regularly during every execution. Then, since the measure cannot decrease infinitely often, there can be no infinite execution of the program.

For example, consider the following C procedure for traversing a null-terminated linked list in memory pointed to by `x`:

```
void TraverseList(Node *x) {
 if x != NULL {
  y := x->nxt; TraverseList(y); TraverseList(y);}}
```

In the case that the linked list is empty (`x == NULL`), termination is immediate. For non-empty lists, intuitively we can infer termination for two reasons: the first recursive call acts on the local variable `y` which references a smaller linked list (i.e. the tail of the original one); and the second recursive call also acts on a smaller list since the first recursive call does not increase the size of the list referenced by `y`.

In this paper, we provide a formalism for proving the termination of pointer programs with procedures (such as the one above), and present a tool which automatically discovers such proofs. The core of our approach is a *cyclic* Hoare-style proof system for total correctness, with pre- and postconditions written in the well-known *symbolic heap* fragment of separation logic [27], where user-defined inductive predicates are used to describe data structures in memory (see [5, 15]). Our cyclic proof system extends an earlier cyclic proof system for simple `while` programs [12] in which termination measures are always obtained from (combinations of) semantic *approximations* of the inductive predicates in the proof. The addition of procedures, however, requires non-trivial extensions to the proof system in order to track how these approximations are affected by procedure calls. For this, our formalism uses explicit ordinal variables, e.g., specifying `TraverseList` as follows:

$$\{\mathsf{List}_\alpha(x)\}\ \texttt{TraverseList(x)}\ \{\mathsf{List}_\alpha(x)\} \qquad (1)$$

Here $\mathsf{List}(x)$ is an inductively defined predicate of separation logic describing null-terminated linked lists with head pointer

x, and α is an ordinal variable referring to an (under-) approximation of this predicate, which is unchanged by the procedure; in this context α can intuitively be read simply as the length of the list. In general we might write, e.g., $\{P_\alpha(x)\}\,\texttt{myproc(x)}\,\{\exists\beta\,.\,\beta < \alpha.\ Q_\beta(x)\}$ if it happens that `myproc` actually decreases the measure referred to by α.

Proofs in our system are cyclic proofs, which are standard finite derivation trees, but with some leaves possibly closed by back-links to identical interior nodes. To ensure soundness of such proofs, a *global soundness condition* (which is decidable by automata-theoretic methods) is imposed on the proof structure[1]. In our case, this condition amounts to the fact that some ordinal termination measure decreases infinitely often on every infinite path in the proof structure. In this sense, our technique is related to *size-change termination* [24], which attempts to extract similarly well-founded measures directly from program data.

We provide an implementation of our proof system inside the generic cyclic theorem prover CYCLIST [14]. This results in a fully-automatic proof search procedure for our system; the cyclic termination proofs it produces are formal mathematical objects, which can be seen as independently verifiable *certificates* of program termination. We employ key theorem-proving features such as *frame inference*, well known to be needed for interprocedural proof in separation logic [5]. We evaluate our implementation on a number of examples taken from the Termination Problems Database [33], demonstrating that our implementation is competitive with other termination tools (e.g. AProVE [20] and HIPTNT+ [23]).

Our approach has two major advantages. Firstly, it is *compositional*, meaning that proofs for procedures may be re-used as part of larger proofs and thus need only be verified once. In addition, building the analysis around separation logic allows taking advantange of the compositionality afforded by its well known *frame* rule [35]. Namely, having proved $\{A\}\,P_1\,\{B\}$ and $\{A'\}\,P_2\,\{B'\}$ valid for program fragments P_1 and P_2, we can then compose the specifications of the individual fragments using the *separating conjunction* $*$ of separation logic to derive a valid specification for the sequential composition $\{A * A'\}\,P_1; P_2\,\{B * B'\}$. Secondly, although our implementation currently requires individual procedures to be annotated with pre- and postcondition summaries, we do *not* require the user to provide global termination measures: these are guaranteed by the global soundness check. Thus our approach provides more automation than might initially be apparent. Moreover, it seems quite plausible that such procedure specifications might be inferred *automatically* (see e.g. [10, 16, 22] for results in specification inference), and so we believe that our framework has the potential for full automation in the future.

[1] See [8, 28, 30] for early examples of cyclic proof systems, and [9, 13] for their application to separation logic.

The remainder of this paper is structured as follows. Section 2 informally explains our cyclic proof-based technique using two motivating examples. Sections 3 and 4 then formally present, respectively, our programming and assertion languages, and our cyclic proof system for total correctness. In Sections 5 and 6 we discuss the details of our implementation and our experimental results. Section 7 gives a comparison with related work, and we conclude in Section 8.

2. Motivating Examples

We now illustrate our approach using two examples. To ease the presentation of these examples, we here use standard logical syntax for assertions rather than the more succinct formal syntax we define in Section 3.

Example 1. We describe how our approach verifies that the `TraverseList` procedure above satisfies the specification given in Eq. (1), where the List predicate is defined inductively in separation logic by:

$$(x = \mathsf{nil} \wedge \mathsf{emp}) \vee (x \neq \mathsf{nil} \wedge x \mapsto y * \mathsf{List}(y)) \Rightarrow \mathsf{List}(x)$$

Here, emp denotes the empty piece of memory; $x \mapsto y$ denotes a single memory cell at the location referenced by x and containing the value y; and $A * B$ denotes a piece of memory that can be split into two *disjoint* parts satisfying A and B respectively.

The key feature of Eq. (1) for proving termination of `TraverseList` is that the instances of the List predicate in both the pre- and the postcondition are labelled by the same α. We may think of this as encapsulating the fact that the procedure does not change the *size* of the list referenced by the parameter.[2] This allows us to infer that the second recursive call to `TraverseList` indeed acts on an input which is smaller than that of its parent call, even though this data has previously been passed through another procedure call. In this example, for simplicity, the two procedure calls in the body of `TraverseList` are recursive; however, if we consider replacing the first recursive call by a call to some arbitrary independent procedure, then it is clear the knowledge that that procedure does not increase the size of the list is absolutely necessary.

This reasoning is encapsulated in the cyclic proof shown in Fig. 1. The overall structure of the proof is dictated by that of the code, as each command is executed symbolically: the initial procedure call is unfolded (proc); the analysis branches at the conditional statement (if-det); and the dereferencing of `x` is handled by the (read) rule. The recursion is naturally

[2] The correspondence is not direct however: the 'size' measured by predicate labels is *how long* it takes the fixed point semantics, described in Section 3 below, to generate portions of memory as a model (i.e. the number of 'unfoldings' of an inductive definition). Thus, while it broadly corresponds to *one* notion of the size of a data-structure (as predicate definitions are monotone), it is not exactly the same concept.

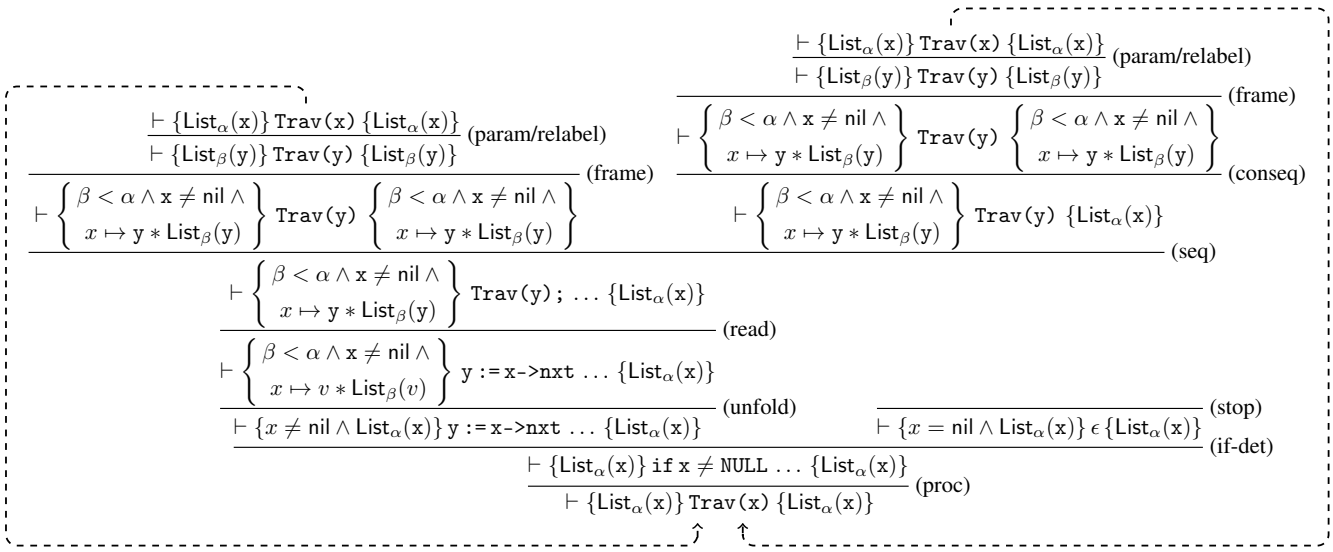

Figure 1: Cyclic proof of correctness for the `TraverseList` procedure (here abbreviated `Trav`).

handled by substitution and *back-linking*, possibly facilitated by a framing step. Notice that in order to obtain a precondition enabling the symbolic execution of the pointer dereference (the precondition must explicitly specify, via a subformula $\mathtt{x} \mapsto y$, that the program variable $\mathtt{x}$ is allocated), the definition of List must be unfolded. We elide the case that leads to inconsistency (we cannot have an empty list in the 'then' branch of the conditional). Note that predicate unfolding in our system generates fresh ordinal variables for each recursive predicate instance, and introduces constraints ($\beta < \alpha$) relating these to the parent instance. These are justified by the semantics of our inductive definitions, defined in Section 3.

It is necessary to check that the proof is globally sound, i.e. that some well-founded measure decreases infinitely often along each infinite path (following symbolic execution) from conclusion through to premise. For this, we track the progression of some ordinal variable through the precondition of each sequent along the path. In the proof in Fig. 1 this 'trace' *progresses* from tracking α to tracking β when the List predicate is unfolded, and the constraint $\beta < \alpha$ corresponds to a decrease in the measure being tracked. The substitution steps immediately preceding the back-links allow renaming of the ordinal variable being tracked. The sequential composition rule (seq) creates different infinite paths through the cyclic proof, but notice that we may trace the ordinal variable β along both branches. Importantly, the ability to trace β along the right-hand branch however is due to its presence in the postcondition of the left-hand premise, deriving ultimately from the specification in Eq. (1).

We draw attention to the compositional nature of our approach. The proof in Fig. 1 is self-contained and thus may constitute, as is, a sub-component of a larger (possibly also cyclic) proof. Moreover, the well-foundedness of any infinite path in the larger proof which enters this sub-proof is implied by the global soundness of the sub-proof alone, and thus does not need to be re-checked. We demonstrate this compositionality with our second example

Example 2. Consider a `while` loop that calls a procedure (suggestively named `Remove`) which reduces a linked list in some way (and does nothing if the list is already empty), for which we assume a specification (with corresponding cyclic termination proof $\mathcal{P}$) given by the Hoare triple:

$$\begin{array}{l}\{\mathtt{x} \mapsto l * \mathsf{List}_\alpha(l)\} \\ \quad \mathtt{Remove(x)} \\ \{\mathtt{x} \mapsto \mathsf{nil} \vee (\exists \beta, l.\ \beta < \alpha \wedge \mathtt{x} \mapsto l * \mathsf{List}_\beta(l))\}\end{array} \tag{2}$$

Here, the argument `x` to this procedure is a *pointer* to the head of the list, rather than a reference to the head of the list itself. The constraint in the postcondition of this specification allows for a cyclic proof of the following Hoare triple, which asserts termination of a while loop that repeatedly invokes `Remove` until the list is empty:

$$\begin{array}{l}\{\mathtt{x} \mapsto \mathtt{list} * \mathsf{List}_\alpha(\mathtt{list})\} \\ \quad \mathtt{while\ list!=NULL\ \{\ Remove(x);\ list:=x\text{->}val;\ \}} \\ \{\mathtt{x} \mapsto \mathsf{nil}\}\end{array}$$

A cyclic termination proof of this triple is shown in Fig. 2, where ψ denotes the postcondition in Eq. (2). Notice that, similarly to procedures, the cyclic proof system treats loops by *unrolling* them. Just as recursion is handled compositionally by forming cycles (back-linking), iteration is handled the same way. The precondition at the back-link point can be seen as an invariant for the loop or recursion, and global soundness of the cycle amounts to the existence of a termination variant.

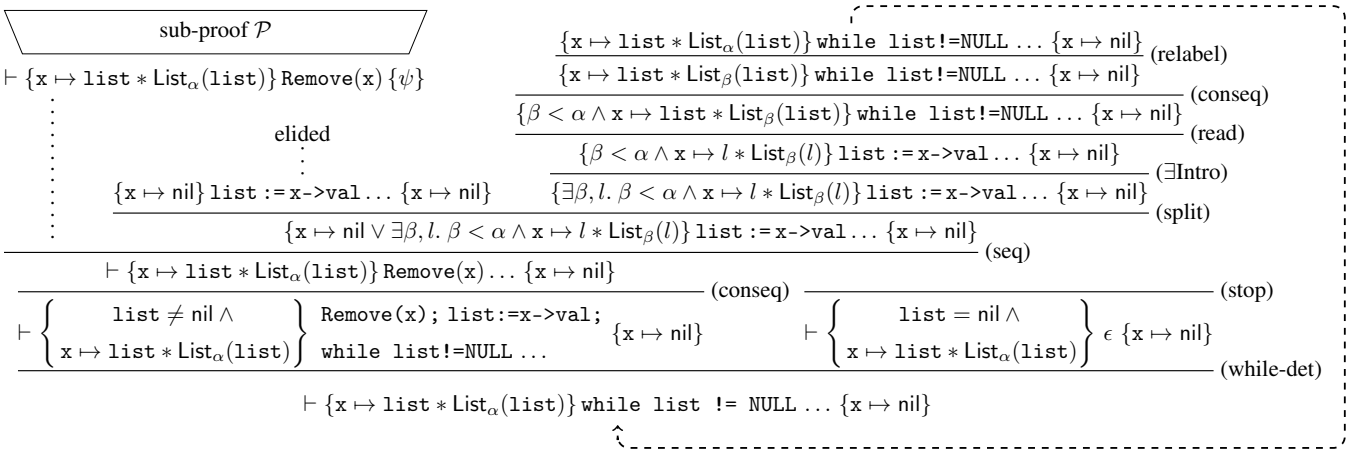

Figure 2: Cyclic proof of a `while` loop invoking the `Remove` procedure

In Fig. 2, after symbolically executing the call to the `Remove` procedure, the (split) rule analyses the continuation of the loop body in the context of each disjunct of ψ separately. We elide one of the branches however since it is a straightforward acyclic symbolic execution proof. The other branch, which we show, forms a cycle in the proof. Following the infinite path generated by this cycle, we may track the ordinal variable α which progresses to β across the (seq) rule, and then eventually is renamed back to α before cycling around the loop again. The well-foundedness of this path derives from the effect of the `Remove` procedure, encoded in the specification of Eq. (2), which decreases the measure tracked by this trace.

We point out that, given a (candidate) cyclic proof, it is possible to determine automatically whether its ordinal variables trace some decreasing measure. This means our technique *infers* (certain kinds of) termination measures. It might seem that the constraints in a procedure specification (which we currently rely on the programmer to provide) specify a measure for its termination. However, this is not precisely the case; while ordinal variable constraints can be used to describe how our termination measures are affected by a procedure call or code fragment, this is *not* equivalent to specifying the measures themselves.

3. Programs and Separation Logic Assertions

We now formally describe the language treated by our system, essentially `while` programs with procedures, and our language of separation logic assertions, based on the well known *symbolic heap* fragment introduced in [4, 5], incorporating user-defined inductive predicates as in [10, 15]. Moreover, we allow predicate instances to be annotated with labels, interpreting these as *approximations* of the predicate; similar notions are employed in [18, 31].

Notational Conventions. We use vector notation for sequences, writing $\boldsymbol{s}_i$ for the i^{th} element of $\boldsymbol{s}$, ϵ for the empty sequence, $\boldsymbol{s_1} \cdot \boldsymbol{s_2}$ for sequence concatenation, and $|\boldsymbol{s}|$ for the number of elements in $\boldsymbol{s}$. We sometimes abuse notation by using $\boldsymbol{s}$ to refer to the set of elements occurring in $\boldsymbol{s}$. We also write $f[\boldsymbol{s} \mapsto \boldsymbol{t}]$ for the update of function f by the partial function $(\boldsymbol{s} \mapsto \boldsymbol{t})$, and $f =_{\setminus \boldsymbol{s}} f'$ to denote equality of f and f' up to all points in $\boldsymbol{s}$.

3.1 Syntax and Semantics of Programs

We use x, y, etc. to range over program variables and f to range over identifiers for fields of records in heap memory. Programs consist of sequences of procedures, declared by $p(\boldsymbol{x})\ \{C\}$ where p ranges over procedure names, C is a command sequence called the body of the procedure (denoted by $\mathit{body}(p)$), and the (distinct) variables $\boldsymbol{x}$ are the parameters (denoted by $\mathit{params}(p)$). We write $\mathit{locals}(p)$ for the local variables of p, the set of program variables occurring in C which are not parameters (i.e. $\mathit{vars}(C) \setminus \mathit{params}(p)$). The atomic commands are: assignment ($x := E$); field read ($y := x.f$); field write ($x.f := E$); memory record allocation ($x := \mathtt{new}(\boldsymbol{f})$) and deallocation ($\mathtt{free}(x)$); or procedure call ($p(\boldsymbol{E})$), where expressions E are either a variable or the constant nil. We also include branching commands ($\mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\ \mathtt{fi}$) and loops ($\mathtt{while}\ B\ \mathtt{do}\ C\ \mathtt{od}$), where branching conditions B are given by (dis)equalities on expressions or non-determinism ($\star$). We write $\overline{B}$ to indicate the negation of a branching condition, and $\mathit{mod}(C)$ for the set of all variables whose values are modified by C (i.e. all x such that C contains a command of the form $x := E$, $x := y.f$, or $x := \mathtt{new}(\boldsymbol{f})$). Formally, we define command sequences C by the following grammar:

$$
\begin{aligned}
B &::= \star \mid E = E \mid E \neq E \\
C &::= \epsilon \mid x := E; C \mid y := x.f; C \mid x.f := E; C \mid \\
&\qquad x := \mathtt{new}(\boldsymbol{f}); C \mid \mathtt{free}(x); C \mid p(\boldsymbol{E}); C \mid
\end{aligned}
$$

$$\texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C\ \texttt{fi}; C \mid \texttt{while}\ B\ \texttt{do}\ C\ \texttt{od}; C$$

The semantics of our language is given by a RAM model of heaps of field-labelled records. *Heap cells* are finite partial functions from field identifiers to values in a set Val, and *heaps* are finite partial functions from heap locations (taken from a set Loc) to heap cells. Since the values manipulated by programs are intended to be pointers, Val contains the set Loc and a distinguished value $nil \notin \mathsf{Loc}$. e denotes the empty heap (undefined everywhere), and $h_1 \circ h_2$ denotes the union of heaps h_1 and h_2 if their domains are *disjoint* (and is otherwise undefined). *Stores* map variables to values, and we define the interpretation $[\![E]\!]s$ of an expression E in a store s by $[\![\mathsf{nil}]\!]s = nil$ and $[\![x]\!]s = s(x)$, extending pointwise to sequences of expressions $\boldsymbol{E}$. The interpretation of a branching condition is a set of stores; we define $s \in [\![\star]\!]$ for all s, $s \in [\![E_1 = E_2]\!]$ if and only if $[\![E_1]\!]s = [\![E_2]\!]s$, and $s \in [\![E_1 \neq E_2]\!]$ if and only if $[\![E_1]\!]s \neq [\![E_2]\!]s$. Furthermore, if a program does not contain a declaration for procedure p then we say that p is *undefined*.

Each procedure call is executed within its own stack frame (C, s), where C is the remainder of the procedure body to be executed and s is a store recording the values of the procedure's formal arguments and local variables. A stack Ξ is a sequence of stack frames. We model a program's state as a configuration κ, which is either a pair (Ξ, h) of a non-empty stack Ξ and a heap h, or the special configuration *fault*. We define the semantics of programs by a standard small-step relation $\rightsquigarrow$ on configurations defined as the smallest relation satisfying the rules in Fig. 3. We say that (s, h) is a *final state for* a configuration κ whenever $\kappa \rightsquigarrow^n ((\epsilon, s), h)$ for some n, and that κ is *terminating* whenever there are no infinite execution sequences $\kappa \rightsquigarrow^\infty$, and no n such that $\kappa \rightsquigarrow^n \mathit{fault}$. Thus our notion of termination is also *memory-safe*.

3.2 Syntax and Semantics of Assertions

We let α, β, etc. range over a set of *predicate labels*, whose intended domain of interpretation is the ordinals. We use P to range over a set Pred of *predicate names*, which each have an associated arity denoted by $\mathsf{ar}(P)$.

Definition 1 (Symbolic heap)**.** Spatial formulas Σ*, and* pure formulas π *are defined by the following grammar:*

$$\pi ::= E = E \mid E \neq E$$
$$\Sigma ::= \bot \mid \top \mid \mathsf{emp} \mid x \overset{\boldsymbol{f}}{\mapsto} \boldsymbol{E} \mid P_\alpha(\boldsymbol{E}) \mid \Sigma * \Sigma$$

where the sequence of fields $\boldsymbol{f}$ in $x \overset{\boldsymbol{f}}{\mapsto} \boldsymbol{E}$ must satisfy $|\boldsymbol{f}| = |\boldsymbol{E}|$, and in $P_\alpha(\boldsymbol{E})$ we require $|\boldsymbol{E}| = \mathsf{ar}(P)$.

A symbolic heap *F is given by $\exists \boldsymbol{x}.\ \Pi : \Sigma$, where Π is a set of pure formulas and $\boldsymbol{x}$ is a sequence of (distinct) variables. When Π or $\boldsymbol{x}$ is empty then we omit them. We write $\mathsf{fv}(F)$ for the set of free variables in a symbolic heap F.*

Symbolic heaps denote concrete memory states, via a satisfaction relation $\models_X$, where $X : \mathsf{Pred} \to \wp(\mathsf{Heaps} \times \bigcup_{n \geq 0}(\mathsf{Val}^n))$ is a function interpreting predicate symbols as sets of memory states (where Heaps is the set of all heaps and $\wp$ is powerset).

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

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

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

$$\frac{l \notin \mathit{dom}(h) \quad \boldsymbol{v} \subset \mathsf{Val} \quad |\boldsymbol{v}| = |\boldsymbol{f}|}{((x := \texttt{new}(\boldsymbol{f}); C, s) \cdot \Xi, h) \rightsquigarrow ((C, s[x \mapsto l]) \cdot \Xi, h[l \mapsto (\boldsymbol{f} \mapsto \boldsymbol{v})])}$$

$$\frac{[\![x]\!]s \in \mathit{dom}(h)}{((\texttt{free}(x); C, s) \cdot \Xi, h) \rightsquigarrow ((C, s) \cdot \Xi, h \restriction \mathit{dom}(h) \setminus \{[\![x]\!]s\})}$$

$$\frac{s \in [\![B]\!]}{((\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \texttt{fi}; C, s) \cdot \Xi, h) \rightsquigarrow ((C_1; C, s) \cdot \Xi, h)}$$

$$\frac{s \in [\![\overline{B}]\!]}{((\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \texttt{fi}; C, s) \cdot \Xi, h) \rightsquigarrow ((C_2; C, s) \cdot \Xi, h)}$$

$$\frac{s \in [\![B]\!]}{((\texttt{while}\ B\ \texttt{do}\ C\ \texttt{od}; C', s) \cdot \Xi, h) \rightsquigarrow ((C; \texttt{while}\ B\ \texttt{do}\ C\ \texttt{od}; C', s) \cdot \Xi, h)}$$

$$\frac{s \in [\![\overline{B}]\!]}{((\texttt{while}\ B\ \texttt{do}\ C\ \texttt{od}; C', s) \cdot \Xi, h) \rightsquigarrow ((C', s) \cdot \Xi, h)}$$

$$\frac{\mathit{body}(p) = C' \quad \mathit{params}(p) = \boldsymbol{x} \quad |\boldsymbol{E}| = |\boldsymbol{x}| \quad s' \restriction \boldsymbol{x} = (\boldsymbol{x} \mapsto [\![\boldsymbol{E}]\!]s)}{((p(\boldsymbol{E}); C, s) \cdot \Xi, h) \rightsquigarrow ((C', s') \cdot (C, s) \cdot \Xi, h)}$$

$$\frac{\Xi \neq \epsilon}{((\epsilon, s) \cdot \Xi, h) \rightsquigarrow (\Xi, h)}$$

$$\frac{[\![x]\!]s \notin \mathit{dom}(h)}{((y := x.f; C, s) \cdot \Xi, h) \rightsquigarrow \mathit{fault}} \qquad \frac{[\![x]\!]s \in \mathit{dom}(h) \quad f \notin \mathit{dom}(h([\![x]\!]s))}{((y := x.f; C, s) \cdot \Xi, h) \rightsquigarrow \mathit{fault}}$$

$$\frac{[\![x]\!]s \notin \mathit{dom}(h)}{((x.f := E; C, s) \cdot \Xi, h) \rightsquigarrow \mathit{fault}} \qquad \frac{[\![x]\!]s \in \mathit{dom}(h) \quad f \notin \mathit{dom}(h([\![x]\!]s))}{((x.f := E; C, s) \cdot \Xi, h) \rightsquigarrow \mathit{fault}}$$

$$\frac{[\![x]\!]s \notin \mathit{dom}(h)}{((\texttt{free}(x); C, s) \cdot \Xi, h) \rightsquigarrow \mathit{fault}} \qquad \frac{p \text{ undefined}}{((p(\boldsymbol{E}); C, s) \cdot \Xi, h) \rightsquigarrow \mathit{fault}}$$

$$\frac{\mathit{params}(p) = \boldsymbol{x} \quad |\boldsymbol{E}| \neq |\boldsymbol{x}|}{((p(\boldsymbol{E}); C, s) \cdot \Xi, h) \rightsquigarrow \mathit{fault}}$$

Figure 3: Operational semantics of `while` programs with pointers and procedures.

Definition 2 (Satisfaction)**.** *The* satisfaction relation $\models$ *between pairs of stores and heaps, predicate interpretations and formulas is defined inductively over the structure of formulas as follows:*

$$\begin{aligned}
(s,h) &\models_X E_1 = E_2 && \Leftrightarrow [\![E_1]\!]s = [\![E_2]\!]s \\
(s,h) &\models_X E_1 \neq E_2 && \Leftrightarrow [\![E_1]\!]s \neq [\![E_2]\!]s \\
(s,h) &\models_X \bot && \Leftrightarrow \mathsf{false} \\
(s,h) &\models_X \top && \Leftrightarrow \mathsf{true} \\
(s,h) &\models_X \mathsf{emp} && \Leftrightarrow h = e \\
(s,h) &\models_X x \overset{\boldsymbol{f}}{\mapsto} \boldsymbol{E} && \Leftrightarrow \mathit{dom}(h) = \{[\![x]\!]s\} \\
& && \qquad \mathit{and}\ h([\![x]\!]s) = (\boldsymbol{f} \mapsto [\![\boldsymbol{E}]\!]s) \\
(s,h) &\models_X P_\alpha(\boldsymbol{E}) && \Leftrightarrow (h, [\![\boldsymbol{E}]\!]s) \in X(P) \\
(s,h) &\models_X \Sigma_1 * \Sigma_2 && \Leftrightarrow h = h_1 \circ h_2\ \mathit{and}\ (s,h_1) \models_X \Sigma_1 \\
& && \qquad \mathit{and}\ (s,h_2) \models_X \Sigma_2
\end{aligned}$$

$$(s,h) \models_X \Pi : \Sigma \quad \Leftrightarrow (s,h) \models_X \pi \text{ for all } \pi \in \Pi \text{ and } (s,h) \models_X \Sigma$$
$$(s,h) \models_X \exists x.\, F \quad \Leftrightarrow \exists v \in \mathsf{Val}.\, (s[x \mapsto v], h) \models_X F$$

Notice that this notion of satisfaction ignores predicate labels. These will be interpreted once we bootstrap this relation to construct predicate interpretations from inductive definitions.

Definition 3 (Inductive rule set)**.** *An* inductive rule set *is a finite set of* inductive rules*, each of the form* $\Pi : \Sigma \Rightarrow P(\boldsymbol{x})$ *where* $\Pi : \Sigma$ *is a symbolic heap formula (in which all predicate instances must have distinct labels), and* $P(\boldsymbol{x})$ *is a predicate formula. We sometimes write* $F \overset{\boldsymbol{z}}{\Rightarrow} P(\boldsymbol{x})$ *to indicate that* $\boldsymbol{z}$ *is the set of variables occurring in the body* F *of the inductive rule which are* not *formal parameters* $\boldsymbol{x}$*; the variables in* $\boldsymbol{z}$ *are implicitly existentially quantified.*

If Φ *denotes an inductive rule set, then we write* Φ_P *for the set of all inductive rules of the form* $F \Rightarrow P(\boldsymbol{x})$ *in* Φ*.*

The List predicate used in Section 2, for example, can be defined by the inductive rule set $\Phi_{\mathsf{List}} = \{x = \mathsf{nil} : \mathsf{emp} \Rightarrow \mathsf{List}(x), x \neq \mathsf{nil} : x \overset{\mathtt{nxt}}{\longmapsto} y * \mathsf{List}_\alpha(y) \Rightarrow \mathsf{List}(x)\}$. Binary trees whose nodes are represented by memory cells with fields named `l` and `r` and leaves by null pointers can be defined by the inductive rule set $\Phi_{\mathsf{bt}} = \{x = \mathsf{nil} : \mathsf{emp} \Rightarrow \mathsf{bt}(x), x \neq \mathsf{nil} : x \overset{\mathtt{l,r}}{\longmapsto} l, r * \mathsf{bt}_\alpha(l) * \mathsf{bt}_\beta(r) \Rightarrow \mathsf{bt}(x)\}$.

Each inductive rule set Φ induces a characteristic operator φ_Φ on predicate interpretations by $\varphi_\Phi(X)(P) =_{\text{def}} \{(h, [\![\boldsymbol{x}]\!]s) \mid \exists F.\, (s,h) \models_X F \;\&\; F \Rightarrow P(\boldsymbol{x}) \in \Phi\}$. Given a predicate interpretation X, φ_Φ will generate a new interpretation X' containing, for each predicate P, models obtained by applying the inductive rules in Φ to the models in X. For example, the models in $\varphi_{\Phi_{\mathsf{List}}}(X)(\mathsf{List})$ are obtained by prepending a new head (i.e. memory cell) to each model in the set $X(\mathsf{List})$.

We define a partial ordering $\leq$ on the set of predicate interpretations $\mathcal{I}$ by $X \leq X' \Leftrightarrow \forall P.\, X(P) \subseteq X'(P)$. One can note that $(\mathcal{I}, \leq)$ is a complete lattice and the least element, denoted by $X_\bot$, maps all predicate names to the empty set. Moreover, characteristic operators are *monotone* with respect to $\leq$, thus admitting the following (standard) construction that builds interpretations via a process of *approximation*.

Definition 4 (Interpretation of inductive rule sets)**.** *An inductive rule set* Φ *is interpreted as the least prefixed point of its characteristic predicate interpretation operator,* $\mu X.\, \varphi_\Phi(X)$*. This least prefixed point, denoted by* $[\![\Phi]\!]$*, can be approached iteratively being the supremum of the (ordinal-indexed) chain*

$$X_\bot \leq \varphi_\Phi(X_\bot) \leq \varphi_\Phi(\varphi_\Phi(X_\bot)) \leq \ldots \leq \varphi^\alpha_\Phi(X_\bot) \leq \ldots$$

For each α*,* $\varphi^\alpha_\Phi(X_\bot)$ *is called an* approximation *of* $[\![\Phi]\!]$ *and may be denoted by* $[\![\Phi]\!]_\alpha$*.*

Our full assertion language makes use of explicit constraints over predicate labels.

Definition 5 (Constrained formula)**.** Constrained formulas *are given by* $\exists \boldsymbol{\alpha}.\, \Omega : \Delta$*, where* $\boldsymbol{\alpha}$ *is a sequence of predicate labels,* Ω *is a set of* constraints *of the form* $\alpha < \beta$ *or* $\alpha \leq \beta$*, where* α *and* β *are predicate labels, and* Δ *is a disjunction of symbolic heaps (cf. Definition 1). When the quantifier sequence or the constraint set is empty, we omit them.*

For a constrained formula $\phi = \exists \boldsymbol{\alpha}.\, \Omega : \Delta$, we write $\mathsf{constraints}(\phi)$ to denote the constraint set Ω, we write $\mathsf{free\text{-}labs}(\phi)$ for the set of labels occurring free in ϕ, and $\mathsf{lab}(\phi)$ for the set of all labels in ϕ.

Constrained formulas are interpreted as triples consisting of predicate label valuations (i.e. maps from predicate labels to ordinals), variable stores and heaps.

Definition 6 (Extended satisfaction)**.** *Let* Φ *be an inductive rule set. Then the satisfaction relation between interpretations and constrained formulas is defined by extending the satisfaction relation in Definition 2*

$$(\rho, s, h) \models_\Phi P_\alpha(\boldsymbol{E}) \quad \Leftrightarrow (h, [\![\boldsymbol{E}]\!]s) \in [\![\Phi]\!]_{\rho(\alpha)}(P)$$
$$(\rho, s, h) \models_X \Delta_1 \vee \Delta_2 \;\Leftrightarrow (\rho, s, h) \models_X \Delta_1 \text{ or } (\rho, s, h) \models_X \Delta_2$$
$$(\rho, s, h) \models_\Phi \exists \boldsymbol{\alpha}.\, \Omega : \Delta \Leftrightarrow \exists \rho'.\, \rho' =_{\setminus \boldsymbol{\alpha}} \rho \text{ and } (\rho', s, h) \models_\Phi \Delta \text{ and } \rho'(\alpha) \sim \rho'(\beta) \text{ for all } \alpha \sim \beta \in \Omega \; (\sim \in \{<, \leq\})$$

When $(\rho, s, h) \models_\Phi \phi$ *holds, we say that* (ρ, s, h) *is a* model *of the constrained formula* ϕ *(wrt. the inductive rule set* Φ*).*

Thus, predicate instances $P_\alpha(\boldsymbol{E})$ are interpreted using the $\rho(\alpha)^{\text{th}}$ approximation of the interpretation of the inductive rule set Φ, and the possible valuations for a model are restricted by the set of predicate label constraints.

Definition 7 (Entailment)**.** *We write* $\phi \vDash_\Phi \psi$*, where* ϕ *and* ψ *are constrained formulas, to mean that* $(\rho, s, h) \models_\Phi \phi$ *implies* $(\rho, s, h) \models_\Phi \psi$ *for all* (ρ, s, h)*.*

We additionally write: $\rho \models \Omega$ *to mean that* $\rho(\alpha) \sim \rho(\beta)$ $(\sim \in \{<, \leq\})$ *for all constraints* $\alpha \sim \beta \in \Omega$*; and* $\Omega \vDash \Omega'$ *to mean that* $\rho \models \Omega$ *implies* $\rho \models \Omega'$ *for all* ρ*.*

4. Cyclic Termination Proofs

In this section, we describe how we verify termination of programs written in the language defined in Section 3, using a Hoare logic built on the assertion language defined above. Our proof system manipulates *Hoare triples* $\{\phi\}\, C\, \{\psi\}$, where C is a program in our language and the pre- and postconditions ϕ and ψ, respectively, are constrained formulas of our assertion language.

Definition 8 (Validity)**.** *We say a Hoare triple* $\{\phi\}\, C\, \{\psi\}$ *is* valid *iff, whenever* $(\rho, s, h) \models_\Phi \phi$*, it is also the case that* $((C, s), h)$ *is terminating and* $(\rho, s', h') \models_\Phi \psi$ *for all final states* (s', h')*.*

$$\text{(stop):}\quad \frac{}{\vdash_\Phi \{\phi\}\, \epsilon\, \{\psi\}}\ (\phi \vDash_\Phi \psi) \qquad \text{(write):}\quad \frac{\vdash_\Phi \{\Omega : \Pi : x \overset{\boldsymbol{f}}{\longmapsto} \boldsymbol{E'} * \Sigma\}\, C\, \{\psi\}}{\vdash_\Phi \{\Omega : \Pi : x \overset{\boldsymbol{f}}{\longmapsto} \boldsymbol{E} * \Sigma\}\, x.\boldsymbol{f}_i := E; C\, \{\psi\}}\ (\boldsymbol{E'} \text{ is } \boldsymbol{E} \text{ with } E \text{ substituted for } \boldsymbol{E}_i)$$

$$\text{(assign)}^*\text{:}\quad \frac{\vdash_\Phi \{\Omega : \Pi[x'/x] \cup \{x = E[x'/x]\} : \Sigma[x'/x]\}\, C\, \{\psi\}}{\vdash_\Phi \{\Omega : \Pi : \Sigma\}\, x := E; C\, \{\psi\}} \qquad \text{(read)}^*\text{:}\quad \frac{\vdash_\Phi \{\Omega : \Pi[x'/x] \cup \{x = \boldsymbol{E}_i[x'/x]\} : (y \overset{\boldsymbol{f}}{\longmapsto} \boldsymbol{E} * \Sigma)[x'/x]\}\, C\, \{\psi\}}{\vdash_\Phi \{\Omega : \Pi : y \overset{\boldsymbol{f}}{\longmapsto} \boldsymbol{E} * \Sigma\}\, x := y.\boldsymbol{f}_i; C\, \{\psi\}}$$

$$\text{(free):}\quad \frac{\vdash_\Phi \{\Omega : \Pi : \Sigma\}\, C\, \{\psi\}}{\vdash_\Phi \{\Omega : \Pi : x \overset{\boldsymbol{f}}{\longmapsto} \boldsymbol{E} * \Sigma\}\, \texttt{free}(x); C\, \{\psi\}} \qquad \text{(new):}\quad \frac{\vdash_\Phi \{\Omega : \Pi[x'/x] : x \overset{\boldsymbol{f}}{\longmapsto} \boldsymbol{x} * \Sigma[x'/x]\}\, C\, \{\psi\}}{\vdash_\Phi \{\Omega : \Pi : \Sigma\}\, x := \texttt{new}(\boldsymbol{f}); C\, \{\psi\}}\ \begin{pmatrix} |\boldsymbol{x}| = |\boldsymbol{f}| \\ x' \text{ and } \boldsymbol{x} \text{ fresh, } \boldsymbol{x} \text{ pairwise distinct} \end{pmatrix}$$

$$\text{(if-nondet):}\quad \frac{\vdash_\Phi \{\phi\}\, C_1; C\, \{\psi\} \quad \vdash_\Phi \{\phi\}\, C_2; C\, \{\psi\}}{\vdash_\Phi \{\phi\}\, \texttt{if} \star \texttt{then}\, C_1\, \texttt{else}\, C_2\, \texttt{fi}; C\, \{\psi\}} \qquad \text{(if-det):}\quad \frac{\vdash_\Phi \{\Omega : \Pi \cup \{\pi\} : \Sigma\}\, C_1; C\, \{\psi\} \quad \vdash_\Phi \{\Omega : \Pi \cup \{\overline{\pi}\} : \Sigma\}\, C_2; C\, \{\psi\}}{\vdash_\Phi \{\Omega : \Pi : \Sigma\}\, \texttt{if}\, \pi\, \texttt{then}\, C_1\, \texttt{else}\, C_2\, \texttt{fi}; C\, \{\psi\}}$$

$$\text{(while-nondet):}\quad \frac{\vdash_\Phi \{\phi\}\, C'; \texttt{while} \star \texttt{do}\, C'\, \texttt{od}; C\, \{\psi\} \quad \vdash_\Phi \{\phi\}\, C\, \{\psi\}}{\vdash_\Phi \{\phi\}\, \texttt{while} \star \texttt{do}\, C'\, \texttt{od}; C\, \{\psi\}} \qquad \text{(proc):}\quad \frac{\vdash_\Phi \{\phi\}\, \mathit{body}(p)\, \{\psi\}}{\vdash_\Phi \{\phi\}\, p(\boldsymbol{x})\, \{\psi\}}\ \begin{pmatrix} \boldsymbol{x} = \mathit{params}(p) \\ \mathit{locals}(p) \cap (\mathsf{fv}(\phi) \cup \mathsf{fv}(\psi)) = \emptyset \end{pmatrix}$$

$$\text{(while-det):}\quad \frac{\vdash_\Phi \{\Omega : \Pi \cup \{\pi\} : \Sigma\}\, C'; \texttt{while}\, \pi\, \texttt{do}\, C'\, \texttt{od}; C\, \{\psi\} \quad \vdash_\Phi \{\Omega : \Pi \cup \{\overline{\pi}\} : \Sigma\}\, C\, \{\psi\}}{\vdash_\Phi \{\Omega : \Pi : \Sigma\}\, \texttt{while}\, \pi\, \texttt{do}\, C'\, \texttt{od}; C\, \{\psi\}}$$

$(*)$ where x' is a fresh variable

Figure 4: Symbolic execution proof rules

$$(\bot)\text{:}\quad \frac{}{\vdash_\Phi \{\bot\}\, C\, \{\psi\}} \qquad \text{(consequence / } \models\text{):}\quad \frac{\vdash_\Phi \{\chi\}\, C\, \{\xi\}}{\vdash_\Phi \{\phi\}\, C\, \{\psi\}}\ (\phi \vDash_\Phi \chi,\ \xi \vDash_\Phi \psi) \qquad \text{(split):}\quad \frac{\vdash_\Phi \{\Omega : \Delta_1\}\, C\, \{\psi\} \quad \vdash_\Phi \{\Omega : \Delta_2\}\, C\, \{\psi\}}{\vdash_\Phi \{\Omega : \Delta_1 \vee \Delta_2\}\, C\, \{\psi\}}$$

$$\text{(frame):}\quad \frac{\vdash_\Phi \{\phi\}\, C\, \{\psi\}}{\vdash_\Phi \{\phi * \chi\}\, C\, \{\psi * \chi\}}\ (\mathsf{fv}(\chi) \cap \mathit{mod}(C) = \emptyset) \qquad \text{(subst):}\quad \frac{\vdash_\Phi \{\phi\}\, C\, \{\psi\}}{\vdash_\Phi \{\phi[E/x]\}\, C\, \{\psi[E/x]\}}\ \begin{pmatrix} x \notin \mathit{vars}(C) \\ x \in \mathsf{fv}(\psi) \Rightarrow E \notin \mathit{vars}(C) \end{pmatrix}$$

$$\text{(relabel):}\quad \frac{\vdash_\Phi \{\phi\}\, C\, \{\psi\}}{\vdash_\Phi \{\phi[\beta/\alpha]\}\, C\, \{\psi[\beta/\alpha]\}} \qquad \text{(param):}\quad \frac{\vdash_\Phi \{\phi\}\, p(\boldsymbol{E})\, \{\psi\}}{\vdash_\Phi \{\phi[E/x]\}\, p(\boldsymbol{E}[E/x])\, \{\psi[E/x]\}} \qquad \text{(seq):}\quad \frac{\vdash_\Phi \{\phi\}\, C_1\, \{\chi\} \quad \vdash_\Phi \{\chi\}\, C_2\, \{\psi\}}{\vdash_\Phi \{\phi\}\, C_1; C_2\, \{\psi\}}$$

$$(\exists\text{Var}):\quad \frac{\vdash_\Phi \{\Omega : F[y/x]\}\, C\, \{\psi\}}{\vdash_\Phi \{\Omega : \exists x.\ F\}\, C\, \{\psi\}}\ (y \notin \mathsf{fv}(\exists x.\ F) \cup \mathsf{fv}(\psi)) \qquad (\exists\text{Lab}):\quad \frac{\vdash_\Phi \{\exists \boldsymbol{\alpha}.\ \Omega[\gamma/\beta] : \Delta[\gamma/\beta]\}\, C\, \{\psi\}}{\vdash_\Phi \{\exists \boldsymbol{\alpha} \cup \{\beta\}.\ \Omega : \Delta\}\, C\, \{\psi\}}\ (\gamma \notin \boldsymbol{\alpha} \cup \mathsf{lab}(\Omega : \Delta) \cup \mathsf{free\text{-}labs}(\psi))$$

$$\text{(unfold):}\quad \frac{\vdash_\Phi \{\Omega \cup \Omega_i : \Pi_i' \cup \Pi : \Sigma_i' * \Sigma\}\, C\, \{\psi\} \quad (\forall\, 1 \le i \le n)}{\vdash_\Phi \{\Omega : \Pi : P_\alpha(\boldsymbol{E}) * \Sigma\}\, C\, \{\psi\}}\ \begin{pmatrix} \Phi_P = \{\Pi_1 : \Sigma_1 \overset{\boldsymbol{z_1}}{\Rightarrow} P(\boldsymbol{x_1}), \ldots, \Pi_n : \Sigma_n \overset{\boldsymbol{z_n}}{\Rightarrow} P(\boldsymbol{x_n})\} \text{ and for all } 1 \le i \le n\text{:} \\ \quad \Pi_i' : \Sigma_i' \text{ is } \Pi_i : \Sigma_i \text{ with variables } \boldsymbol{z_i} \text{ and labels freshened, and arguments } \boldsymbol{E} \\ \qquad \text{substituted for parameters } \boldsymbol{x_i} \\ \quad \Omega_i = \{\beta < \alpha \mid \beta \in \mathsf{lab}(\Sigma_i')\} \end{pmatrix}$$

Figure 5: Logical proof rules

In contrast to the standard notion of proof in Hoare-style logic, our framework employs *cyclic proofs*, which are finite derivation trees in which leaves may be closed by *back-links* to identical interior nodes. Thus, as usual in cyclic proof systems, an additional *global* soundness condition must be imposed on the derivation graph, which amounts to ensuring that all infinite paths in the proof correspond to valid arguments by *infinite descent* (cf. [11, 14]).

The rules of our proof system are given in Figs. 4 and 5. The *symbolic execution* rules, viewed from conclusion to premise, capture the effect on the precondition of executing commands. The logical rules are mainly standard; but we draw particular attention to the well-known *frame* rule of separation logic [35] which permits portions of the symbolic heap that are not affected by the program to be disregarded. Note that we lift the separating conjunction $(*)$ to constrained formulas in the obvious way. The unfold rule performs a case-split on a predicate instance in the precondition, by replacing it with the body of each clause of its inductive definition in turn (generating extra label constraints as appropriate).

Cyclic *pre-proofs* are finite derivation trees with *back-links*: to every leaf that is not the conclusion of an axiom we assign a syntactically identical interior node. By identifying back-linked nodes, a pre-proof may thus be seen as a representation of an infinite, regular derivation tree by a cyclic graph. If $\mathcal{P}$ is a pre-proof, then we write $\mathcal{G}_\mathcal{P}$ to denote this graph. A *path* $\boldsymbol{\nu}$ in $\mathcal{G}_\mathcal{P}$ is a (possibly infinite) sequence of nodes in $\mathcal{G}_\mathcal{P}$ such that for each element $\boldsymbol{\nu}_i$ $(i > 1)$ there is

an edge in $\mathcal{G}_\mathcal{P}$ from $\boldsymbol{\nu}_{i-1}$ to $\boldsymbol{\nu}_i$. In an abuse of notation, we may write $\boldsymbol{\nu} = (S_1, r_1), (S_2, r_2), \ldots$ where S_i is the sequent associated with node $\boldsymbol{\nu}_i$ and r_i is the rule for which S_i is the conclusion and S_{i+1} is a premise.

Our global soundness condition, required to qualify such pre-proofs as genuine *cyclic proofs*, is formulated in terms of the following concept of a *trace* through the pre-proof (cf. [11, 12, 31]).

Definition 9 (Traces)**.** *1. Let sequents $S_1 = \vdash_\Phi \{\phi\}\, C\, \{\psi\}$ and $S_2 = \vdash_\Phi \{\phi'\}\, C'\, \{\psi'\}$ be, respectively, the conclusion and a premise of some instance of a proof rule r; a tuple (α, β) of labels is a* trace pair *for (S_1, S_2, r) if and only if the following conditions hold:*

- *if $r \neq (\text{relabel})$ then $\mathsf{constraints}(\phi') \vDash \{\beta \leq \alpha\}$;*
- *if $r = (\text{relabel})$ and the relabelling applied in the rule instance is $[\gamma/\delta]$, then either $\mathsf{constraints}(\phi') \vDash \{\beta \leq \alpha\}$ with $\alpha \neq \delta$ if δ not bound in ϕ', or $\mathsf{constraints}(\phi') \vDash \{\beta \leq \delta\}$ with $\alpha = \gamma$ and δ not bound in ϕ';*
- *if $r = (\text{consequence})$ or $r = (\text{seq})$ with S_2 the right-hand premise of the rule instance, then α is free in both ϕ and ϕ'.*

If these conditions hold with $<$ in place of $\leq$, then we say that (α, β) is a progressing *trace pair for (S_1, S_2, r).*

2. Let $\boldsymbol{\nu} = (S_1, r_1), (S_2, r_2), \ldots$ be a (possibly infinite) path; a trace $\boldsymbol{\tau}$ following $\boldsymbol{\nu}$ *is a sequence of predicate labels such that $|\boldsymbol{\tau}| = |\boldsymbol{\nu}|$ and $(\boldsymbol{\tau}_i, \boldsymbol{\tau}_{i+1})$ is a trace pair for (S_i, S_{i+1}, r_i) for each $1 \leq i < |\boldsymbol{\tau}|$. Whenever $(\boldsymbol{\tau}_i, \boldsymbol{\tau}_{i+1})$ is a progressing trace pair, then we say that $\boldsymbol{\tau}$* progresses *at i. We say that $\boldsymbol{\tau}$ is* infinitely progressing *if it progresses at infinitely many points.*

Cyclic pre-proofs may only be considered valid proofs when they satisfy a global soundness condition.

Definition 10 (Cyclic proof)**.** *A cyclic pre-proof $\mathcal{P}$ is a* cyclic proof *if for every infinite path $\boldsymbol{\nu}$ in $\mathcal{G}_\mathcal{P}$ there is an infinitely progressing trace $\boldsymbol{\tau}$ following some tail of $\boldsymbol{\nu}$.*

This global soundness condition can be decided (modulo decidability of entailment between predicate label constraints, which is here trivial) by transforming the pre-proof graph into a Büchi automaton with the accepting states given by those edges that admit a progressing trace pair (cf. [14]).

The following soundness result follows similarly to [12].

Theorem 11 (Soundness)**.** *If $\vdash_\Phi \{\phi\}\, C\, \{\psi\}$ has a cyclic proof, then $\{\phi\}\, C\, \{\psi\}$ is valid.*

Proof. (Sketch) Each of the inference rules r satisfies the following property:

(Local Soundness) Suppose $S \equiv \{\phi\}\, C\, \{\psi\}$ is the conclusion of r and that it is *not* valid; i.e., there is some model $(\rho, s, h) \models \phi$, but either $\kappa = ((C, s), h)$ is not terminating or (s', h') is a final state for κ and $(\rho, s', h') \not\models \psi$. Then there is a premise $S' \equiv \{\phi'\}\, C'\, \{\psi'\}$ of r and a model $(\rho', s'', h'') \models \phi'$ such that $\kappa' = ((C', s''), h'')$ is not terminating or (s''', h''') is a final state for κ' and $(\rho', s''', h''') \not\models \psi'$ — that is S' is not valid either. Moreover, if (α, β) is a trace pair for (S, S', r) then $\rho'(\beta) \leq \rho(\alpha)$, and $\rho'(\beta) < \rho(\alpha)$ if the trace pair is progressing.

Now $\vdash_\Phi \{\phi\}\, C\, \{\psi\}$ has a cyclic proof $\mathcal{P}$, so suppose for contradiction that $\{\phi\}\, C\, \{\psi\}$ is not valid. Then it follows from the local soundness property above that there is an infinite path $\boldsymbol{\nu}$ of (invalid) sequents in $\mathcal{G}_\mathcal{P}$. Since $\mathcal{P}$ is a valid cyclic proof, there is also an infinitely progressing trace $\boldsymbol{\tau}$ following $\boldsymbol{\nu}$. It then also follows from the local soundness property that there is an infinite sequence of valuations $\boldsymbol{\rho}$ such that for each pair $(\boldsymbol{\tau}_i, \boldsymbol{\tau}_{i+1})$ of the trace it is the case that $\boldsymbol{\rho}_{i+1}(\boldsymbol{\nu}_{i+1}) \leq \boldsymbol{\rho}_i(\boldsymbol{\nu}_i)$, with $\boldsymbol{\rho}_{i+1}(\boldsymbol{\nu}_{i+1}) < \boldsymbol{\rho}_i(\boldsymbol{\nu}_i)$ if the trace pair is progressing. Therefore, since the trace is infinitely progressing, this comprises an infinitely decreasing chain of ordinals. This contradicts the well-foundedness of the ordinals, thus we conclude $\{\phi\}\, C\, \{\psi\}$ is indeed valid. □

Remark 1. One might also consider the question of *relative completeness*: if one assumes a complete proof system for entailments between assertions, does every valid triple $\{\phi\}\, C\, \{\psi\}$ then have a cyclic termination proof? Such a result was shown to hold of the cyclic system for simple `while` programs without procedures in [12]. The result in [12] crucially depends on the presence of the *separating implication*, $-\!\!*$, to express weakest termination preconditions. It seems that, at a minimum, we would need to add this connective to our system to have any hope of relative completeness; we would also need to express the weakest termination precondition relative to a given postcondition (cf. [32]). Here, however, we deliberately exclude $-\!\!*$ from our assertion language, since our main focus here is on automation, for which $-\!\!*$ is well known to cause difficulties (and it is not typically handled by most separation logic provers).

5. Implementation

We have implemented a tool that can automatically verify the termination of programs written in the language described in Section 3, based upon the cyclic proof system in Section 4: given as input a set of inductive predicate definitions and a Hoare triple in our language, it searches for a cyclic termination proof. The tool is built on top of CYCLIST [1], a general framework for implementing cyclic theorem provers which provides a general search procedure for cyclic proofs (see [14] for details). Logic-specific theorem provers can be obtained by instantiating this general procedure with the appropriate syntax, proof rules and tactics implementing a particular proof system. CYCLIST is able to produce a cyclic proof object as output. Our tool executable, source code and benchmarks are available online [2].

Basic Proof Search Strategy. The tactics that we implement for the proof system of Section 4 are largely guided by the syntax of the program: we attempt to apply as many symbolic

execution rules as possible; when no symbolic execution rules can be applied, we perform predicate unfoldings until sufficient structure is revealed in the current symbolic state to allow further symbolic execution. When all commands have been symbolically executed, we attempt to apply the axiom (`stop`), which involves deciding whether the postcondition is entailed by the current symbolic state. To decide such entailment questions our tool uses a separate instantiation of CYCLIST with an entailment proof system for our logic (see below). Thus it is capable of proving entailments that require inductive reasoning.

Loops and Recursion. These features are handled naturally in cyclic proof by back-linking. When a procedure call or loop is first encountered during symbolic execution, it is *unfolded* using the (`proc`) or (`while`) rules. On subsequent symbolic iterations of the loop or recursions/calls of the procedure, our tactics will typically attempt to form a back-link to a node created by a previous encounter with the loop/procedure. Global soundness of the generated proof is checked incrementally as new back-links are formed.

Frame Inference. To symbolically execute a procedure call, we must verify that the state at the call-site ϕ entails the procedure precondition ψ. However in general this state will contain a portion of memory that is not required by the procedure and which must be considered separately. Thus we must find a solution X that validates the entailment $\phi \vDash \psi * X$; this known as the *frame inference* problem [5]. To allow proofs verifying procedures to be reused at multiple call-sites (i.e. to treat procedure calls *compositionally*), our implementation currently relies on procedures being annotated with pre- and postconditions. An interesting avenue for future work is to extend our implementation with specification inference techniques such as *biabduction* [16].

We perform frame inference by unfolding predicates in the procedure precondition, up to some pre-specified limit, until syntactic matching of atomic formulas is possible. A candidate frame can then be computed by subtracting the matching atomic formulas from the current symbolic state. For example, consider the case where a proof search must process the open subgoal

$$\{\mathsf{List}(z) * x \mapsto y * \mathsf{List}(y)\}\ \texttt{TraverseList(x)}\,;\, C\,\{\psi\}$$

for some program fragment C, and where the precondition of the `TraverseList` procedure is $\mathsf{List}(x)$, as inSection 2. We can proceed be unfolding the List predicate in the procedure precondition to obtain $\exists v.\ x \mapsto v * \mathsf{List}(v)$. At this point, we are able to match (i.e. instantiate) the existentially quantified variable v with y, and subtract the result from the subgoal precondition to infer the frame $\mathsf{List}(z)$. To then symbolically execute the procedure call a backlink may be formed with a previous generating unfolding of the `TraverseList`, and the frame is combined with the procedure postcondition to generate a new subgoal $\{\mathsf{List}(x) * \mathsf{List}(z)\}\ C\ \{\psi\}$ from which proof search can continue. For this simple illustration we have omitted details of generating and matching predicate labels and constraints, which our implementation also handles during frame inference. In general, this unfold-and-match will also generate substitutions of (logical) variables, predicate labels and procedure parameters.

Note that this simple frame inference procedure may fail, not only in the case that no frame exists but also if we do not unfold sufficiently many times or if inductive reasoning is required. Since our tool encounters many potential back-link candidates during proof search, this simple unfold-and-match approach provides a relatively cheap if somewhat weak solution to frame inference. Our experimental results show that, combined with a more powerful procedure for deciding entailments at axioms as well as at other locations specified by the user, this is an effective trade-off.

Entailment Proof System. In Fig. 6 we give the rules for a cyclic proof system deriving entailments between constrained formulas. In the (Equiv) rule we refer to an equivalence $\equiv$ on constrained formulas which is given by the least congruence containing alpha-equivalence on existentially quantified variables and labels, and satisfying the following equations:

($*$-commutativity): $\Sigma_1 * \Sigma_2 \equiv \Sigma_2 * \Sigma_1$

($*$-associativity): $\Sigma_1 * (\Sigma_2 * \Sigma_3) \equiv (\Sigma_1 * \Sigma_2) * \Sigma_3$

(emp-unit): $\mathsf{emp} * \Sigma \equiv \Sigma$

($\top$-absorb): $\top * \Sigma \equiv \top$

($\bot$-absorb): $\bot * \Sigma \equiv \bot$

($\vee$-commutativity): $\Delta_1 \vee \Delta_2 \equiv \Delta_2 \vee \Delta_1$

($\vee$-associativity): $\Delta_1 \vee (\Delta_2 \vee \Delta_3) \equiv (\Delta_1 \vee \Delta_2) \vee \Delta_3$

For the (Id) rule, we refer to the following restricted entailment relation on sets of predicate label constraints:

$$\Omega_1 \vDash_{\backslash \boldsymbol{\alpha}} \Omega_2 \Leftrightarrow \forall \rho. \rho \models \Omega_1 \Rightarrow\ \exists \rho'.\ \rho =_{\backslash \boldsymbol{\alpha}} \rho' \wedge \rho' \models \Omega_2$$

Traces for this cyclic entailment system are defined similarly to traces for the program termination proof system (see Definition 9), using predicate labels: for most rules (α, β) is a trace pair for the conclusion-premise pair $(\phi \vdash_\Phi \psi, \chi \vdash_\Phi \xi)$ if $\mathsf{constraints}(\chi) \vDash \{\beta \leq \alpha\}$. The same conditions apply for the (Relab) rule as for the (relabel) rule in Definition 9. The (cut) rule carries the same condition as the (seq) rule that α be free in both ϕ and ξ. We allow that (β, γ) is a trace pair for the ($\exists$LabL) rule, where the free label γ in the premise is existentially quantifies as γ in the conclusion; this can be seen as a renaming. Lastly, we also allow (α, β) as a trace pair for the (Equiv) rule where β in the premise has been alpha-renamed to α in the conclusion. The entailment proof system can be shown to be sound using a similar argument to the one given in the proof sketch for Theorem 11, including a similar notion of local soundness for the entailment rules.

$$\text{(Id):}\ \frac{}{\Omega_1 : F \vdash_\Phi \exists \boldsymbol{\alpha}.\Omega_2 : F}\ (\Omega_1 \vDash_{\setminus \boldsymbol{\alpha}} \Omega_2) \qquad \text{(}\top\text{):}\ \frac{}{\Sigma \vdash_\Phi \top} \qquad \text{(}\bot\text{L):}\ \frac{}{\bot \vdash_\Phi \Omega : \Pi : \Sigma} \qquad \text{(}\bot\text{R}_1\text{):}\ \frac{}{x \xmapsto{f_1} \boldsymbol{E_1} * x \xmapsto{f_2} \boldsymbol{E_2} \vdash_\Phi \bot}$$

$$\text{(}\bot\text{R}_2\text{):}\ \frac{}{\{E_1 = E_2, E_1 \neq E_2\} : \Sigma \vdash_\Phi \bot} \qquad \text{(}\bot\text{R}_3\text{):}\ \frac{}{\{E \neq E\} : \Sigma \vdash_\Phi \bot} \qquad \text{(}\bot\text{R}_4\text{):}\ \frac{}{\Omega : \Sigma \vdash_\Phi \bot}\ (\Omega \vDash \bot) \qquad \text{(=R):}\ \frac{}{\Sigma \vdash_\Phi E = E : \Sigma}$$

$$\text{(Wk):}\ \frac{\Omega : \Pi : \Sigma \vdash_\Phi \psi}{\Omega \cup \Omega' : \Pi \cup \Pi' : \Sigma \vdash_\Phi \psi} \qquad \text{(}\vee\text{L):}\ \frac{\Omega : \Delta_1 \vdash_\Phi \psi \quad \Omega : \Delta_2 \vdash_\Phi \psi}{\Omega : \Delta_1 \vee \Delta_2 \vdash_\Phi \psi} \qquad \text{(}\vee\text{R):}\ \frac{\phi \vdash_\Phi \Omega : \Delta}{\phi \vdash_\Phi \Omega : \Delta \vee \Delta'} \qquad \text{(Relab):}\ \frac{\phi \vdash_\Phi \psi}{\phi[\beta/\alpha] \vdash_\Phi \psi[\beta/\alpha]}$$

$$\text{(Subst):}\ \frac{\phi \vdash_\Phi \psi}{\phi[E/x] \vdash_\Phi \psi[E/x]} \qquad \text{(Equiv):}\ \frac{\chi \vdash_\Phi \xi}{\phi \vdash_\Phi \psi}\ (\phi \equiv \chi, \psi \equiv \xi) \qquad (*)\text{:}\ \frac{\phi \vdash_\Phi \psi \quad \phi' \vdash_\Phi \psi'}{\phi * \phi' \vdash_\Phi \psi * \psi'} \qquad \text{(Cut):}\ \frac{\phi \vdash_\Phi \chi \quad \chi \vdash_\Phi \psi}{\phi \vdash_\Phi \psi}$$

$$\text{(}\exists\text{VarL):}\ \frac{\Omega : F[y/x] \vdash_\Phi \psi}{\Omega : \exists x.\, F \vdash_\Phi \psi}\ (y \notin \mathsf{fv}(\exists x F) \cup \mathsf{fv}(\psi)) \qquad \text{(}\exists\text{VarR):}\ \frac{\phi \vdash_\Phi \Omega : F[E/x]}{\phi \vdash_\Phi \Omega : \exists x.\, F} \qquad \text{(}\exists\text{LabR):}\ \frac{\phi \vdash_\Phi \exists \boldsymbol{\beta}.\Omega[\gamma/\delta] : \Delta[\gamma/\delta]}{\phi \vdash_\Phi \exists \boldsymbol{\beta} \cup \{\delta\}.\Omega : \Delta}$$

$$\text{(}\exists\text{LabL):}\ \frac{\exists \boldsymbol{\alpha}.\Omega[\gamma/\beta] : \Delta[\gamma/\beta] \vdash_\Phi \psi}{\exists \boldsymbol{\alpha} \cup \{\beta\}.\Omega : \Delta \vdash_\Phi \psi}\ (\gamma \notin \boldsymbol{\alpha} \cup \mathsf{lab}(\Omega : \Delta) \cup \mathsf{free\text{-}labs}(\psi)) \qquad \text{(Exchange):}\ \frac{\Omega : (\Pi \cup \{v = w\} : \Sigma)[v/x, w/y] \vdash_\Phi \psi[v/x, w/y]}{\Omega : (\Pi \cup \{v = w\} : \Sigma)[w/x, v/y] \vdash_\Phi \psi[w/x, v/y]}$$

$$\text{(UnfoldL):}\ \frac{\Omega \cup \Omega_i : \Pi_i' \cup \Pi : \Sigma_i' * \Sigma) \vdash_\Phi \psi \quad (\forall\, 1 \leq i \leq n)}{\Omega : \Pi : P_\alpha(\boldsymbol{E}) * \Sigma \vdash_\Phi \psi}\ \left(\begin{array}{l} \Phi_P = \{\Pi_1 : \Sigma_1 \overset{\boldsymbol{z_1}}{\Rightarrow} P(\boldsymbol{x_1}), \ldots, \Pi_n : \Sigma_n \overset{\boldsymbol{z_n}}{\Rightarrow} P(\boldsymbol{x_n})\} \text{ and for all } 1 \leq i \leq n\text{:} \\ \quad \Pi_i' : \Sigma_i' \text{ is } \Pi_i : \Sigma_i \text{ with variables } \boldsymbol{z_i} \text{ and labels freshened, and arguments } \boldsymbol{E} \\ \qquad \text{substituted for parameters } \boldsymbol{x_i} \\ \quad \Omega_i = \{\beta < \alpha \mid \beta \in \mathsf{lab}(\Sigma_i')\} \end{array}\right)$$

$$\text{(UnfoldR):}\ \frac{\phi \vdash_\Phi \exists \boldsymbol{\beta}.\{\beta < \alpha \mid \beta \in \boldsymbol{\beta}\} : \exists \boldsymbol{z}.\, \Pi[\boldsymbol{E}/\boldsymbol{x}] : \Sigma[\boldsymbol{E}/\boldsymbol{x}]}{\phi \vdash_\Phi P_\alpha(\boldsymbol{E})}\ \left(\begin{array}{c} \Pi : \Sigma \overset{\boldsymbol{z}}{\Rightarrow} P(\boldsymbol{x}) \in \Phi \\ \alpha \notin \boldsymbol{\beta},\ \boldsymbol{\beta} = \mathsf{lab}(\Pi : \Sigma) \\ \boldsymbol{E} \cap \boldsymbol{z} = \emptyset \end{array}\right)$$

Figure 6: Proof rules for entailment.

6. Evaluation

We tested our tool on a number of hand-crafted example programs consisting of recursive procedures operating on various flavours of lists and trees; we also included some iterative examples. Furthermore, we translated a number of examples from the Recursive Java Bytecode suite in the latest version (10.3) of the Termination Problems Database [33] into a form that can be parsed by our tool. These examples are each individually on the order of up to 100 lines of code comprising up to 6 procedures. We also compared the performance of our prover against two other state-of-the-art tools that prove termination of heap-manipulating procedural code: HIPTNT+ and APROVE. The results are shown in Figs. 7 and 8. For each benchmark test we give the running time in seconds for each tool, as well as the number of annotations required as a percentage of the number of lines of code (no annotation percentage is given for APROVE, since it does not require any). All tests were carried out on a 2.93GHz Intel Core i7-870 with 8GB RAM.

The examples that we treat include non-trivial recursion schemes. For example, the `Alternate` benchmark contains a procedure that creates a binary tree using the left and right subtrees of its two inputs in an alternating fashion; the recursive calls also swap the arguments and so proving termination requires a lexicographic measure, which is discovered by our tool. The `Shuffle` benchmark rearranges the elements of a linked list by first reversing the tail of its input, and then making a recursive call on the result. Thus, it contains a nested recursion over which it is crucial to know that the termination measure of the outer recursion is preserved. Other examples require complex reasoning about the shape of the heap resulting from procedure calls; these include the `SharingAnalysisRec`, `CyclicAnalysisRec`, and `TwoWay` benchmarks, with the latter two requiring to prove that a cyclic heap structure and non-terminating loop, respectively, is unreachable. Two of our hand-crafted benchmarks implement different forms of queue data structures, and we also verified an implementation of a union-find data structure in which the representative elements are indicated using a self-loop. Since our tool does not currently support numeric features we focussed on examples containing solely or predominantly heap-manipulating features. However, we did model some arithmetic-based control flow using linked-lists to stand for natural numbers; this includes an implementation of the Ackermann function.

As a caveat, we note that our tool does not operate directly on C code or Java bytecode, but rather on translations of such programs into our basic procedural `while` language; however, these translations faithfully model the operation and recursion schemes of the original benchmarks.

Compared with HIPTNT+ and APROVE, our tool displayed much shorter execution times on almost all of the examples in our benchmark suite. Notably, our tool easily

Benchmark Suite / Test	Time (sec) / % Annotated AProVE	CYCLIST
Costa_Julia_09-recursive		
Ackermann	3.82	0.14 (18%)
BinarySearchTree (tree copy)	1.41	0.95 (13%)
BTree	1.77	0.03 (22%)
List	1.43	1.74 (19%)
Julia_10_Recursive		
AckR	3.22	0.14 (18%)
BTreeR	2.68	0.03 (22%)
Test8 (bubble sort)	2.95	0.97 (13%)
AProVE_11_recursive		
CyclicAnalysisRec	2.61	5.21 (27%)
RotateTree	5.86	0.32 (14%)
SharingAnalysisRec	2.47	4.72 (16%)
UnionFind	TIMEOUT	1.21 (25%)
BOG_RTA_11		
Alternate	5.47	1.47 (12%)
AppE	2.19	0.09 (23%)
BinTreeChanger	3.38	3.33 (20%)
CAppE	2.04	1.78 (25%)
ConvertRec	3.72	0.06 (38%)
DupTreeRec	4.18	0.03 (20%)
GrowTreeR	3.53	0.05 (20%)
MirrorBinTreeRec	4.96	0.02 (22%)
MirrorMultiTreeRec	5.16	0.63 (33%)
SearchTreeR	2.74	0.34 (14%)
Shuffle	(MAYBE) 11.72	0.21 (29%)
TwoWay	1.94	0.02 (25%)

Figure 7: Comparison with AProVE on TPDB v10.3 Java_Bytecode_Recursive benchmarks

handles the `UnionFind` benchmark, which AProVE cannot. We also note that `AProVE` returns the answer 'MAYBE' for the `Shuffle` example, rather than a definite termination result. HIPTNT+ also handles these two examples. The relative length of the execution time for the iterative version of the binary tree traversal custom benchmark is due to its relative complexity; re-establishing the loop invariant requires an inductive entailment to be proven.

Benchmark test	Time (sec) / % Annotated HIPTNT+	CYCLIST
traverse acyclic linked list	0.31 (25%)	0.02 (33%)
traverse cyclic linked list	0.52 (29%)	0.02 (38%)
append acyclic linked lists	0.36 (25%)	0.03 (10%)
TPDB Shuffle	1.79 (22%)	0.21 (29%)
TPDB Alternate	6.33 (13%)	1.47 (12%)
TPDB UnionFind	4.03 (26%)	1.21 (25%)

(a) Comparison with HIPTNT+ benchmarks

Benchmark test	Time (sec) / % Annotated
traverse acyclic linked list	0.80 (100%)
traverse cyclic linked list	0.15 (100%)
traverse binary tree	2.99 (50%)
reverse acyclic linked list	0.09 (20%)
deallocated linked list	0.03 (25%)

(b) CYCLIST custom benchmarks (iterative)

Benchmark test	Time (sec) / % Annotated
reverse linked list	0.13 (36%)
reverse linked list (tail recursive)	0.04 (11%)
reverse linked list (using append)	0.05 (11%)
deallocate linked list	0.02 (29%)
deallocate binary tree	0.02 (25%)
append cyclic linked list	0.03 (10%)
filter linked list	0.03 (13%)
partition linked list	0.03 (8%)
remove linked list tail (example in paper)	0.05 (17%)
queue data structure	0.19 (20%)
functional queue data structure	0.26 (14%)

(c) CYCLIST custom benchmarks (recursive)

Figure 8: Comparison with HIPTNT+ (table (a)) and evaluation on custom benchmarks (tables (b) and (c))

7. Related Work

There are a number of other existing tools that verify termination of procedural, heap-manipulating code, some of which we have already mentioned in the previous section. Many of these are, like our system, built on top of Hoare-style program logics. Closest to our work in this respect is HIPTNT+ [23] which extends a Hoare-style separation logic system with temporal operators expressing termination and both possible and definite non-termination, and is able to infer such temporal predicates for procedural programs. Also like our work, HIPTNT+ requires pre-/postcondition annotations for heap-manipulating procedures, and requires arithmetic parameters to be incorporated into inductive predicates (reflecting e.g. the length of a list) in order to support the inference of termination measures. This is comparable to our use of ordinal-valued labels. While it is mentioned in [23] that existing techniques for inferring separation logic specifications (e.g. bi-abduction [16]) can be integrated with the termination inference, HIPTNT+ does not currently implement this.

The AProVE termination prover [20] takes an alternative approach, handling C code and Java bytecode by transforming it into a term rewriting system and then using existing techniques for proving termination of these. For Java it can also prove non-termination, but it does consider termination due to the raising of a checked exception as safe termination. An advantage of AProVE is that it does not require procedure annotations; however, since it is not based on a program logic, it does not allow for the verification of functional correctness. In contrast, our analysis also allows to verify that the final state of the program conforms to some given shape predicate. A further advantage of basing our analysis on a logic with user-defined predicates is that, effectively, our abstract domain is easily modified. In contrast, extending the abstract domain of AProVE to handle the Union-Find example would likely not be straightforward. The Julia [29] and COSTA [3] tools also prove termination of Java bytecode in a similar way via an encoding as constraint logic programs.

The Verifast tool verifies the safety of heap manipulating programs using separation logic, and has recently incorporated termination checking using an approach similar to our ordinal-valued labels [21]. However, Verifast is closer to an interactive proof assistant than to a fully-automatic verification tool. The Dafny integrated programming language and verifier [25] affords a greater level of automation, translating programs and their assertions into the Boogie intermediate verification language, which uses the Z3 SMT solver to discharge verification conditions. Similar to HIPTNT+, Dafny infers termination based on arithmetic measures and requires heap-based termination measures (e.g. linked-list length) to be manually associated to arithmetic parameters. Boström et al. [7] have also recently described a termination analysis for concurrent imperative programs via an encoding to Boogie which requires explicit arithmetical measures, and includes termination of sequential programs as a special case. Pinto et al. consider an alternative approach based on explicit ordinal-valued measures [17]. However, to our knowledge, neither of these latter two analyses have been implemented.

The MUTANT tool [6], part of the TERMINATOR/T2 termination prover, uses an abstract interpretation for a list-based fragment separation logic to prove termination of loops by synthesis of ranking functions. The THOR tool [26] extends this approach to procedural programs and also allows the abstraction to be guided by user-defined inductive predicates in separation logic.

8. Conclusions and Future Work

We have defined a cyclic proof system for verifying the safe termination of procedural programs written in a C-like syntax. Loops and recursion are handled naturally by cyclic proof, and termination is guaranteed by the well-foundedness of the inductively defined predicates in program specifications. We have implemented a prototype tool that can automatically prove termination of such programs, when its component procedures are annotated with pre- and postconditions. We have evaluated our tool using a number of non-trivial example programs taken from standard benchmarks, demonstrating that it performs favourably compared with existing related tools.

Our work can be seen as a natural extension of the system for simple `while` programs in [12] and its corresponding implementation [14], treating programs with arbitrary, possibly recursive procedures. The extension to procedures requires both the proof system and its implementation to be extended in a number of ways (cf. Sections 4 and 5). The work we describe in this paper comprises a significant development over what could previously be achieved using cyclic proof techniques in practice. Key to its practicability is the ability to treat procedures directly (as opposed to in-lining) and compositionally; here we show how this can be achieved within the cyclic proof framework. Our tool puts cyclic proof-based termination reasoning roughly on a par with the current state-of-the-art, as discussed in Sections 5 and 7, and lays the foundations for further development of the technique.

The ordinal variable annotations in our system, specifying how predicate approximations are related over procedure calls, are analogous to the arithmetic parameters required in most other approaches (see Section 7). We believe that these annotations are actually unnecessary, and that the constraints on ordinal labels might, in principle, be inferred directly from the structure of a cyclic proof. We consider this direction a high priority for future work. We would also like to investigate the possibility of inferring entire pre-/postcondition specifications, most probably using *biabduction* [16], as well as extending the assertion language supported by our implementation in order to verify functional properties of programs, rather than just shape properties.

Acknowledgments

This work was supported by EPSRC grant EP/K040049/1.

References

[1] CYCLIST: software distribution. `http://www.cyclist-prover.org/`.

[2] `www.github.com/ngorogiannis/cyclist/releases`.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Proceedings of FMCO-6*, pages 113–132, 2007.

[4] J. Berdine, C. Calcagno, and P. O'Hearn. A Decidable Fragment of Separation Logic. In *Proc. FSTTCS-24*, volume 3328 of *LNCS*, pages 97–109. Springer, 2004.

[5] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *Proc. APLAS-3*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.

[6] J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic Termination Proofs for Programs with Shape-shifting Heaps. In *Proceedings of CAV-18*, volume 4144 of *LNCS*, pages 386–400. Springer, 2006.

[7] P. Boström and P. Müller. Modular Verification of Finite Blocking in Non-terminating Programs. In *Proceedings of ECOOP-29*, pages 639–663, 2015.

[8] J. Brotherston. Cyclic Proofs for First-Order Logic with Inductive Definitions. In *Proceedings of TABLEAUX-14*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.

[9] J. Brotherston. Formalised Inductive Reasoning in the Logic of Bunched Implications. In *Proc. SAS-14*, volume 4634 of *LNCS*, pages 87–103. Springer-Verlag, 2007.

[10] J. Brotherston and N. Gorogiannis. Cyclic Abduction of Inductively Defined Safety and Termination Preconditions. In *SAS-21*, volume 8723 of *LNCS*, pages 68–84. Springer, 2014.

[11] J. Brotherston and A. Simpson. Sequent Calculi for Induction and Infinite Descent. *Journal of Logic and Computation*, 21 (6):1177–1216, December 2011.

[12] J. Brotherston, R. Bornat, and C. Calcagno. Cyclic Proofs of Program Termination in Separation Logic. In *Proceedings of POPL-35*, pages 101–112. ACM, 2008.

[13] J. Brotherston, D. Distefano, and R. L. Petersen. Automated Cyclic Entailment Proofs in Separation Logic. In *Proceedings of CADE-23*, volume 6803 of *LNAI*, pages 131–146. Springer, 2011.

[14] J. Brotherston, N. Gorogiannis, and R. L. Petersen. A Generic Cyclic Theorem Prover. In *Proceedings of APLAS-10*, LNCS, pages 350–367. Springer, 2012.

[15] J. Brotherston, N. Gorogiannis, M. Kanovich, and R. Rowe. Model Checking for Symbolic-Heap Separation Logic with Inductive Predicates. In *Proc. POPL-43*. ACM, 2016.

[16] 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.

[17] P. da Rocha Pinto, T. Dinsdale-Young, P. Gardner, and J. Sutherland. Modular Termination Verification for Non-blocking Concurrency. In *Proc. ESOP*, pages 176–201, 2016.

[18] M. Dam and D. Gurov. μ-Calculus with Explicit Points and Approximations. *Journal of Logic and Computation*, 12(2): 255–269, April 2002.

[19] R. W. Floyd. Assigning Meanings to Programs. In *Proc. Amer. Math. Soc.*, volume 19 of *Symposia in Applied Mathematics*, pages 19–31, 1967.

[20] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving Termination of Programs Automatically with AProVE. In *IJCAR-7*, pages 184–191, 2014.

[21] B. Jacobs, D. Bosnacki, and R. Kuiper. Modular Termination Verification. In *Proc. ECOOP-29*, pages 664–688, 2015.

[22] Q. L. Le, C. Gherghina, S. Qin, and W. Chin. Shape Analysis via Second-Order Bi-Abduction. In *Proceedings of CAV-26*, volume 8559 of *LNCS*, pages 52–68. Springer, 2014.

[23] T. C. Le, S. Qin, and W.-N. Chin. Termination and Non-termination Specification Inference. In *Proceedings of PLDI-15*, pages 489–498, 2015.

[24] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The Size-change Principle for Program Termination. In *Proceedings of POPL*, pages 81–92. ACM, 2001.

[25] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR-16*, pages 348–370, 2010.

[26] S. Magill, M. Tsai, P. Lee, and Y. Tsay. Automatic Numeric Abstractions for Heap-manipulating Programs. In *Proceedings of POPL*, pages 211–222, 2010.

[27] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of LICS-17*, pages 55–74. IEEE Computer Society, 2002.

[28] L. Santocanale. A Calculus of Circular Proofs and its Categorical Semantics. In *Proc. FOSSACS*, volume 2303 of *LNCS*, pages 357–371. Springer-Verlag, 2002.

[29] F. Spoto, F. Mesnard, and É. Payet. A Termination Analyzer for Java Bytecode Based on Path-length. *ACM Trans. Program. Lang. Syst.*, 32(3), 2010.

[30] C. Sprenger and M. Dam. On the Structure of Inductive Reasoning: Circular and Tree-shaped Proofs in the μ-calculus. In *Proceedings of FOSSACS-6*, volume 2620 of *LNCS*, pages 425–440. Springer-Verlag, 2003.

[31] C. Sprenger and M. Dam. On Global Induction Mechanisms in a μ-calculus with Explicit Approximations. *ITA*, 37(4): 365–391, 2003.

[32] M. Tatsuta and W. Chin. Completeness of Separation Logic with Inductive Definitions for Program Verification. In *Proc. SEFM*, pages 20–34, 2014.

[33] TPDB. Termination Problems Database. `http://termination-portal.org/wiki/TPDB`.

[34] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 1937.

[35] H. Yang and P. O'Hearn. A Semantic Basis for Local Reasoning. In *Proc. FOSSACS-5*, pages 402–416. Springer, 2002.