Cyclic abduction of inductive safety & termination preconditions

James Brotherston

University College London

LIX Colloquium, Tues 5 Nov 2013

Joint work with Nikos Gorogiannis (Middlesex)

Part I

Introduction and motivations

Introduction

- Classical CS questions: is my program memory-safe, and does it terminate?

Introduction

- Classical CS questions: is my program memory-safe, and does it terminate?
- Refined version: is my program safe and/or terminating, given that it satisfies some precondition?

Introduction

- Classical CS questions: is my program memory-safe, and does it terminate?
- Refined version: is my program safe and/or terminating, given that it satisfies some precondition?
- Even more refined version: can we find a reasonable precondition under which my program is safe and/or terminating?

Introduction

- Classical CS questions: is my program memory-safe, and does it terminate?
- Refined version: is my program safe and/or terminating, given that it satisfies some precondition?
- Even more refined version: can we find a reasonable precondition under which my program is safe and/or terminating?
- In this talk, we focus on this last question, using inductive definitions in separation logic to describe preconditions.

A simple example

Consider the following list traversal program:

$$\texttt{while}\, x \neq \mathsf{nil}\, \texttt{do}\, x = x.next\, \texttt{od};$$

Which preconditions guarantee safe termination?

A simple example

Consider the following list traversal program:

$$\texttt{while}\, x \neq \mathsf{nil}\, \texttt{do}\, x = x.next\, \texttt{od};$$

Which preconditions guarantee safe termination?

$$x = \mathsf{nil}$$

A simple example

Consider the following list traversal program:

$$\texttt{while}\, x \neq \mathsf{nil}\, \texttt{do}\, x = x.next\, \texttt{od};$$

Which preconditions guarantee safe termination?

$$x = \mathsf{nil}$$
$$x \mapsto \mathsf{nil}$$

A simple example

Consider the following list traversal program:

$$\texttt{while}\, x \neq \mathsf{nil}\, \texttt{do}\, x = x.next\, \texttt{od};$$

Which preconditions guarantee safe termination?

$$x = \mathsf{nil}$$
$$x \mapsto \mathsf{nil}$$
$$x \mapsto x' * x' \mapsto \mathsf{nil}$$
$$\vdots$$

A simple example

Consider the following list traversal program:

$$\texttt{while}\, x \neq \mathsf{nil}\, \texttt{do}\, x = x.next\, \texttt{od};$$

Which preconditions guarantee safe termination?

$$x = \mathsf{nil}$$
$$x \mapsto \mathsf{nil}$$
$$x \mapsto x' * x' \mapsto \mathsf{nil}$$
$$\vdots$$

Most general solution is an acyclic linked list, given by

$$\begin{aligned} x = \mathsf{nil} \quad &\Rightarrow \quad \mathsf{list}(x) \\ x \neq \mathsf{nil} * x \mapsto y * \mathsf{list}(y) \quad &\Rightarrow \quad \mathsf{list}(x) \end{aligned}$$

Rôle in automated verification

- A number of automatic verifiers employ separation logic to analyse industrial code (e.g. SpaceInvader, SLAyer)

Rôle in automated verification

- A number of automatic verifiers employ separation logic to analyse industrial code (e.g. SpaceInvader, SLAyer)
- These analysers rely on inductive predicates to describe data structures manipulated by programs (lists, trees etc.)

Rôle in automated verification

- A number of automatic verifiers employ separation logic to analyse industrial code (e.g. SpaceInvader, SLAyer)
- These analysers rely on inductive predicates to describe data structures manipulated by programs (lists, trees etc.)
- Presently, these tools are limited to a few hard-wired such definitions. . .

Rôle in automated verification

- A number of automatic verifiers employ separation logic to analyse industrial code (e.g. SpaceInvader, SLAyer)
- These analysers rely on inductive predicates to describe data structures manipulated by programs (lists, trees etc.)
- Presently, these tools are limited to a few hard-wired such definitions. . .
- . . . which means they must fail, or ask for advice, when encountering a "foreign" data structure.

Rôle in automated verification

- A number of automatic verifiers employ separation logic to analyse industrial code (e.g. SpaceInvader, SLAyer)
- These analysers rely on inductive predicates to describe data structures manipulated by programs (lists, trees etc.)
- Presently, these tools are limited to a few hard-wired such definitions. . .
- . . . which means they must fail, or ask for advice, when encountering a "foreign" data structure.
- It would be nice if we could automatically infer the definitions of these data structures.

Abduction

Proposed by Charles Peirce in the late C19th as a pragmatic process of formulating scientific hypotheses:

Abduction

Proposed by Charles Peirce in the late C19th as a pragmatic process of formulating scientific hypotheses:

> *. . . the hypothesis cannot be admitted, even as a hypothesis, unless it be supposed that it would account for the facts or some of them.*

Abduction

Proposed by Charles Peirce in the late C19th as a pragmatic process of formulating scientific hypotheses:

> *. . . the hypothesis cannot be admitted, even as a hypothesis, unless it be supposed that it would account for the facts or some of them. The form of inference, therefore, is this:*

Abduction

Proposed by Charles Peirce in the late C19th as a pragmatic process of formulating scientific hypotheses:

> *. . . the hypothesis cannot be admitted, even as a hypothesis, unless it be supposed that it would account for the facts or some of them. The form of inference, therefore, is this:*
> *The surprising fact, C, is observed;*

Abduction

Proposed by Charles Peirce in the late C19th as a pragmatic process of formulating scientific hypotheses:

> *...the hypothesis cannot be admitted, even as a hypothesis, unless it be supposed that it would account for the facts or some of them. The form of inference, therefore, is this:*
> *The surprising fact, C, is observed;*
> *But if A were true, C would be a matter of course,*

Abduction

Proposed by Charles Peirce in the late C19th as a pragmatic process of formulating scientific hypotheses:

> *...the hypothesis cannot be admitted, even as a hypothesis, unless it be supposed that it would account for the facts or some of them. The form of inference, therefore, is this:*
> *The surprising fact, C, is observed;*
> *But if A were true, C would be a matter of course,*
> *Hence, there is reason to suspect that A is true.*
>
> *(Peirce, Pragmatism and Abduction, 1903)*

Abduction

Proposed by Charles Peirce in the late C19th as a pragmatic process of formulating scientific hypotheses:

> *. . . the hypothesis cannot be admitted, even as a hypothesis, unless it be supposed that it would account for the facts or some of them. The form of inference, therefore, is this:*
> *The surprising fact, C, is observed;*
> *But if A were true, C would be a matter of course,*
> *Hence, there is reason to suspect that A is true.*
>
> *(Peirce, Pragmatism and Abduction, 1903)*

Our aim is to abduce a precondition or "hypothesis" that would justify the "surprising fact" of program safety / termination.

Overview of our approach

- Our approach builds on the cyclic termination proofs in

  J. Brotherston, R. Bornat and C. Calcagno.
  Cyclic proofs of program termination in separation logic.
  In *Proceedings of POPL*, 2008.

Overview of our approach

- Our approach builds on the cyclic termination proofs in

J. Brotherston, R. Bornat and C. Calcagno.
Cyclic proofs of program termination in separation logic.
In *Proceedings of POPL*, 2008.

- Given a program, we search for a cyclic proof that the program has the desired property.

Overview of our approach

- Our approach builds on the cyclic termination proofs in

J. Brotherston, R. Bornat and C. Calcagno.
Cyclic proofs of program termination in separation logic.
In *Proceedings of POPL*, 2008.

- Given a program, we search for a cyclic proof that the program has the desired property.

- When we inevitably get stuck, we are allowed to abduce (i.e. guess) definitions to help us out.

Overview of our approach

- Our approach builds on the cyclic termination proofs in

  J. Brotherston, R. Bornat and C. Calcagno.
  Cyclic proofs of program termination in separation logic.
  In *Proceedings of POPL*, 2008.

- Given a program, we search for a cyclic proof that the program has the desired property.

- When we inevitably get stuck, we are allowed to abduce (i.e. guess) definitions to help us out.

- We employ lots of heuristics to help the search process.

Overview of our approach

- Our approach builds on the cyclic termination proofs in

  J. Brotherston, R. Bornat and C. Calcagno.
  Cyclic proofs of program termination in separation logic.
  In *Proceedings of POPL*, 2008.

- Given a program, we search for a cyclic proof that the program has the desired property.

- When we inevitably get stuck, we are allowed to abduce (i.e. guess) definitions to help us out.

- We employ lots of heuristics to help the search process.

- Tool, CABER, implemented on top of cyclic theorem prover CYCLIST:

  J. Brotherston, N. Gorogiannis, and R.L. Petersen.
  A generic cyclic theorem prover.
  In *APLAS* 2012.

Part II

Cyclic safety and termination proofs

Syntax of programs

- Expressions are either a variable or nil.

Syntax of programs

- Expressions are either a variable or nil.
- Branching conditions B and commands C are given by

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

Syntax of programs

- Expressions are either a variable or nil.
- Branching conditions B and commands C are given by

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

where E ranges over expressions, x over variables, n over field names and j over $\mathbb{N}$.

Syntax of programs

- Expressions are either a variable or nil.
- Branching conditions B and commands C are given by

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

  where E ranges over expressions, x over variables, n over field names and j over $\mathbb{N}$.
- A program is given by $\texttt{fields}\ n_1, \ldots, n_k;\ C$ where each n_i is a field name and C a command.

Semantics of programs

- A program state is either *fault* or a triple (C, s, h), where
  - C is a command;

Semantics of programs

- A program state is either *fault* or a triple (C, s, h), where
  - C is a command;
  - $s : \mathsf{Var} \to \mathsf{Val}$ is a stack;

Semantics of programs

- A program state is either *fault* or a triple (C, s, h), where
  - C is a command;
  - $s : \mathsf{Var} \to \mathsf{Val}$ is a stack;
  - $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$ is a heap (we write $\circ$ for union of disjoint heaps).

Semantics of programs

- A program state is either *fault* or a triple (C, s, h), where
  - C is a command;
  - $s : \mathsf{Var} \to \mathsf{Val}$ is a stack;
  - $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$ is a heap (we write $\circ$ for union of disjoint heaps).
- (C, s, h) is called safe if there is no computation sequence $(C, s, h) \leadsto^* \mathit{fault}$. And $(C, s, h) \downarrow$ means there is no infinite computation sequence $(C, s, h) \leadsto \ldots$

Semantics of programs

- A program state is either *fault* or a triple (C, s, h), where
  - C is a command;
  - $s : \mathsf{Var} \to \mathsf{Val}$ is a stack;
  - $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$ is a heap (we write $\circ$ for union of disjoint heaps).
- (C, s, h) is called safe if there is no computation sequence $(C, s, h) \rightsquigarrow^* \textit{fault}$. And $(C, s, h)\downarrow$ means there is no infinite computation sequence $(C, s, h) \rightsquigarrow \dots$

Proposition (Safety / termination monotonicity)

If (C, s, h) is safe and $h \circ h'$ defined then $(C, s, h \circ h')$ is safe.
If $(C, s, h)\downarrow$ and $h \circ h'$ defined then $(C, s, h \circ h')\downarrow$.

Preconditions

- Formulas F are given by

$$F \quad ::= \quad E = E \mid E \neq E \mid \mathsf{emp} \mid E \mapsto \mathbf{E} \mid P\mathbf{E} \mid F * F$$

  where P ranges over predicate symbols (of appropriate arity).

Preconditions

- Formulas F are given by

$$F \quad ::= \quad E = E \mid E \neq E \mid \mathsf{emp} \mid E \mapsto \mathbf{E} \mid P\mathbf{E} \mid F * F$$

  where P ranges over predicate symbols (of appropriate arity).

- An inductive rule for predicate P is a rule of the form

$$F \Rightarrow P\mathbf{t}$$

Preconditions

- Formulas F are given by

$$F \quad ::= \quad E = E \mid E \neq E \mid \mathsf{emp} \mid E \mapsto \mathbf{E} \mid P\mathbf{E} \mid F * F$$

  where P ranges over predicate symbols (of appropriate arity).

- An inductive rule for predicate P is a rule of the form

$$F \Rightarrow P\mathbf{t}$$

- Semantics given by standard forcing relation $s, h \models_\Phi F$

Proof rules

- We write proof judgements of the form

$$F \vdash C$$

where F is a formula and C a command.

Proof rules

- We write proof judgements of the form

$$F \vdash C$$

where F is a formula and C a command.

- Symbolic execution rules capture the effect of commands.

Proof rules

- We write proof judgements of the form

$$F \vdash C$$

where F is a formula and C a command.

- Symbolic execution rules capture the effect of commands.

- E.g., if C is $x := E.f; C'$, we have the symbolic execution rule:

$$\frac{x = \mathbf{E}_{\overline{f}}[x'/x] * (F * E \mapsto \mathbf{E})[x'/x] \vdash C'}{F * E \mapsto \mathbf{E} \vdash C} \quad |\mathbf{E}| \geq \overline{f}$$

Proof rules

- We write proof judgements of the form

$$F \vdash C$$

where F is a formula and C a command.

- Symbolic execution rules capture the effect of commands.

- E.g., if C is $x := E.f; C'$, we have the symbolic execution rule:

$$\frac{x = \mathbf{E}_{\overline{f}}[x'/x] * (F * E \mapsto \mathbf{E})[x'/x] \vdash C'}{F * E \mapsto \mathbf{E} \vdash C} \ |\mathbf{E}| \geq \overline{f}$$

(Here, $\overline{f} \in \mathbb{N}$ and $\mathbf{E}_{\overline{f}}$ is the $\overline{f}$th element of $\mathbf{E}$. The variable x' is a fresh variable used to record the "old value" of x.)

Proof rules (contd.)

- We also have logical rules affecting the precondition, e.g.:

$$\frac{F \vdash C}{F * G \vdash C} \quad \Pi' \subseteq \Pi \quad \text{(Frame)}$$

Proof rules (contd.)

- We also have logical rules affecting the precondition, e.g.:

$$\frac{F \vdash C}{F * G \vdash C} \quad \Pi' \subseteq \Pi \quad \text{(Frame)}$$

- The inductive rules for a predicate P determine its unfolding rule.

Proof rules (contd.)

- We also have logical rules affecting the precondition, e.g.:

$$\frac{F \vdash C}{F * G \vdash C} \quad \Pi' \subseteq \Pi \quad \text{(Frame)}$$

- The inductive rules for a predicate P determine its unfolding rule. E.g., define "binary tree" predicate bt by

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

Proof rules (contd.)

- We also have logical rules affecting the precondition, e.g.:

$$\frac{F \vdash C}{F * G \vdash C} \quad \Pi' \subseteq \Pi \quad \text{(Frame)}$$

- The inductive rules for a predicate P determine its unfolding rule. E.g., define "binary tree" predicate bt by

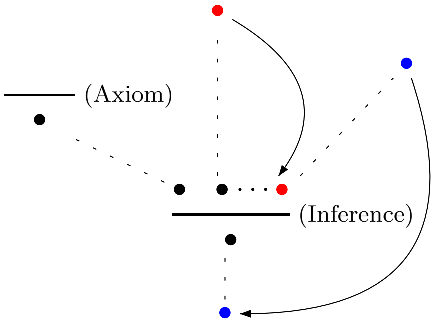
$$\begin{aligned} x = \mathsf{nil} \quad &\Rightarrow \quad \mathsf{bt}(x) \\ x \neq \mathsf{nil} * x \mapsto (y, z) * \mathsf{bt}(y) * \mathsf{bt}(z) \quad &\Rightarrow \quad \mathsf{bt}(x) \end{aligned}$$

  This gives the unfolding rule:

$$\frac{F * u = \mathsf{nil} \vdash C \qquad F * u \neq \mathsf{nil} * u \mapsto (y, z) * \mathsf{bt}(y) * \mathsf{bt}(z) \vdash C}{F * \mathsf{bt}(u) \vdash C}$$

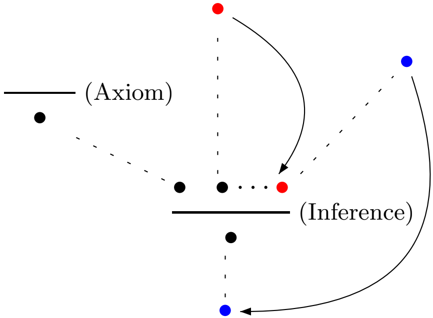
Cyclic proofs

- A cyclic pre-proof is a derivation tree with back-links:



Cyclic proofs

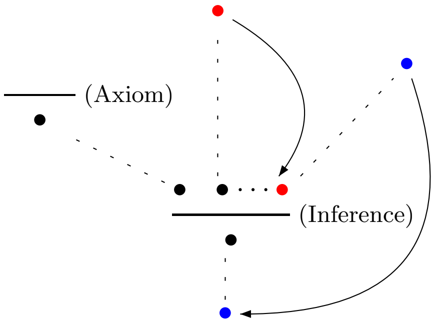
- A cyclic pre-proof is a derivation tree with back-links:



- Safety proof condition: there are infinitely many symbolic executions on every infinite path.

Cyclic proofs

- A cyclic pre-proof is a derivation tree with back-links:



- Safety proof condition: there are infinitely many symbolic executions on every infinite path.
- Termination condition: some inductive predicate is unfolded infinitely often on every infinite path.

Soundness

Theorem

Fix rule set Φ, and program C, and suppose there is a cyclic proof $\mathcal{P}$ of $F \vdash C$. Let stack s and heap h satisfy $s, h \models_\Phi F$.

Soundness

Theorem

Fix rule set Φ, and program C, and suppose there is a cyclic proof $\mathcal{P}$ of $F \vdash C$. Let stack s and heap h satisfy $s, h \models_\Phi F$.

- *If $\mathcal{P}$ satisfies the safety condition, (C, s, h) is safe;*

Soundness

Theorem

Fix rule set Φ, and program C, and suppose there is a cyclic proof $\mathcal{P}$ of $F \vdash C$. Let stack s and heap h satisfy $s, h \models_\Phi F$.

- *If $\mathcal{P}$ satisfies the safety condition, (C, s, h) is safe;*
- *If $\mathcal{P}$ satisfies the termination condition, $(C, s, h) \downarrow$.*

Soundness

Theorem

Fix rule set Φ, and program C, and suppose there is a cyclic proof $\mathcal{P}$ of $F \vdash C$. Let stack s and heap h satisfy $s, h \models_\Phi F$.

- *If $\mathcal{P}$ satisfies the safety condition, (C, s, h) is safe;*
- *If $\mathcal{P}$ satisfies the termination condition, $(C, s, h) \downarrow$.*

Proof.

Inductive argument over proofs. □

Part III

Cyclic abduction

Problem statement

- Initial problem: Given program C with input variables $\mathbf{x}$, find inductive rules Φ such that

$$P\mathbf{x} \vdash C \quad \text{is valid wrt. } \Phi.$$

  where P is a fresh predicate symbol, and "valid" may have either a safety or a termination interpretation.

Problem statement

- Initial problem: Given program C with input variables $\mathbf{x}$, find inductive rules Φ such that

$$P\mathbf{x} \vdash C \quad \text{is valid wrt. } \Phi.$$

  where P is a fresh predicate symbol, and "valid" may have either a safety or a termination interpretation.

- General problem: Given inductive rules Φ and subgoal $F \vdash C$, find inductive rules Φ' such that

$$F \vdash C \quad \text{is valid wrt. } \Phi \cup \Phi'$$

Problem statement

- Initial problem: Given program C with input variables $\mathbf{x}$, find inductive rules Φ such that

$$P\mathbf{x} \vdash C \quad \text{is valid wrt. } \Phi.$$

  where P is a fresh predicate symbol, and "valid" may have either a safety or a termination interpretation.

- General problem: Given inductive rules Φ and subgoal $F \vdash C$, find inductive rules Φ' such that

$$F \vdash C \quad \text{is valid wrt. } \Phi \cup \Phi'$$

- Our approach: search for a cyclic safety/termination proof of $F \vdash C$, inventing inductive rules as necessary.

Principia abductica (I)

Principle I (Proof search priorities)

Priority 1: *apply axiom rule*

Principia abductica (I)

Principle I (Proof search priorities)

Priority 1: *apply axiom rule*

Priority 2: *form backlink*

Principia abductica (I)

Principle I (Proof search priorities)

Priority 1: *apply axiom rule*

Priority 2: *form backlink*

Priority 3: *apply symbolic execution*

Principia abductica (I)

Principle I (Proof search priorities)

Priority 1: *apply axiom rule*

Priority 2: *form backlink*

Priority 3: *apply symbolic execution*

Principle II (Guessing things)

- *In order to serve Priorities 2 and 3 we are allowed to apply logical rules and/or abduce inductive rules.*

Principia abductica (I)

Principle I (Proof search priorities)

Priority 1: *apply axiom rule*

Priority 2: *form backlink*

Priority 3: *apply symbolic execution*

Principle II (Guessing things)

- *In order to serve Priorities 2 and 3 we are allowed to apply logical rules and/or abduce inductive rules.*
- *We may only abduce rules for undefined predicates.*

Principia abductica (I)

Principle I (Proof search priorities)

Priority 1: *apply axiom rule*

Priority 2: *form backlink*

Priority 3: *apply symbolic execution*

Principle II (Guessing things)

- *In order to serve Priorities 2 and 3 we are allowed to apply logical rules and/or abduce inductive rules.*
- *We may only abduce rules for undefined predicates.*
- *When we abduce rules for a predicate P in the current subgoal, we immediately unfold that predicate in the subgoal.*

Principia abductica (I)

Principle I (Proof search priorities)

Priority 1: *apply axiom rule*

Priority 2: *form backlink*

Priority 3: *apply symbolic execution*

Principle II (Guessing things)

- *In order to serve Priorities 2 and 3 we are allowed to apply logical rules and/or abduce inductive rules.*
- *We may only abduce rules for undefined predicates.*
- *When we abduce rules for a predicate P in the current subgoal, we immediately unfold that predicate in the subgoal. (We write $\mathcal{A}(P)$ for a combined abduction-and-unfold step.)*

Principia abductica (II)

When forming back-links, we need to avoid:

- violating the soundness condition on cyclic proofs;

Principia abductica (II)

When forming back-links, we need to avoid:

- violating the soundness condition on cyclic proofs;
- abducing trivially inconsistent definitions like $P\mathbf{x} \Rightarrow P\mathbf{x}$:

$$\frac{P\mathbf{x} \vdash 0}{P\mathbf{x} \vdash 0}\ \mathcal{A}(P)$$

Principia abductica (II)

When forming back-links, we need to avoid:

- violating the soundness condition on cyclic proofs;
- abducing trivially inconsistent definitions like $P\mathbf{x} \Rightarrow P\mathbf{x}$:

$$\dfrac{P\mathbf{x} \vdash 0}{P\mathbf{x} \vdash 0}\,\mathcal{A}(P)$$

Principle III (Avoidance tactic)

We may not form a backlink yielding an infinite path that violates the safety condition, even if searching for a termination proof.

Principia abductica (II)

When forming back-links, we need to avoid:

- violating the soundness condition on cyclic proofs;
- abducing trivially inconsistent definitions like $P\mathbf{x} \Rightarrow P\mathbf{x}$:

$$\frac{P\mathbf{x} \vdash 0}{P\mathbf{x} \vdash 0}\,\mathcal{A}(P)$$

Principle III (Avoidance tactic)

We may not form a backlink yielding an infinite path that violates the safety condition, even if searching for a termination proof.

We can use a model checker to enforce Principle III.

Worked example: binary tree search

$0 :$ `while` $(x \neq \mathsf{nil})\{$
$1 :$ `if`$(\star)$
$2 : x := x.l$
$3 :$ `else`
$4 : x := x.r \quad \}$
$5 : \epsilon$

$$P_0(x) \vdash 0$$

Worked example: binary tree search

$0:$ `while` $(x \neq \mathsf{nil})\{$
$1:$ `if`$(\star)$
$2: x := x.l$
$3:$ `else`
$4: x := x.r \quad \}$
$5: \epsilon$

$$x = \mathsf{nil} * P_1(x) \Rightarrow P_0(x)$$
$$x \neq \mathsf{nil} * P_2(x) \Rightarrow P_0(x)$$

$$P_0(x) \vdash 0$$

Worked example: binary tree search

$0:$ `while` $(x \neq \mathsf{nil})\{$
$1:$ `if`$(\star)$
$2: x := x.l$
$3:$ `else`
$4: x := x.r \quad \}$
$5: \epsilon$

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

$$\frac{x = \mathsf{nil} * P_1(x) \vdash 0 \qquad\qquad x \neq \mathsf{nil} * P_2(x) \vdash 0}{P_0(x) \vdash 0}\ \mathcal{A}(P_0)$$

Worked example: binary tree search

```
0 : while (x ≠ nil){
1 : if(⋆)
2 : x := x.l
3 : else
4 : x := x.r   }
5 : ε
```

$$x = \mathsf{nil} * P_1(x) \Rightarrow P_0(x)$$
$$x \neq \mathsf{nil} * P_2(x) \Rightarrow P_0(x)$$

$$\dfrac{\dfrac{x = \mathsf{nil} * P_1(x) \vdash 5}{x = \mathsf{nil} * P_1(x) \vdash 0}\,\texttt{while} \qquad x \neq \mathsf{nil} * P_2(x) \vdash 0}{P_0(x) \vdash 0}\,\mathcal{A}(P_0)$$

Worked example: binary tree search

0 : `while` $(x \neq \mathsf{nil})\{$
1 : `if`$(\star)$
2 : $x := x.l$
3 : `else`
4 : $x := x.r \quad \}$
5 : ϵ

$$x = \mathsf{nil} * P_1(x) \Rightarrow P_0(x)$$
$$x \neq \mathsf{nil} * P_2(x) \Rightarrow P_0(x)$$

$$\dfrac{\dfrac{\overline{x = \mathsf{nil} * P_1(x) \vdash 5}\;\epsilon}{x = \mathsf{nil} * P_1(x) \vdash 0}\;\texttt{while} \qquad x \neq \mathsf{nil} * P_2(x) \vdash 0}{P_0(x) \vdash 0}\;\mathcal{A}(P_0)$$

Worked example: binary tree search

```
0 : while (x ≠ nil){
1 : if(⋆)
2 : x := x.l
3 : else
4 : x := x.r   }
5 : ε
```

$$x = \mathsf{nil} * P_1(x) \quad \Rightarrow P_0(x)$$
$$x \neq \mathsf{nil} * P_2(x) \quad \Rightarrow P_0(x)$$

$$\dfrac{\dfrac{\dfrac{}{x = \mathsf{nil} * P_1(x) \vdash 5}\,\epsilon}{x = \mathsf{nil} * P_1(x) \vdash 0}\,\texttt{while} \qquad \dfrac{x \neq \mathsf{nil} * P_2(x) \vdash 1}{x \neq \mathsf{nil} * P_2(x) \vdash 0}\,\texttt{while}}{P_0(x) \vdash 0}\,\mathcal{A}(P_0)$$

Worked example: binary tree search

0 : `while` $(x \neq \mathsf{nil})\{$
1 : `if`$(\star)$
2 : $x := x.l$
3 : `else`
4 : $x := x.r \quad \}$
5 : ϵ

$$x = \mathsf{nil} * P_1(x) \Rightarrow P_0(x)$$
$$x \neq \mathsf{nil} * P_2(x) \Rightarrow P_0(x)$$

$$\dfrac{\dfrac{\dfrac{}{x = \mathsf{nil} * P_1(x) \vdash 5}\,\epsilon}{x = \mathsf{nil} * P_1(x) \vdash 0}\,\texttt{while} \qquad \dfrac{\dfrac{x \neq \mathsf{nil} * P_2(x) \vdash 2 \qquad x \neq \mathsf{nil} * P_2(x) \vdash 4}{x \neq \mathsf{nil} * P_2(x) \vdash 1}\,\texttt{if}}{x \neq \mathsf{nil} * P_2(x) \vdash 0}\,\texttt{while}}{P_0(x) \vdash 0}\,\mathcal{A}(P_0)$$

Worked example: binary tree search

$0 :$ `while` $(x \neq \mathsf{nil})\{$
$1 :$ `if`$(\star)$
$2 : x := x.l$
$3 :$ `else`
$4 : x := x.r \quad \}$
$5 : \epsilon$

$$\begin{aligned} x = \mathsf{nil} * P_1(x) &\Rightarrow P_0(x) \\ x \neq \mathsf{nil} * P_2(x) &\Rightarrow P_0(x) \\ x \mapsto (y,z) * P_3(x,y,z) &\Rightarrow P_2(x) \end{aligned}$$

$$\dfrac{\dfrac{\dfrac{}{x = \mathsf{nil} * P_1(x) \vdash 5}\,\epsilon}{x = \mathsf{nil} * P_1(x) \vdash 0}\,\mathtt{while} \qquad \dfrac{\dfrac{x \neq \mathsf{nil} * P_2(x) \vdash 2 \qquad x \neq \mathsf{nil} * P_2(x) \vdash 4}{x \neq \mathsf{nil} * P_2(x) \vdash 1}\,\mathtt{if}}{x \neq \mathsf{nil} * P_2(x) \vdash 0}\,\mathtt{while}}{P_0(x) \vdash 0}\,\mathcal{A}(P_0)$$

Worked example: binary tree search

```
0 : while (x ≠ nil){
1 : if(⋆)
2 : x := x.l
3 : else
4 : x := x.r   }
5 : ε
```

$$
\begin{aligned}
x = \mathsf{nil} * P_1(x) &\Rightarrow P_0(x)\\
x \neq \mathsf{nil} * P_2(x) &\Rightarrow P_0(x)\\
x \mapsto (y,z) * P_3(x,y,z) &\Rightarrow P_2(x)
\end{aligned}
$$

$$
\dfrac{
\dfrac{\dfrac{}{x = \mathsf{nil} * P_1(x) \vdash 5}\,\epsilon}{x = \mathsf{nil} * P_1(x) \vdash 0}\,\texttt{while}
\qquad
\dfrac{\dfrac{\dfrac{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 2}{x \neq \mathsf{nil} * P_2(x) \vdash 2}\,\mathcal{A}(P_2) \qquad x \neq \mathsf{nil} * P_2(x) \vdash 4}{x \neq \mathsf{nil} * P_2(x) \vdash 1}\,\texttt{if}}{x \neq \mathsf{nil} * P_2(x) \vdash 0}\,\texttt{while}
}{P_0(x) \vdash 0}\,\mathcal{A}(P_0)
$$

Worked example: binary tree search

```
0 : while (x ≠ nil){
1 : if(⋆)
2 : x := x.l
3 : else
4 : x := x.r   }
5 : ε
```

$$
\begin{aligned}
x = \mathsf{nil} * P_1(x) &\Rightarrow P_0(x)\\
x \neq \mathsf{nil} * P_2(x) &\Rightarrow P_0(x)\\
x \mapsto (y,z) * P_3(x,y,z) &\Rightarrow P_2(x)
\end{aligned}
$$

$$
\dfrac{
\dfrac{\dfrac{}{x = \mathsf{nil} * P_1(x) \vdash 5}\,\epsilon}{x = \mathsf{nil} * P_1(x) \vdash 0}\,\texttt{while}
\qquad
\dfrac{\dfrac{\dfrac{\begin{array}{l} x \neq \mathsf{nil} * \\ x \mapsto (y,z) * P_3(x,y,z) \end{array} \vdash 2}{x \neq \mathsf{nil} * P_2(x) \vdash 2}\,\mathcal{A}(P_2)
\qquad x \neq \mathsf{nil} * P_2(x) \vdash 4}{x \neq \mathsf{nil} * P_2(x) \vdash 1}\,\texttt{if}}{x \neq \mathsf{nil} * P_2(x) \vdash 0}\,\texttt{while}
}{P_0(x) \vdash 0}\,\mathcal{A}(P_0)
$$

Worked example: binary tree search

$0:$ `while` $(x \neq \mathsf{nil})\{$
$1:$ `if`$(\star)$
$2: x := x.l$
$3:$ `else`
$4: x := x.r \quad \}$
$5: \epsilon$

$$
\begin{aligned}
x = \mathsf{nil} * P_1(x) &\Rightarrow P_0(x)\\
x \neq \mathsf{nil} * P_2(x) &\Rightarrow P_0(x)\\
x \mapsto (y,z) * P_3(x,y,z) &\Rightarrow P_2(x)
\end{aligned}
$$

$$
\dfrac{
  \dfrac{\dfrac{}{x = \mathsf{nil} * P_1(x) \vdash 5}\,\epsilon}{x = \mathsf{nil} * P_1(x) \vdash 0}\,\texttt{while}
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_3(x',x,z) \vdash 0}{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 2}\, x := x.l
      }{x \neq \mathsf{nil} * P_2(x) \vdash 2}\,\mathcal{A}(P_2)
      \qquad
      x \neq \mathsf{nil} * P_2(x) \vdash 4
    }{x \neq \mathsf{nil} * P_2(x) \vdash 1}\,\texttt{if}
  }{x \neq \mathsf{nil} * P_2(x) \vdash 0}\,\texttt{while}
}{P_0(x) \vdash 0}\,\mathcal{A}(P_0)
$$

Worked example: binary tree search

```
0 : while (x ≠ nil){
1 : if(⋆)
2 : x := x.l
3 : else
4 : x := x.r   }
5 : ε
```

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

$$
\dfrac{
  \dfrac{}{x = \mathsf{nil} * P_1(x) \vdash 5}\,\epsilon
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{
        \begin{array}{l} x' \neq \mathsf{nil} * \\ x' \mapsto (x,z) * P_3(x',x,z) \end{array} \vdash 0
      }{
        \begin{array}{l} x \neq \mathsf{nil} * \\ x \mapsto (y,z) * P_3(x,y,z) \end{array} \vdash 2
      }\, x := x.l
    }{
      x \neq \mathsf{nil} * P_2(x) \vdash 2
    }\, \mathcal{A}(P_2)
    \qquad
    x \neq \mathsf{nil} * P_2(x) \vdash 4
  }{
    x \neq \mathsf{nil} * P_2(x) \vdash 1
  }\, \texttt{if}
}{
  \dfrac{x = \mathsf{nil} * P_1(x) \vdash 0 \qquad x \neq \mathsf{nil} * P_2(x) \vdash 0}{P_0(x) \vdash 0}\, \mathcal{A}(P_0)
}\, \texttt{while}
$$

Worked example: binary tree search

0 : `while` $(x \neq \mathsf{nil})\{$
1 : `if`$(\star)$
2 : $x := x.l$
3 : `else`
4 : $x := x.r \quad \}$
5 : ϵ

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

$$
\dfrac{
  \dfrac{
    \dfrac{}{x = \mathsf{nil} * P_1(x) \vdash 5}\,\epsilon
  }{x = \mathsf{nil} * P_1(x) \vdash 0}\,\texttt{while}
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{
            \dfrac{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_0(x) * P_4(x',x,z) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_3(x',x,z) \vdash 0}\,\mathcal{A}(P_3)
          }{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 2}\,x := x.l
        }{x \neq \mathsf{nil} * P_2(x) \vdash 2}\,\mathcal{A}(P_2)
        \qquad
        x \neq \mathsf{nil} * P_2(x) \vdash 4
      }{x \neq \mathsf{nil} * P_2(x) \vdash 1}\,\texttt{if}
    }{x \neq \mathsf{nil} * P_2(x) \vdash 0}\,\texttt{while}
}{P_0(x) \vdash 0}\,\mathcal{A}(P_0)
$$

Worked example: binary tree search

0 : `while` $(x \neq \mathsf{nil})\{$
1 : `if`$(\star)$
2 : $x := x.l$
3 : `else`
4 : $x := x.r \quad \}$
5 : ϵ

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

$$
\dfrac{
  \dfrac{\dfrac{}{x = \mathsf{nil} * P_1(x) \vdash 5}\,\epsilon}{x = \mathsf{nil} * P_1(x) \vdash 0}\,\texttt{while}
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{
            \dfrac{P_0(x) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_0(x) * P_4(x',x,z) \vdash 0}\,(\text{Frame})
          }{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_3(x',x,z) \vdash 0}\,\mathcal{A}(P_3)
        }{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 2}\,x := x.l
      }{x \neq \mathsf{nil} * P_2(x) \vdash 2}\,\mathcal{A}(P_2)
      \qquad
      x \neq \mathsf{nil} * P_2(x) \vdash 4
    }{x \neq \mathsf{nil} * P_2(x) \vdash 1}\,\texttt{if}
  }{x \neq \mathsf{nil} * P_2(x) \vdash 0}\,\texttt{while}
}{P_0(x) \vdash 0}\,\mathcal{A}(P_0)
$$

Worked example: binary tree search

$0:$ `while` $(x \neq \mathsf{nil})\{$
$1:$ `if`$(\star)$
$2: x := x.l$
$3:$ `else`
$4: x := x.r \quad \}$
$5: \epsilon$

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

$$
\dfrac{
\dfrac{x = \mathsf{nil} * P_1(x) \vdash 5}{x = \mathsf{nil} * P_1(x) \vdash 0}\,\epsilon\ \texttt{while}
\qquad
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{P_0(x) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_0(x) * P_4(x',x,z) \vdash 0}\,(\text{Frame})
}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_3(x',x,z) \vdash 0}\,\mathcal{A}(P_3)
}{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 2}\,x := x.l
}{x \neq \mathsf{nil} * P_2(x) \vdash 2}\,\mathcal{A}(P_2)
\quad x \neq \mathsf{nil} * P_2(x) \vdash 4
}{x \neq \mathsf{nil} * P_2(x) \vdash 1}\,\texttt{if}
}{x \neq \mathsf{nil} * P_2(x) \vdash 0}\,\texttt{while}
}{P_0(x) \vdash 0}\,\mathcal{A}(P_0)
$$

Worked example: binary tree search

```
0 : while (x ≠ nil){
1 : if(⋆)
2 : x := x.l
3 : else
4 : x := x.r   }
5 : ε
```

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

$$\frac{P_0(x) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_0(x) * P_4(x',x,z) \vdash 0}\ \text{(Frame)}$$

$$\frac{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_0(x) * P_4(x',x,z) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_3(x',x,z) \vdash 0}\ \mathcal{A}(P_3)$$

$$\frac{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_3(x',x,z) \vdash 0}{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 2}\ x := x.l$$

$$\frac{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 2}{x \neq \mathsf{nil} * P_2(x) \vdash 2}\ \mathcal{A}(P_2) \qquad \frac{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 4}{x \neq \mathsf{nil} * P_2(x) \vdash 4}\ (P_2)$$

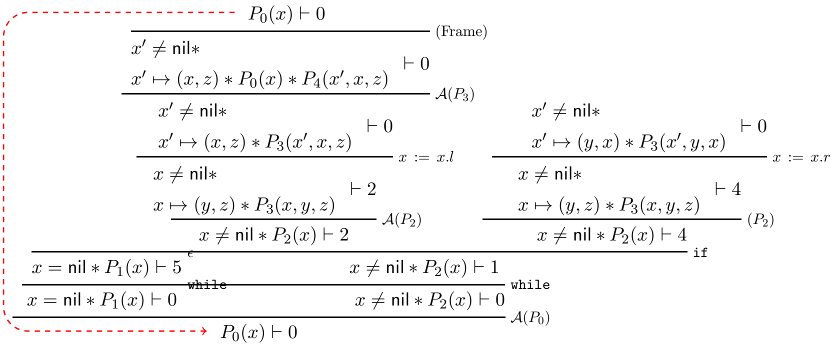
$$\frac{\dfrac{}{x = \mathsf{nil} * P_1(x) \vdash 5}\ \epsilon}{x = \mathsf{nil} * P_1(x) \vdash 0}\ \texttt{while} \qquad \frac{\dfrac{x \neq \mathsf{nil} * P_2(x) \vdash 2 \quad x \neq \mathsf{nil} * P_2(x) \vdash 4}{x \neq \mathsf{nil} * P_2(x) \vdash 1}\ \texttt{if}}{x \neq \mathsf{nil} * P_2(x) \vdash 0}\ \texttt{while}$$

$$\frac{x = \mathsf{nil} * P_1(x) \vdash 0 \qquad x \neq \mathsf{nil} * P_2(x) \vdash 0}{P_0(x) \vdash 0}\ \mathcal{A}(P_0)$$

Worked example: binary tree search

$0:$ `while` $(x \neq \mathsf{nil})\{$
$1:$ `if`$(\star)$
$2: x := x.l$
$3:$ `else`
$4: x := x.r \quad \}$
$5: \epsilon$

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



Worked example: binary tree search

```
0 : while (x ≠ nil){
1 : if(⋆)
2 : x := x.l
3 : else
4 : x := x.r    }
5 : ε
```

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

$$\dfrac{P_0(x) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_0(x) * P_4(x',x,z) \vdash 0}\ \text{(Frame)}$$

$$\dfrac{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_0(x) * P_4(x',x,z) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_3(x',x,z) \vdash 0}\ \mathcal{A}(P_3)$$

$$\dfrac{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_3(x',x,z) \vdash 0}{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 2}\ x := x.l$$

$$\dfrac{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 2}{x \neq \mathsf{nil} * P_2(x) \vdash 2}\ \mathcal{A}(P_2)$$

$$\dfrac{x' \neq \mathsf{nil} * x' \mapsto (y,x) * P_3(x',y,x) \vdash 0}{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 4}\ x := x.r$$

$$\dfrac{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 4}{x \neq \mathsf{nil} * P_2(x) \vdash 4}\ (P_2)$$

$$\dfrac{x \neq \mathsf{nil} * P_2(x) \vdash 2 \qquad x \neq \mathsf{nil} * P_2(x) \vdash 4}{x \neq \mathsf{nil} * P_2(x) \vdash 1}\ \texttt{if}$$

$$\dfrac{}{x = \mathsf{nil} * P_1(x) \vdash 5}\ \epsilon$$

$$\dfrac{x = \mathsf{nil} * P_1(x) \vdash 5}{x = \mathsf{nil} * P_1(x) \vdash 0}\ \texttt{while} \qquad \dfrac{x \neq \mathsf{nil} * P_2(x) \vdash 1}{x \neq \mathsf{nil} * P_2(x) \vdash 0}\ \texttt{while}$$

$$\dfrac{x = \mathsf{nil} * P_1(x) \vdash 0 \qquad x \neq \mathsf{nil} * P_2(x) \vdash 0}{P_0(x) \vdash 0}\ \mathcal{A}(P_0)$$

Worked example: binary tree search

0 : `while` $(x \neq \mathsf{nil})\{$
1 : `if`$(\star)$
2 : $x := x.l$
3 : `else`
4 : $x := x.r \quad \}$
5 : ϵ

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

$$\dfrac{\dfrac{\dfrac{x = \mathsf{nil} * P_1(x) \vdash 5}{x = \mathsf{nil} * P_1(x) \vdash 0}\,\epsilon\ \texttt{while} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P_0(x) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_0(x) * P_4(x',x,z) \vdash 0}\,(\text{Frame})}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_3(x',x,z) \vdash 0}\,\mathcal{A}(P_3)}{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 2}\,x := x.l}{x \neq \mathsf{nil} * P_2(x) \vdash 2}\,\mathcal{A}(P_2) \quad \dfrac{\dfrac{\dfrac{x' \neq \mathsf{nil} * x' \mapsto (y,x) * P_0(y) * P_4(x',y,x) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (y,x) * P_3(x',y,x) \vdash 0}\,(P_3)}{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 4}\,x := x.r}{x \neq \mathsf{nil} * P_2(x) \vdash 4}\,(P_2)}{x \neq \mathsf{nil} * P_2(x) \vdash 1}\,\texttt{if}}{x \neq \mathsf{nil} * P_2(x) \vdash 0}\,\texttt{while}}{P_0(x) \vdash 0}\,\mathcal{A}(P_0)$$

Worked example: binary tree search

$0:\texttt{while}\ (x \neq \mathsf{nil})\{$
$1:\texttt{if}(\star)$
$2: x := x.l$
$3:\texttt{else}$
$4: x := x.r \quad \}$
$5:\epsilon$

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

$$
\dfrac{
\dfrac{\dfrac{}{x = \mathsf{nil} * P_1(x) \vdash 5}\,\epsilon}{x = \mathsf{nil} * P_1(x) \vdash 0}\,\texttt{while}
\quad
\dfrac{\dfrac{
\dfrac{\dfrac{\dfrac{\dfrac{P_0(x) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_0(x) * P_4(x',x,z) \vdash 0}\,(\text{Frame})}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_3(x',x,z) \vdash 0}\,\mathcal{A}(P_3)}{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 2}\,x := x.l}{x \neq \mathsf{nil} * P_2(x) \vdash 2}\,\mathcal{A}(P_2)
\quad
\dfrac{\dfrac{\dfrac{x' \neq \mathsf{nil} * x' \mapsto (y,x) * P_0(y) * P_4(x',y,x) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (y,x) * P_3(x',y,x) \vdash 0}\,(P_3)}{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 4}\,x := x.r}{x \neq \mathsf{nil} * P_2(x) \vdash 4}\,(P_2)
}{x \neq \mathsf{nil} * P_2(x) \vdash 1}\,\texttt{if}}{x \neq \mathsf{nil} * P_2(x) \vdash 0}\,\texttt{while}
}{P_0(x) \vdash 0}\,\mathcal{A}(P_0)
$$

Worked example: binary tree search

$0:$ `while` $(x \neq \mathsf{nil})\{$
$1:$ `if`$(\star)$
$2: x := x.l$
$3:$ `else`
$4: x := x.r \quad \}$
$5: \epsilon$

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

$$\frac{P_0(x) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_0(x) * P_4(x',x,z) \vdash 0}\ (\text{Frame})$$

$$\frac{x' \neq \mathsf{nil} * x' \mapsto (y,x) * P_0(y) * P_0(x) * P_5(x',y,x) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (y,x) * P_0(y) * P_4(x',y,x) \vdash 0}\ \mathcal{A}(P_4)$$

$$\frac{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_0(x) * P_4(x',x,z) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_3(x',x,z) \vdash 0}\ \mathcal{A}(P_3) \qquad \frac{x' \neq \mathsf{nil} * x' \mapsto (y,x) * P_0(y) * P_4(x',y,x) \vdash 0}{x' \neq \mathsf{nil} * x' \mapsto (y,x) * P_3(x',y,x) \vdash 0}\ (P_3)$$

$$\frac{x' \neq \mathsf{nil} * x' \mapsto (x,z) * P_3(x',x,z) \vdash 0}{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 2}\ x := x.l \qquad \frac{x' \neq \mathsf{nil} * x' \mapsto (y,x) * P_3(x',y,x) \vdash 0}{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 4}\ x := x.r$$

$$\frac{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 2}{x \neq \mathsf{nil} * P_2(x) \vdash 2}\ \mathcal{A}(P_2) \qquad \frac{x \neq \mathsf{nil} * x \mapsto (y,z) * P_3(x,y,z) \vdash 4}{x \neq \mathsf{nil} * P_2(x) \vdash 4}\ (P_2)$$

$$\frac{}{x = \mathsf{nil} * P_1(x) \vdash 5}\ \epsilon \qquad \frac{x \neq \mathsf{nil} * P_2(x) \vdash 2 \quad x \neq \mathsf{nil} * P_2(x) \vdash 4}{x \neq \mathsf{nil} * P_2(x) \vdash 1}\ \texttt{if}$$

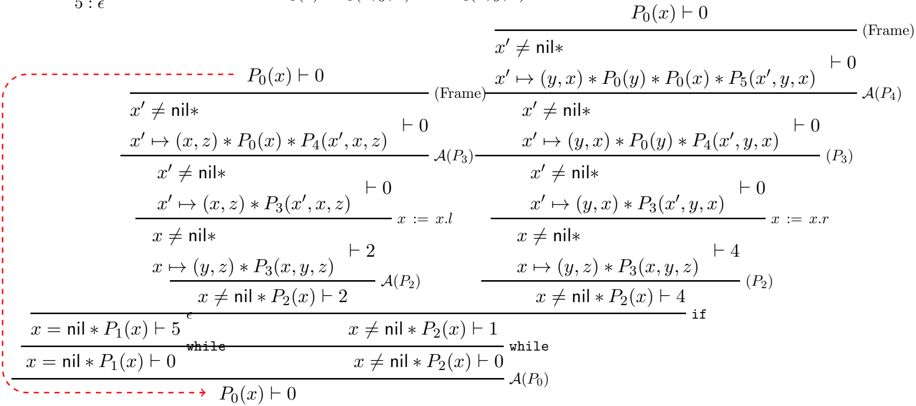
$$\frac{x = \mathsf{nil} * P_1(x) \vdash 5}{x = \mathsf{nil} * P_1(x) \vdash 0}\ \texttt{while} \qquad \frac{x \neq \mathsf{nil} * P_2(x) \vdash 1}{x \neq \mathsf{nil} * P_2(x) \vdash 0}\ \texttt{while}$$

$$\frac{x = \mathsf{nil} * P_1(x) \vdash 0 \quad x \neq \mathsf{nil} * P_2(x) \vdash 0}{P_0(x) \vdash 0}\ \mathcal{A}(P_0)$$

Worked example: binary tree search

$0 : \texttt{while}\ (x \neq \mathsf{nil})\{$
$1 : \texttt{if}(\star)$
$2 : x := x.l$
$3 : \texttt{else}$
$4 : x := x.r \quad \}$
$5 : \epsilon$

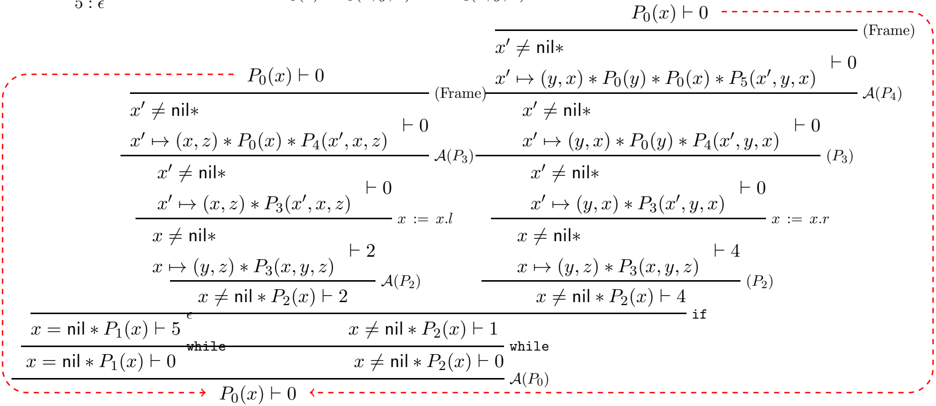
$$
\begin{aligned}
x = \mathsf{nil} * P_1(x) &\Rightarrow P_0(x)\\
x \neq \mathsf{nil} * P_2(x) &\Rightarrow P_0(x)\\
x \mapsto (y,z) * P_3(x,y,z) &\Rightarrow P_2(x)\\
P_0(y) * P_4(x,y,z) &\Rightarrow P_3(x,y,z)\\
P_0(z) * P_5(x,y,z) &\Rightarrow P_4(x,y,z)
\end{aligned}
$$



Worked example: binary tree search

```
0 : while (x ≠ nil){
1 : if(⋆)
2 : x := x.l
3 : else
4 : x := x.r    }
5 : ε
```

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



Simplifying inductive rule sets

$$\begin{array}{rcl} x = \mathsf{nil} : P_1(x) & \Rightarrow & P_0(x) \\ x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\ x \mapsto (y,z) * P_3(x,y,z) & \Rightarrow & P_2(x) \\ P_0(y) * P_4(x,y,z) & \Rightarrow & P_3(x,y,z) \\ P_0(z) * P_5(x,y,z) & \Rightarrow & P_4(x,y,z) \end{array}$$

Simplifying inductive rule sets

- instantiate undefined predicates to emp;

$$\begin{array}{rcl} x = \mathsf{nil} : P_1(x) & \Rightarrow & P_0(x) \\ x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\ x \mapsto (y, z) * P_3(x, y, z) & \Rightarrow & P_2(x) \\ P_0(y) * P_4(x, y, z) & \Rightarrow & P_3(x, y, z) \\ P_0(z) * P_5(x, y, z) & \Rightarrow & P_4(x, y, z) \end{array}$$

Simplifying inductive rule sets

- instantiate undefined predicates to emp;

$$\begin{array}{rcl} x = \mathsf{nil} : P_1(x) & \Rightarrow & P_0(x) \\ x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\ x \mapsto (y,z) * P_3(x,y,z) & \Rightarrow & P_2(x) \\ P_0(y) * P_4(x,y,z) & \Rightarrow & P_3(x,y,z) \\ P_0(z) * P_5(x,y,z) & \Rightarrow & P_4(x,y,z) \end{array} \Longrightarrow \begin{array}{rcl} x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & P_0(x) \\ x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\ x \mapsto (y,z) * P_3(x,y,z) & \Rightarrow & P_2(x) \\ P_0(y) * P_4(x,y,z) & \Rightarrow & P_3(x,y,z) \\ P_0(z) & \Rightarrow & P_4(x,y,z) \end{array}$$

Simplifying inductive rule sets

- instantiate undefined predicates to emp;
- eliminate redundant parameters;

$$
\begin{array}{rcl}
x = \mathsf{nil} : P_1(x) & \Rightarrow & P_0(x) \\
x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\
x \mapsto (y,z) * P_3(x,y,z) & \Rightarrow & P_2(x) \\
P_0(y) * P_4(x,y,z) & \Rightarrow & P_3(x,y,z) \\
P_0(z) * P_5(x,y,z) & \Rightarrow & P_4(x,y,z)
\end{array}
\quad \Longrightarrow \quad
\begin{array}{rcl}
x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & P_0(x) \\
x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\
x \mapsto (y,z) * P_3(x,y,z) & \Rightarrow & P_2(x) \\
P_0(y) * P_4(x,y,z) & \Rightarrow & P_3(x,y,z) \\
P_0(z) & \Rightarrow & P_4(x,y,z)
\end{array}
$$

Simplifying inductive rule sets

- instantiate undefined predicates to emp;
- eliminate redundant parameters;

$$
\begin{array}{rcl}
x = \mathsf{nil} : P_1(x) & \Rightarrow & P_0(x) \\
x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\
x \mapsto (y,z) * P_3(x,y,z) & \Rightarrow & P_2(x) \\
P_0(y) * P_4(x,y,z) & \Rightarrow & P_3(x,y,z) \\
P_0(z) * P_5(x,y,z) & \Rightarrow & P_4(x,y,z)
\end{array}
\quad \Longrightarrow \quad
\begin{array}{rcl}
x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & P_0(x) \\
x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\
x \mapsto (y,z) * P_3(x,y,z) & \Rightarrow & P_2(x) \\
P_0(y) * P_4(x,y,z) & \Rightarrow & P_3(x,y,z) \\
P_0(z) & \Rightarrow & P_4(x,y,z)
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{rcl}
x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & P_0(x) \\
x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\
x \mapsto (y,z) * P_3(x,y) & \Rightarrow & P_2(x) \\
P_0(y) * P_4(z) & \Rightarrow & P_3(x,y) \\
P_0(z) & \Rightarrow & P_4(z)
\end{array}
$$

Simplifying inductive rule sets

- instantiate undefined predicates to emp;
- eliminate redundant parameters;
- inline single-clause predicates.

$$\begin{array}{rcl} x = \mathsf{nil} : P_1(x) & \Rightarrow & P_0(x) \\ x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\ x \mapsto (y,z) * P_3(x,y,z) & \Rightarrow & P_2(x) \\ P_0(y) * P_4(x,y,z) & \Rightarrow & P_3(x,y,z) \\ P_0(z) * P_5(x,y,z) & \Rightarrow & P_4(x,y,z) \end{array} \quad \Longrightarrow \quad \begin{array}{rcl} x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & P_0(x) \\ x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\ x \mapsto (y,z) * P_3(x,y,z) & \Rightarrow & P_2(x) \\ P_0(y) * P_4(x,y,z) & \Rightarrow & P_3(x,y,z) \\ P_0(z) & \Rightarrow & P_4(x,y,z) \end{array}$$

$$\Downarrow$$

$$\begin{array}{rcl} x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & P_0(x) \\ x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\ x \mapsto (y,z) * P_3(x,y) & \Rightarrow & P_2(x) \\ P_0(y) * P_4(z) & \Rightarrow & P_3(x,y) \\ P_0(z) & \Rightarrow & P_4(z) \end{array}$$

Simplifying inductive rule sets

- instantiate undefined predicates to emp;
- eliminate redundant parameters;
- inline single-clause predicates.

$$
\begin{array}{rcl}
x = \mathsf{nil} : P_1(x) & \Rightarrow & P_0(x) \\
x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\
x \mapsto (y,z) * P_3(x,y,z) & \Rightarrow & P_2(x) \\
P_0(y) * P_4(x,y,z) & \Rightarrow & P_3(x,y,z) \\
P_0(z) * P_5(x,y,z) & \Rightarrow & P_4(x,y,z)
\end{array}
\quad \Longrightarrow \quad
\begin{array}{rcl}
x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & P_0(x) \\
x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\
x \mapsto (y,z) * P_3(x,y,z) & \Rightarrow & P_2(x) \\
P_0(y) * P_4(x,y,z) & \Rightarrow & P_3(x,y,z) \\
P_0(z) & \Rightarrow & P_4(x,y,z)
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{rcl}
x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & P_0(x) \\
x \neq \mathsf{nil} : x \mapsto (y,z) * P_0(y) * P_0(z) & \Rightarrow & P_0(x)
\end{array}
\quad \Longleftarrow \quad
\begin{array}{rcl}
x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & P_0(x) \\
x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\
x \mapsto (y,z) * P_3(x,y) & \Rightarrow & P_2(x) \\
P_0(y) * P_4(z) & \Rightarrow & P_3(x,y) \\
P_0(z) & \Rightarrow & P_4(z)
\end{array}
$$

Simplifying inductive rule sets

- instantiate undefined predicates to emp;
- eliminate redundant parameters;
- inline single-clause predicates.

$$\begin{array}{rcl} x = \mathsf{nil} : P_1(x) & \Rightarrow & P_0(x) \\ x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\ x \mapsto (y,z) * P_3(x,y,z) & \Rightarrow & P_2(x) \\ P_0(y) * P_4(x,y,z) & \Rightarrow & P_3(x,y,z) \\ P_0(z) * P_5(x,y,z) & \Rightarrow & P_4(x,y,z) \end{array} \quad \Longrightarrow \quad \begin{array}{rcl} x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & P_0(x) \\ x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\ x \mapsto (y,z) * P_3(x,y,z) & \Rightarrow & P_2(x) \\ P_0(y) * P_4(x,y,z) & \Rightarrow & P_3(x,y,z) \\ P_0(z) & \Rightarrow & P_4(x,y,z) \end{array}$$

$$\Downarrow$$

(nil-terminated binary tree)

$$\begin{array}{rcl} x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & P_0(x) \\ x \neq \mathsf{nil} : x \mapsto (y,z) * P_0(y) * P_0(z) & \Rightarrow & P_0(x) \end{array} \quad \Longleftarrow \quad \begin{array}{rcl} x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & P_0(x) \\ x \neq \mathsf{nil} : P_2(x) & \Rightarrow & P_0(x) \\ x \mapsto (y,z) * P_3(x,y) & \Rightarrow & P_2(x) \\ P_0(y) * P_4(z) & \Rightarrow & P_3(x,y) \\ P_0(z) & \Rightarrow & P_4(z) \end{array}$$

Simplifying inductive rule sets

- instantiate undefined predicates to emp;
- eliminate redundant parameters;
- inline single-clause predicates.
- remove unsatisfiable clauses (not shown)

$$\begin{aligned} x = \mathsf{nil} : P_1(x) &\Rightarrow P_0(x) \\ x \neq \mathsf{nil} : P_2(x) &\Rightarrow P_0(x) \\ x \mapsto (y,z) * P_3(x,y,z) &\Rightarrow P_2(x) \\ P_0(y) * P_4(x,y,z) &\Rightarrow P_3(x,y,z) \\ P_0(z) * P_5(x,y,z) &\Rightarrow P_4(x,y,z) \end{aligned} \quad \Longrightarrow \quad \begin{aligned} x = \mathsf{nil} : \mathsf{emp} &\Rightarrow P_0(x) \\ x \neq \mathsf{nil} : P_2(x) &\Rightarrow P_0(x) \\ x \mapsto (y,z) * P_3(x,y,z) &\Rightarrow P_2(x) \\ P_0(y) * P_4(x,y,z) &\Rightarrow P_3(x,y,z) \\ P_0(z) &\Rightarrow P_4(x,y,z) \end{aligned}$$

$$\Downarrow$$

$$\begin{aligned} x = \mathsf{nil} : \mathsf{emp} &\Rightarrow P_0(x) \\ x \neq \mathsf{nil} : P_2(x) &\Rightarrow P_0(x) \\ x \mapsto (y,z) * P_3(x,y) &\Rightarrow P_2(x) \\ P_0(y) * P_4(z) &\Rightarrow P_3(x,y) \\ P_0(z) &\Rightarrow P_4(z) \end{aligned}$$

$$\Longleftarrow$$

(nil-terminated binary tree)

$$\begin{aligned} x = \mathsf{nil} : \mathsf{emp} &\Rightarrow P_0(x) \\ x \neq \mathsf{nil} : x \mapsto (y,z) * P_0(y) * P_0(z) &\Rightarrow P_0(x) \end{aligned}$$

Part IV

Challenges and subtleties

Evaluating solution quality

- Backtracking search can yield different solutions.

Evaluating solution quality

- Backtracking search can yield different solutions.
- We can decide whether a predicate is satisfiable

Evaluating solution quality

- Backtracking search can yield different solutions.
- We can decide whether a predicate is satisfiable

J. Brotherston, C. Fuhs, N, Gorogiannis and J. Navarro Perez.
A decision procedure for satisfiability of inductive predicates in separation logic.
Submitted.

Evaluating solution quality

- Backtracking search can yield different solutions.
- We can decide whether a predicate is satisfiable

  J. Brotherston, C. Fuhs, N, Gorogiannis and J. Navarro Perez.
  A decision procedure for satisfiability of inductive predicates in separation logic.
  Submitted.

- Comparing predicates via entailment ($\vdash$) is not practical.

Evaluating solution quality

- Backtracking search can yield different solutions.
- We can decide whether a predicate is satisfiable

  J. Brotherston, C. Fuhs, N, Gorogiannis and J. Navarro Perez.
  A decision procedure for satisfiability of inductive predicates in separation logic.
  Submitted.

- Comparing predicates via entailment ($\vdash$) is not practical.
- Currently we use a simple grading scheme for predicate quality.

Evaluating solution quality

- Backtracking search can yield different solutions.
- We can decide whether a predicate is satisfiable

  J. Brotherston, C. Fuhs, N, Gorogiannis and J. Navarro Perez.
  A decision procedure for satisfiability of inductive predicates in separation logic.
  Submitted.

- Comparing predicates via entailment ($\vdash$) is not practical.
- Currently we use a simple grading scheme for predicate quality.
- We can simplify predicates and replay the proof to improve quality, sometimes.

Experimental results

Program	LOC	Time	Depth	Quality	Term.
List traverse	3	20	3	A	✓
List insert	14	8	7	B	✓
List copy	12	0	8	B	✓
List append	10	12	5	B	✓
Delete last from list	16	12	9	B	✓
Filter list	21	48	11	C	✓
Dispose list	5	4	5	A	✓
Reverse list	7	8	7	A	✓
Cyclic list traverse	5	4	5	A	✓
Binary tree search	7	8	4	A	✓
Binary tree insert	18	4	7	B	✓
List of lists traverse	7	8	5	B	✓
Traverse even-length list	4	8	4	A	✓
Traverse odd-length list	4	4	4	A	✓
Ternary tree search	10	8	5	A	✓
Conditional diverge	3	4	3	B	×
Traverse list of trees	11	12	6	B	✓
Traverse tree of lists	17	68	7	A	✓
Traverse list twice	8	64	9	B	✓

Problem: initial variable assignment

- Consider a local variable assignment $y := x$ at line 0. In the proof we get

$$\frac{y = x * P\mathbf{x} \vdash 1}{P\mathbf{x} \vdash 0}\, y := x$$

Problem: initial variable assignment

- Consider a local variable assignment $y := x$ at line 0. In the proof we get

$$\frac{y = x * P\mathbf{x} \vdash 1}{P\mathbf{x} \vdash 0}\, y := x$$

- The equality $y = x$ might prevent back-links later, so we have to deal with it somehow.

Problem: initial variable assignment

- Consider a local variable assignment $y := x$ at line 0. In the proof we get

$$\frac{y = x * P\mathbf{x} \vdash 1}{P\mathbf{x} \vdash 0}\, y := x$$

- The equality $y = x$ might prevent back-links later, so we have to deal with it somehow.

- But there are lots of choices!

Problem: initial variable assignment

- Consider a local variable assignment $y := x$ at line 0. In the proof we get

$$\frac{y = x * P\mathbf{x} \vdash 1}{P\mathbf{x} \vdash 0}\, y := x$$

- The equality $y = x$ might prevent back-links later, so we have to deal with it somehow.
- But there are lots of choices!
- Currently our standard approach is to generalise P to include y, which helps us abduce e.g. cyclic lists.

Problem: initial variable assignment

- Consider a local variable assignment $y := x$ at line 0. In the proof we get

$$\frac{y = x * P\mathbf{x} \vdash 1}{P\mathbf{x} \vdash 0}\, y := x$$

- The equality $y = x$ might prevent back-links later, so we have to deal with it somehow.

- But there are lots of choices!

- Currently our standard approach is to generalise P to include y, which helps us abduce e.g. cyclic lists.

- In principle, we could also use the control flow graph of the program to help us decide what to do.

Problem: abstraction

- The abstraction problem is inherited from program analysis in general.

Problem: abstraction

- The abstraction problem is inherited from program analysis in general.
- Here it shows up in the need for lemmas:

$$\frac{\Pi : F * \mathsf{list}(x) \vdash i}{\Pi : F * x \mapsto y \vdash i} \quad x \mapsto y \vdash \mathsf{list}(x) \quad \text{(Cut)}$$

Problem: abstraction

- The abstraction problem is inherited from program analysis in general.
- Here it shows up in the need for lemmas:

$$\frac{\Pi : F * \mathsf{list}(x) \vdash i}{\Pi : F * x \mapsto y \vdash i} \quad x \mapsto y \vdash \mathsf{list}(x) \quad \text{(Cut)}$$

- Our tool has a limited abstraction capability, mainly based on existentially quantifying variables modified by loops.

Problem: abstraction

- The abstraction problem is inherited from program analysis in general.
- Here it shows up in the need for lemmas:

$$\frac{\Pi : F * \mathsf{list}(x) \vdash i}{\Pi : F * x \mapsto y \vdash i} \quad x \mapsto y \vdash \mathsf{list}(x) \quad \text{(Cut)}$$

- Our tool has a limited abstraction capability, mainly based on existentially quantifying variables modified by loops.
- Lemma speculation is a well known problem in inductive theorem proving. In our setting, where parts of the lemma may be undefined, it is harder still!

Problem: abstraction

- The abstraction problem is inherited from program analysis in general.
- Here it shows up in the need for lemmas:

$$\dfrac{\Pi : F * \mathsf{list}(x) \vdash i \quad\quad x \mapsto y \vdash \mathsf{list}(x)}{\Pi : F * x \mapsto y \vdash i}\ (\mathrm{Cut})$$

- Our tool has a limited abstraction capability, mainly based on existentially quantifying variables modified by loops.
- Lemma speculation is a well known problem in inductive theorem proving. In our setting, where parts of the lemma may be undefined, it is harder still!
- Cyclist gives us an entailment prover which could be used to prove conjectured lemmas.

Thanks for listening!

Get Caber / Cyclist online (source / virtual machine image):

google "cyclist theorem prover".

J. Brotherston and N, Gorogiannis.
Cyclic abduction of inductive safety and termination preconditions.
Submitted.