Cyclic Proofs of Program Termination in Separation Logic

James Brotherston[*1], Richard Bornat[2],
and Cristiano Calcagno[1]

[1]Imperial College, London

[2]Middlesex 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.

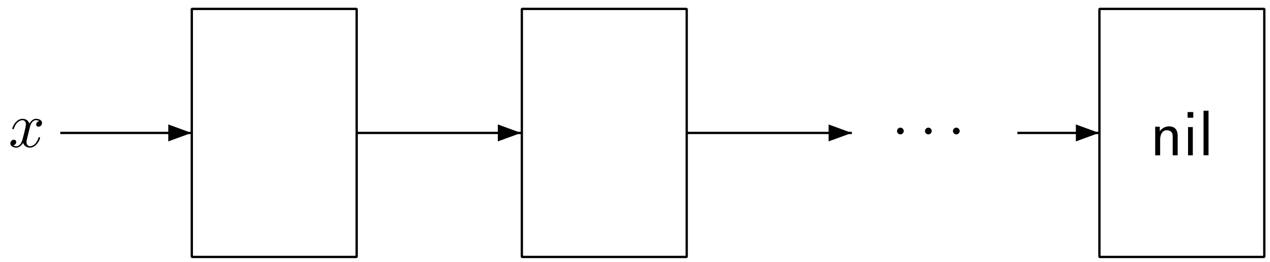
TOY-C: a simple imperative programming language

$$
\begin{array}{rcl}
E & ::= & \mathsf{nil} \mid x \ \ (x \in \mathsf{Var}) \mid \ldots \\
Cond & ::= & E = E \mid E \neq E \\
C & ::= & x := E \mid x := [E] \mid [E] := E \mid x := \mathtt{new}() \\
 & & \mid \mathtt{free}(E) \mid \mathtt{if}\, Cond\, \mathtt{goto}\, j \mid \mathtt{stop}
\end{array}
$$

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

Example (Linked list traversal)

$$1 : \mathtt{if}\, x = \mathsf{nil}\, \mathtt{goto}\, 4, \quad 2 : x := [x], \quad 3 : \mathtt{goto}\, 1, \quad 4 : \mathtt{stop}$$



Semantics of **TOY-C**

- A program state is a triple (i, s, h), where i is a index of the program, s is a stack and h is a heap.
- The semantics of `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 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:

$$\begin{array}{rcl} \mathsf{emp} & \Rightarrow & \mathtt{ls}\, x\, x \\ x \mapsto x' * \mathtt{ls}\, x'\, y & \Rightarrow & \mathtt{ls}\, x\, y \end{array}$$

- $F \vdash_i \downarrow$ is valid if for all s, h. $s, h \models F$ implies $(i, s, h)\downarrow$

Proof rules

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`:

$$\frac{\Gamma(t_1 = t_2 \wedge \mathsf{emp}) \vdash_i \downarrow \quad \Gamma(t_1 \mapsto x * \mathtt{ls}\, x\, t_2) \vdash_i \downarrow}{\Gamma(\mathtt{ls}\, t_1\, t_2) \vdash_i \downarrow} \; x \text{ fresh (Case } \mathtt{ls}\text{)}$$

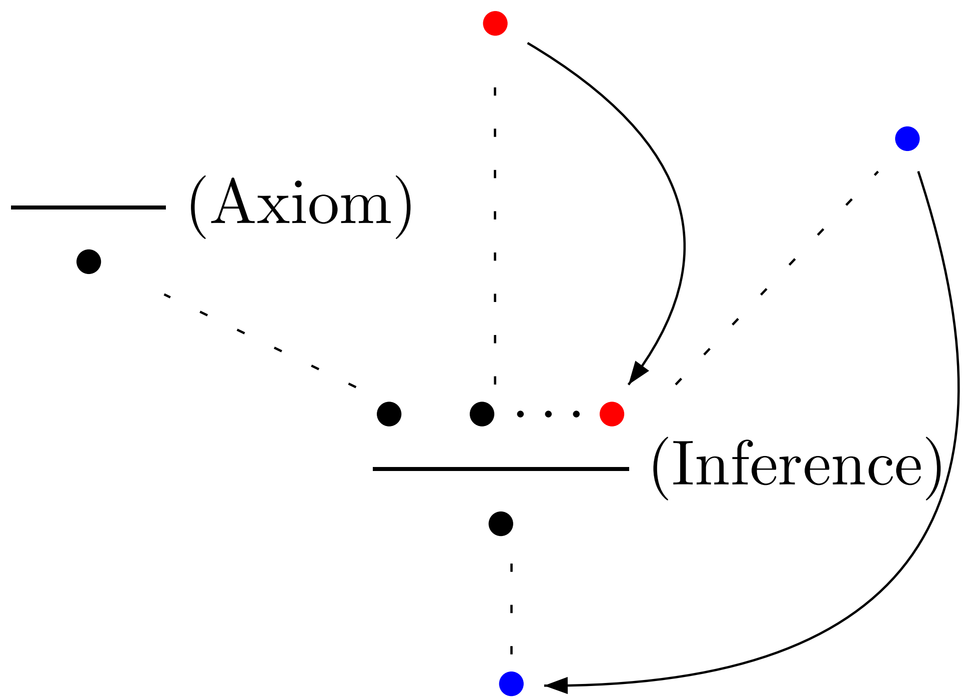
2. symbolic execution rules which simulate commands. E.g.:

$$\frac{Cond \wedge F \vdash_j \downarrow \quad \neg Cond \wedge F \vdash_{i+1} \downarrow}{F \vdash_i \downarrow} \; C_i \equiv \mathtt{if}\, Cond\, \mathtt{goto}\, j$$

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)

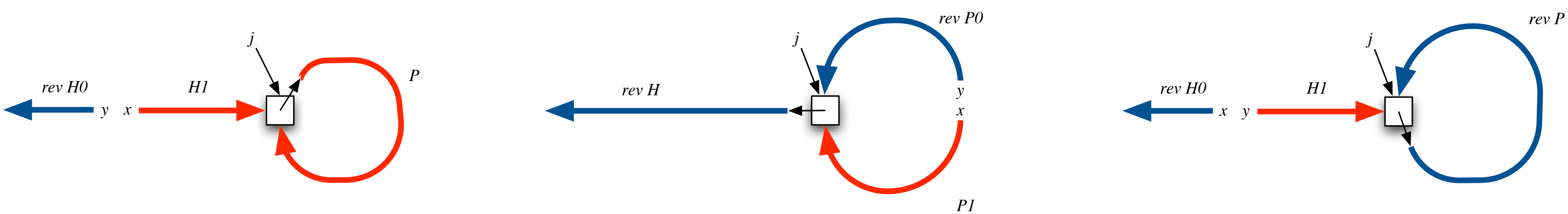
Reversing a "frying-pan" list

- The classical list reverse algorithm is:

1.	$y := \mathsf{nil}$	4.	$x := [x]$	7.	$\texttt{goto}\, 2$
2.	$\texttt{if}\; x = \mathsf{nil}\; \texttt{goto}\, 8$	5.	$[z] := y$	8.	$\texttt{stop}$
3.	$z := x$	6.	$y := z$		

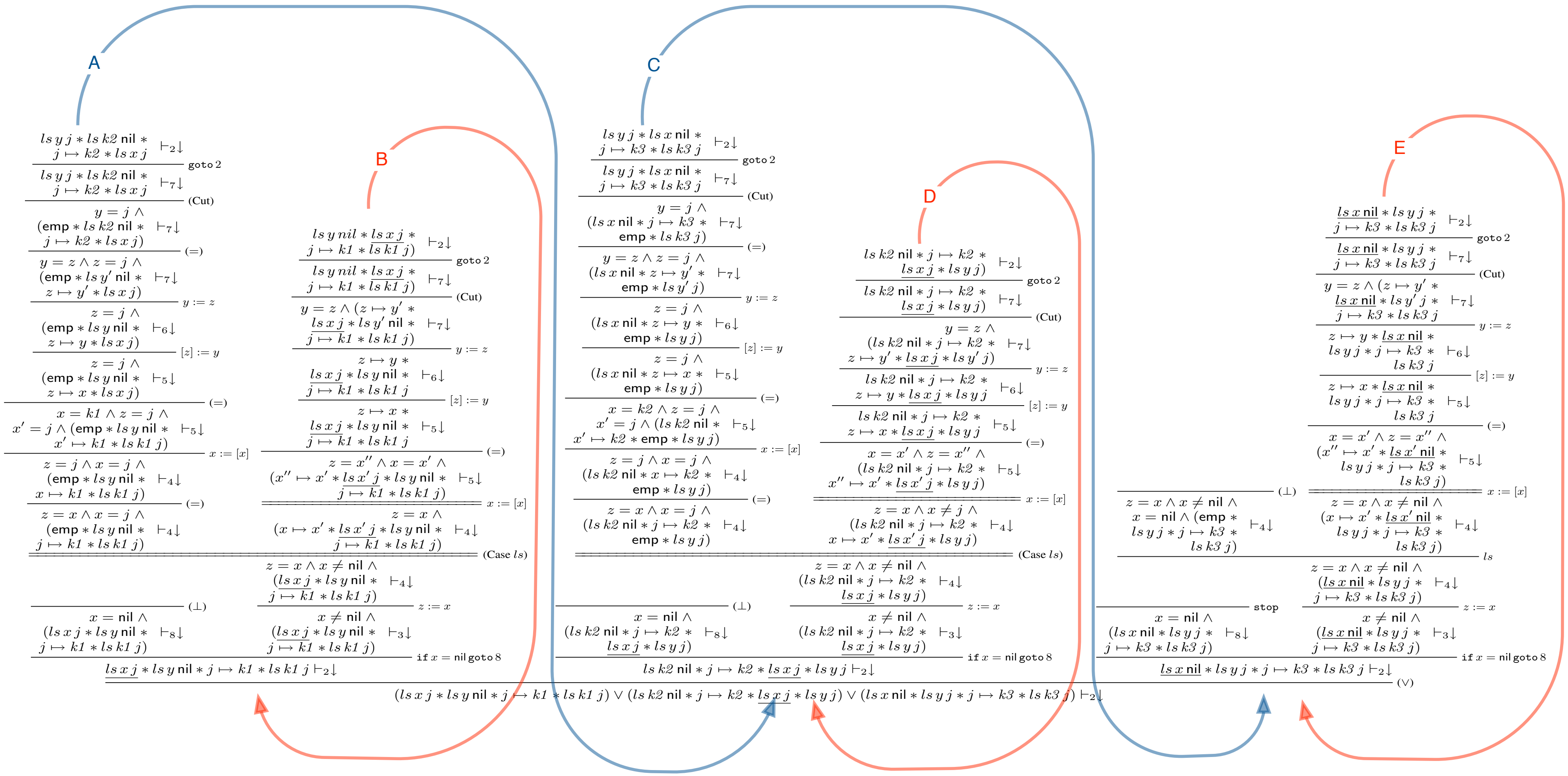
- The invariant for this algorithm given a cyclic list is:

$$\begin{array}{l}\exists k_1, k_2, k_3 . \\ (\texttt{ls}\; x\; j * \texttt{ls}\; y\; \mathsf{nil} * j \mapsto k_1 * \texttt{ls}\; k_1\; j) \vee \\ (\texttt{ls}\; k_2\; \mathsf{nil} * j \mapsto k_2 * \texttt{ls}\; x\; j * \texttt{ls}\; y\; j) \vee \\ (\texttt{ls}\; x\; \mathsf{nil} * \texttt{ls}\; y\; j * j \mapsto k_3 * \texttt{ls}\; k_3\; j)\end{array}$$



- 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.