Cyclic Proofs of Program Termination in Separation Logic

James Brotherston*¹, Richard Bornat², and Cristiano Calcagno¹

¹Imperial College, London
²Middlesex University, London

*Me

10 January, 2008
Overview

• We give a new method for proving program termination.
• We consider simple, heap-manipulating imperative programs.
• We use separation logic with inductive definitions to express termination preconditions.
• Our proofs of termination are cyclic proofs: cyclic derivations satisfying a soundness condition.
**TOY-C: a simple imperative programming language**

\[
E ::= \text{nil} \mid x \ (x \in \text{Var}) \mid \ldots \\
\text{Cond} ::= E = E \mid E \neq E \\
C ::= x := E \mid x := [E] \mid [E] := E \mid x := \text{new()} \\
\quad \mid \text{free}(E) \mid \text{if Cond goto } j \mid \text{stop}
\]

A program in TOY-C is a finite sequence \(1 : C_1; \cdots n : C_n\).

**Example (Linked list traversal)**

1: if \(x = \text{nil}\) goto 4, 2: \(x := [x]\), 3: goto 1, 4: stop

\[
x \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \text{nil}
\]
Semantics of \textit{TOY–C}

• A program \textit{state} is a triple \((i, s, h)\), where \(i\) is an index of the program, \(s\) is a stack and \(h\) is a heap.

• The \textit{semantics} of \textit{TOY–C} programs is then given by a “one-step” binary relation \(\rightsquigarrow\) on program states.

• We write \((i, s, h)\downarrow\) to mean there is no infinite \(\rightsquigarrow\)-sequence \((i, s, h)\rightsquigarrow\ldots\), i.e., the program \textit{terminates} (without faulting) when started in the state \((i, s, h)\).
A Hoare proof system for termination

- We write termination judgements $F \vdash_i \downarrow$ where $i$ is a program label and $F$ is a formula of separation logic with inductive predicates.
- E.g. we can define (possibly cyclic) linked list segments in separation logic by the definition:
  
  \[
  \text{emp} \Rightarrow \text{ls } x \ x \\
  x \mapsto x' \ast \text{ls } x' \ y \Rightarrow \text{ls } x \ y
  \]

- $F \vdash_i \downarrow$ is valid if for all $s, h$. $s, h \models F$ implies $(i, s, h)\downarrow$
We have two types of proof rule:

1. **logical rules** similar to *left-introduction* rules in sequent calculus. Each inductive predicate also has a *case-split* rule, e.g. for `ls`:

   \[
   \Gamma(t_1 = t_2 \land \text{emp}) \vdash_i \quad \Gamma(t_1 \leftrightarrow x \ast ls \ast x t_2) \vdash_i \quad \Gamma(\text{ls} \ast t_1 t_2) \vdash_i \quad x \text{ fresh (Case } ls)
   \]

2. **symbolic execution rules** which simulate commands. E.g.:

   \[
   \text{Cond} \land F \vdash_j \quad \neg\text{Cond} \land F \vdash_{i+1} \quad C_i \equiv \text{if Cond goto } j
   \]

Paths in a derivation thus correspond to *program computations*. 
Cyclic proofs of termination judgements

- A cyclic pre-proof is a regular, infinite derivation tree, represented as a cyclic graph:

- A cyclic proof is a pre-proof satisfying the condition: Every infinite path in the pre-proof has a tail on which one can “trace” some inductive definition that is unfolded infinitely often (using the case-split rules)
Reversing a “frying-pan” list

• The classical list reverse algorithm is:

1. \( y := \text{nil} \)
2. \( \text{if } x = \text{nil} \text{ goto 8} \)
3. \( z := x \)
4. \( x := [x] \)
5. \( [z] := y \)
6. \( y := z \)
7. \( \text{goto 2} \)
8. \( \text{stop} \)

• The invariant for this algorithm given a cyclic list is:

\[ \exists k_1, k_2, k_3 \cdot \]
\[ (\text{ls } x j \ast \text{ls } y \text{nil} \ast j \mapsto k_1 \ast \text{ls } k_1 j) \lor \]
\[ (\text{ls } k_2 \text{nil} \ast j \mapsto k_2 \ast \text{ls } x j \ast \text{ls } y j) \lor \]
\[ (\text{ls } x \text{nil} \ast \text{ls } y j \ast j \mapsto k_3 \ast \text{ls } k_3 j) \]

• We want to prove that the invariant implies termination.
Reversing a “frying-pan” list — the cyclic proof
Properties of the proof system

**Theorem (Soundness)**
If there is a cyclic proof of \( F \vdash i \downarrow \) then \( F \vdash i \downarrow \) is valid.

**Proposition**
It is **decidable** whether a cyclic pre-proof is a cyclic proof, i.e. whether it satisfies the soundness condition.

**Theorem (Relative completeness)**
If \( F \vdash i \downarrow \) is valid then there is a formula \( G \) such that \( F \vdash G \) is a valid implication of separation logic and:

\[
F \vdash G \text{ provable} \Rightarrow F \vdash i \downarrow \text{ provable}
\]
Conclusion

• We have developed a novel method for proving program termination, based on cyclic proof.
• Use of the soundness condition for cyclic proofs means that termination measures are employed only implicitly.
• We plan to extend the programming language we consider to include e.g. procedures.
• We could also consider proving arbitrary postconditions.
• Possibility of adapting the approach to other programming paradigms, e.g. functional programming.