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

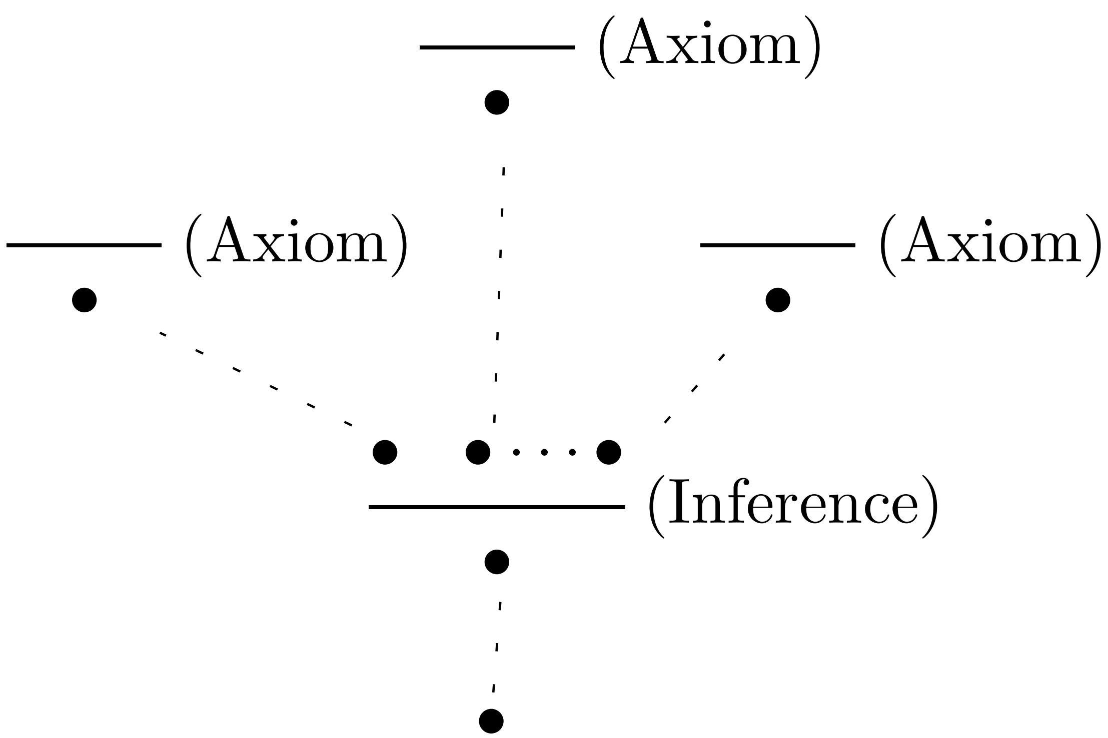
12 December, 2007

Overview

- We give a new method for proving program termination, based upon cyclic proof.
- We consider simple imperative programs that may access the heap.
- We use separation logic to express termination preconditions, which typically also involve some inductive definitions.
- This work will appear in a paper with Bornat and Calcagno at POPL 2008.

Tree proof vs. cyclic proof (1)

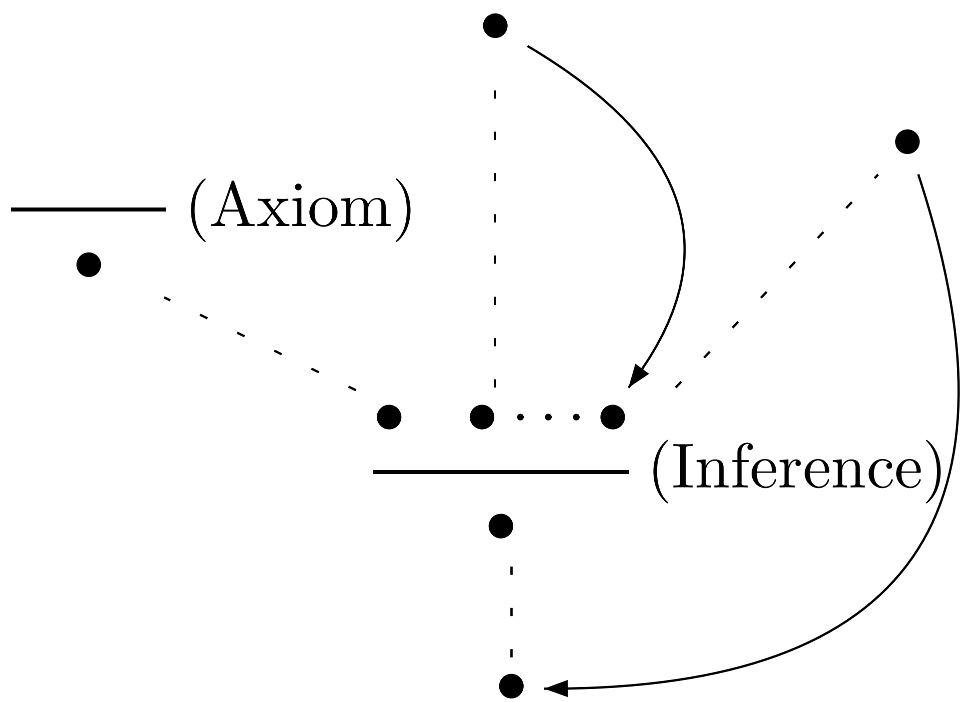
- Usually a proof is a finite tree of sequents (•):



- Soundness of such proofs follows from the local soundness of each inference rule / axiom.

Tree proof vs. cyclic proof (2)

- A cyclic pre-proof is a regular, infinite tree of sequents, usually represented as a rooted cyclic graph:



- Cyclic pre-proofs are not sound in general — we need some extra condition.
- Cyclic proof = cyclic pre-proof $\mathcal{P}$ + soundness condition $S(\mathcal{P})$.

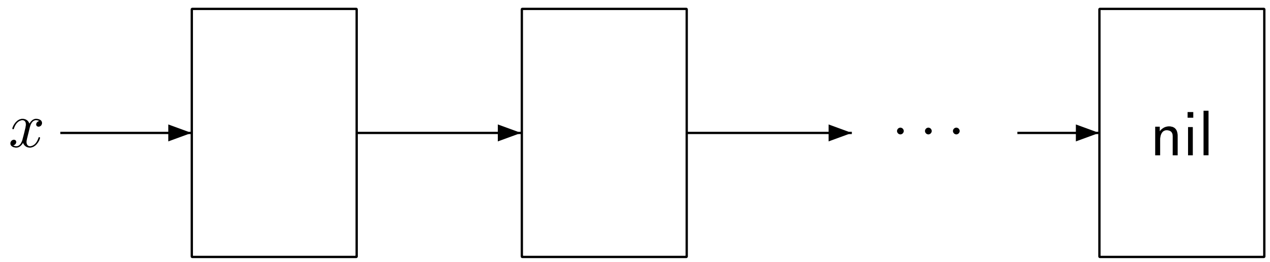
TOY-C: a simple imperative programming language

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

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

Example (Linked list traversal)

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



Semantics of **TOY-C** *(1)*

- We use a basic RAM-type model.
- Fix sets of variables Var, values Val and locations $\mathsf{Loc} \subset \mathsf{Val}$.
- A stack is a function $s : \mathsf{Var} \to \mathsf{Val}$.
- A heap is a partial, finitely-defined function $h : \mathsf{Loc} \rightharpoonup_{fin} \mathsf{Val}$. We write e for the empty heap and $\circ$ for composition of disjoint heaps.
- A program state is then a triple (i, s, h), where i is a index of the program, s is a stack and h is a heap.

Semantics of TOY-C *(2)*

- The semantics of `TOY-C` programs is then given by a "one-step" binary relation $\rightsquigarrow$ on program states. E.g.:

$$\frac{C_i \equiv x := [E] \quad [\![E]\!]s \in dom(h)}{(i, s, h) \rightsquigarrow (i+1, s[x \mapsto h([\![E]\!]s)], h)}$$

$$\frac{C_i \equiv x := [E] \quad [\![E]\!]s \notin dom(h)}{(i, s, h) \rightsquigarrow (fault, s, h)}$$

$$\frac{}{(fault, s, h) \rightsquigarrow (fault, s, h)}$$

- We write $(i, s, h)\downarrow$ to mean there is no infinite $\rightsquigarrow$-sequence $(i, s, h) \rightsquigarrow \ldots$, i.e., the program terminates when started in the state (i, s, h).
- Program faulting is equated in our model with program divergence.

Separation logic

- Separation logic adds new connectives to standard first-order logic, which let us reason about heap resource.
- The proposition emp expresses emptiness of the heap:

$$s, h \models \mathsf{emp} \Leftrightarrow h = e$$

- $*$ characterises heap composition:

$$s, h \models F_1 * F_2 \Leftrightarrow h = h_1 \circ h_2 \text{ and } s, h_1 \models F_1 \text{ and } s, h_2 \models F_2$$

- $-\!\!*$ expresses a property of (fresh) heap addition:

$$s, h \models F_1 -\!\!* F_2 \quad \Leftrightarrow \quad s, h' \models F_1 \text{ and } h' \circ h \text{ defined implies } s, h' \circ h \models F_2 \text{ for all heaps } h'$$

- We also need to alter the usual treatment of predicates: their interpretation should depend on the current heap.

Predicates in separation logic

- For any predicate symbol P of arity k (say) we define its interpretation:

$$[\![P]\!] \subseteq \mathrm{Pow}(\mathsf{Heaps} \times \mathsf{Val}^k)$$

whence we have:

$$s, h \models P\mathbf{t} \Leftrightarrow (h, s(\mathbf{t})) \in [\![P]\!]$$

- We have two types of predicate symbol: ordinary and inductive.
- The interpretation of each ordinary predicate symbol is fixed in our model.
- The interpretation of the inductive predicate symbols is determined by a given set of inductive definitions.

Inductive definitions: an example

The following definition of an inductive predicate `ls` defines (possibly cyclic) linked list segments:

$$\frac{\mathsf{emp}}{\texttt{ls}\, x\, x} \qquad\qquad \frac{x \mapsto x' * \texttt{ls}\, x'\, y}{\texttt{ls}\, x\, y}$$

where $\mapsto$ is an ordinary predicate with interpretation:

$$[\![\mapsto]\!] = \{(h, (v_1, v_2)) \mid dom(h) = \{v_1\} \text{ and } h(v_1) = v_2\}$$

$[\![\texttt{ls}]\!]$ is then the least fixed point of the monotone operator $\varphi_{\texttt{ls}}$ defined by:

$$\begin{array}{rcl} \varphi_{\texttt{ls}}(X) & = & \{(e, (v, v)) \mid v \in \mathsf{Val}\} \\ & \cup & \{(h_1 \circ h_2, (v, v')) \mid (h_1, (v, v'')) \in [\![\mapsto]\!] \\ & & \text{ and } (h_2, (v'', v')) \in X\} \end{array}$$

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. $\Gamma(-)$ is notation for a "context", given by:

$$\Gamma ::= - \mid \Gamma(-) \wedge F \mid \Gamma(-) * F$$

- $F \vdash_i \downarrow$ is valid if:

$$\text{for all } s, h.\ s, h \models F \text{ implies } (i, s, h)\downarrow$$

- We have two types of rules for termination judgements: logical rules, and symbolic execution rules.

Logical rules

- Similar to the left-introduction rules in sequent calculus, e.g.:

$$\frac{\Gamma(F_1) \vdash_i \downarrow \quad \Gamma(F_2) \vdash_i \downarrow}{\Gamma(F_1 \vee F_2) \vdash_i \downarrow} (\vee\mathrm{I}) \qquad \frac{\Gamma(F_2) \vdash_i \downarrow}{\Gamma(F * (F_1 -\!\!* F_2)) \vdash_i \downarrow} \; F \vdash F_1 \; (-\!\!*\mathrm{I})$$

- Each inductive predicate has a case-split rule obtained from its definition. E.g. the definition of `ls`:

$$\frac{\mathsf{emp}}{\mathtt{ls}\, x\, x} \qquad \frac{x \mapsto x' * \mathtt{ls}\, x'\, y}{\mathtt{ls}\, x\, y}$$

gives the following case-split rule for `ls`:

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

Symbolic execution rules

- These encapsulate the effect of executing a single program command. E.g.:

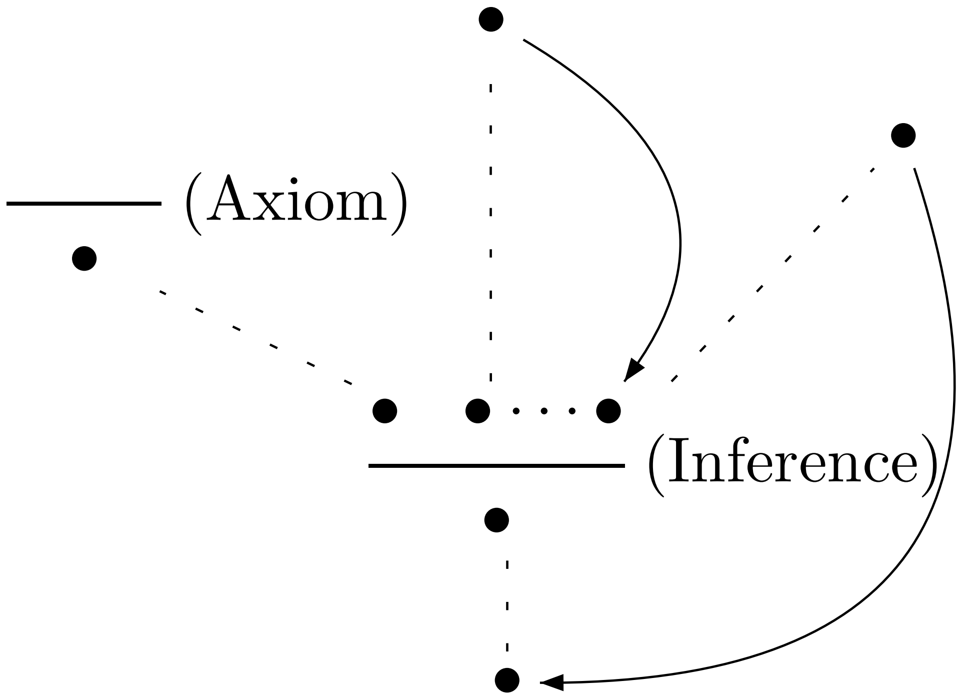
$$\frac{Cond \wedge F \vdash_j \downarrow \quad \neg Cond \wedge F \vdash_{i+1} \downarrow}{F \vdash_i \downarrow} \; C_i \equiv \texttt{if}\, Cond\, \texttt{goto}\, j$$

$$\frac{x = t[x'/x] \wedge (E \mapsto t * F)[x'/x] \vdash_{i+1} \downarrow}{E \mapsto t * F \vdash_i \downarrow} \; C_i \equiv x := [E]$$

$$\frac{F \vdash_{i+1} \downarrow}{E \mapsto t * F \vdash_i \downarrow} \; C_i \equiv \texttt{free}(E)$$

Cyclic proofs of termination judgements

- Recall the notion of a cyclic pre-proof:



- A cyclic proof is a pre-proof satisfying the following condition (stated informally):

> *For every infinite path in the pre-proof one can "trace" some inductive definition along the path, and moreover this definition is unfolded infinitely often (using the case-split rules)*

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:*

$$F \vdash G \text{ provable } \Rightarrow F \vdash_i \downarrow \text{ provable}$$

Example: termination of linked list traversal

Recall the TOY-C program for traversing a linked list:

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

We give a pre-proof of $\texttt{ls}\ x\ \mathsf{nil} \vdash_1 \downarrow$:

$$\frac{(\dagger)\quad \texttt{ls}\ x\ \mathsf{nil} \vdash_1 \downarrow}{\texttt{ls}\ x\ \mathsf{nil} \vdash_3 \downarrow}\ (\texttt{goto})$$

$$\frac{\texttt{ls}\ x\ \mathsf{nil} \vdash_3 \downarrow}{\top \wedge x{\neq}\mathsf{nil} \wedge (x'' \mapsto x * \texttt{ls}\ x\ \mathsf{nil}) \vdash_3 \downarrow}\ (\text{Weak})$$

$$\frac{\top \wedge x{\neq}\mathsf{nil} \wedge (x'' \mapsto x * \texttt{ls}\ x\ \mathsf{nil}) \vdash_3 \downarrow}{x = x' \wedge x{\neq}\mathsf{nil} \wedge (x'' \mapsto x' * \texttt{ls}\ x'\ \mathsf{nil}) \vdash_3 \downarrow}\ (=)$$

$$\frac{x = x' \wedge x{\neq}\mathsf{nil} \wedge (x'' \mapsto x' * \texttt{ls}\ x'\ \mathsf{nil}) \vdash_3 \downarrow}{x \neq \mathsf{nil} \wedge (x{\mapsto}x' * \texttt{ls}\ x'\ \mathsf{nil}) \vdash_2 \downarrow}\ (\texttt{-:=[-]})$$

$$\frac{x \neq \mathsf{nil} \wedge (x{\mapsto}x' * \texttt{ls}\ x'\ \mathsf{nil}) \vdash_2 \downarrow}{x \neq \mathsf{nil} \wedge \texttt{ls}\ x\ \mathsf{nil} \vdash_2 \downarrow}\ (\text{Case}\ \texttt{ls})$$

$$\frac{}{x = \mathsf{nil} \wedge \texttt{ls}\ x\ \mathsf{nil} \vdash_4 \downarrow}\ (\texttt{stop})$$

$$\frac{x = \mathsf{nil} \wedge \texttt{ls}\ x\ \mathsf{nil} \vdash_4 \downarrow \qquad x \neq \mathsf{nil} \wedge \texttt{ls}\ x\ \mathsf{nil} \vdash_2 \downarrow}{(\dagger)\quad \texttt{ls}\ x\ \mathsf{nil} \vdash_1 \downarrow}\ (\texttt{if})$$

Note that there is only one infinite path, which goes around the loop and has a progressing trace (highlighted). So this pre-proof is indeed a cyclic proof.

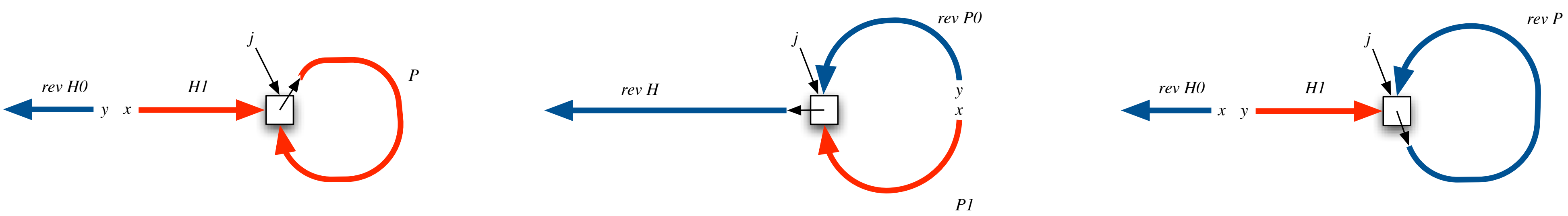
Reversing a "frying-pan" list

- The classical list reverse algorithm is:

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

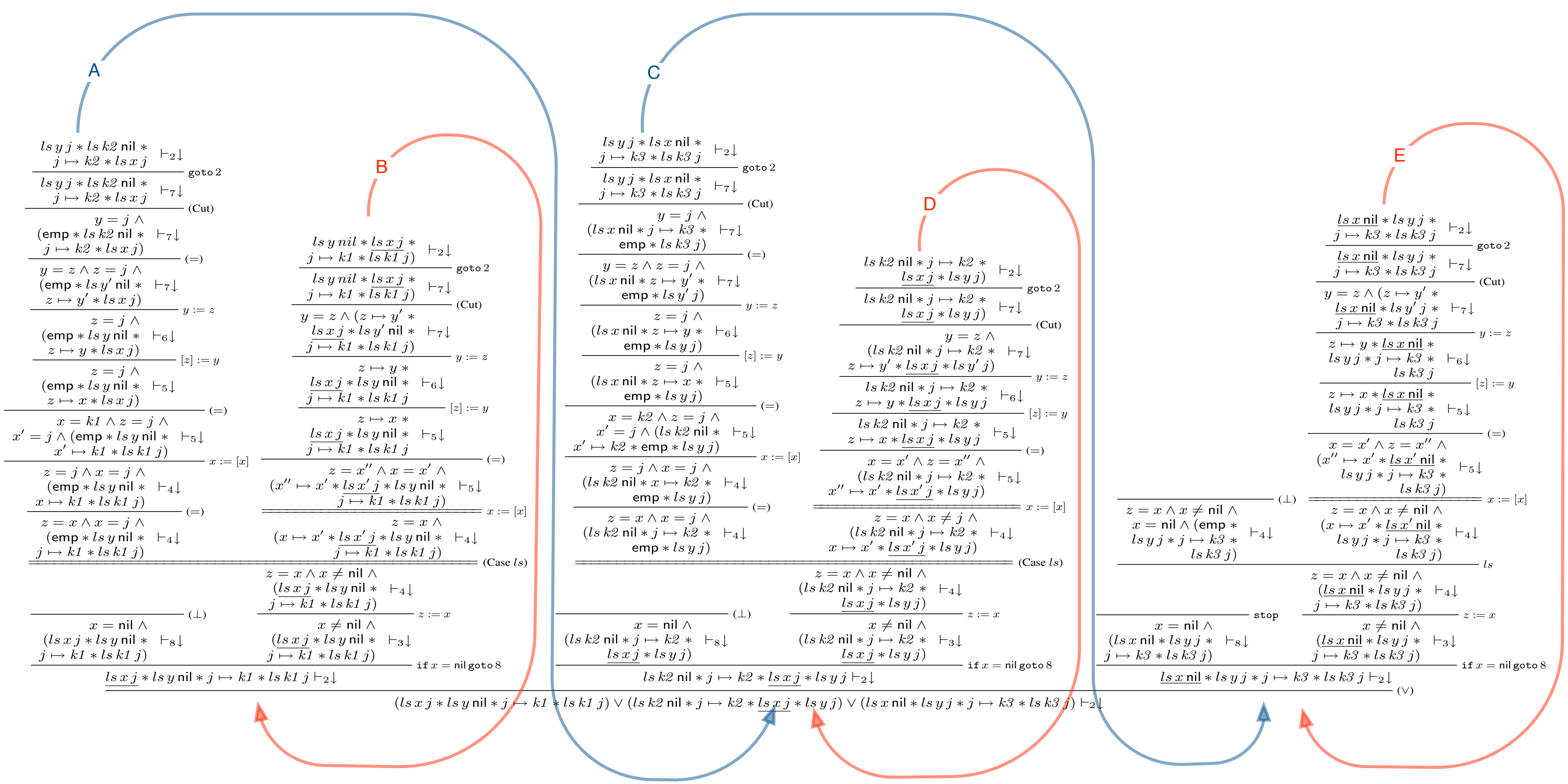
- The invariant for this algorithm given a cyclic list is:

$$\begin{array}{l}\exists k1, k2, k3 \cdot \\ (\mathtt{ls}\, x\, j * \mathtt{ls}\, y\, \mathsf{nil} * j \mapsto k1 * \mathtt{ls}\, k1\, j) \vee \\ (\mathtt{ls}\, k2\, \mathsf{nil} * j \mapsto k2 * \mathtt{ls}\, x\, j * \mathtt{ls}\, y\, j) \vee \\ (\mathtt{ls}\, x\, \mathsf{nil} * \mathtt{ls}\, y\, j * j \mapsto k3 * \mathtt{ls}\, k3\, j)\end{array}$$



- We want to prove that the invariant implies termination.

Reversing a "frying-pan" list — the cyclic proof



Endnotes

- James Brotherston, Richard Bornat and Cristiano Calcagno.
  Cyclic proofs of program termination in separation logic.
  To appear in *Proceedings of POPL 2008.*

- James Brotherston.
  Formalised inductive reasoning in the logic of bunched implications.
  In *Proceedings of SAS 2007.*

- James Brotherston and Alex Simpson.
  Complete sequent calculi for induction and infinite descent.
  In *Proceedings of LICS 2007.*

- Josh Berdine, Cristiano Calcagno and Peter OHearn.
  Symbolic execution with separation logic.
  In *Proceedings of APLAS 2005.*

- John C. Reynolds.
  Separation logic: a logic for shared mutable data structures.
  In *Proceedings of LICS 2002.*